

Commenced Publication in 1973

Founding and Former Series Editors:

Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison

Lancaster University, UK

Takeo Kanade

Carnegie Mellon University, Pittsburgh, PA, USA

Josef Kittler

University of Surrey, Guildford, UK

Jon M. Kleinberg

Cornell University, Ithaca, NY, USA

Alfred Kobsa

University of California, Irvine, CA, USA

Friedemann Mattern

ETH Zurich, Switzerland

John C. Mitchell

Stanford University, CA, USA

Moni Naor

Weizmann Institute of Science, Rehovot, Israel

Oscar Nierstrasz

University of Bern, Switzerland

C. Pandu Rangan

Indian Institute of Technology, Madras, India

Bernhard Steffen

University of Dortmund, Germany

Madhu Sudan

Microsoft Research, Cambridge, MA, USA

Demetri Terzopoulos

University of California, Los Angeles, CA, USA

Doug Tygar

University of California, Berkeley, CA, USA

Gerhard Weikum

Max-Planck Institute of Computer Science, Saarbruecken, Germany

Matti Ropo Jan Westerholm
Jack Dongarra (Eds.)

Recent Advances in Parallel Virtual Machine and Message Passing Interface

16th European PVM/MPI Users' Group Meeting
Espoo, Finland, September 7-10, 2009
Proceedings

Volume Editors

Matti Ropo
Jan Westerholm
Åbo Akademi University
Department of Information Technology
20520 Turku, Finland
E-mail: {matti.ropo,jawester}@abo.fi

Jack Dongarra
University of Tennessee
Department of Electrical Engineering and Computer Science
Knoxville, TN 37996-3450, USA
E-mail: dongarra@cs.utk.edu

Library of Congress Control Number: 2009933189

CR Subject Classification (1998): B.2.4, C.2.5, D.1.3, D.3, F.1.2, B.2.1, C.1.2

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-642-03769-0 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-03769-6 Springer Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

springer.com

© Springer-Verlag Berlin Heidelberg 2009
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12743063 06/3180 5 4 3 2 1 0

Preface

The current trend in the state-of-the-art infrastructure for computational science is dominated by two concepts: multicore processors and a massive number of processors in supercomputers. The number of cores in processors has increased rapidly in last few years and this has driven the number of processors in the supercomputers to new scales. As the scale of the supercomputers increases so does the dominance of message passing between processors. The EuroPVM/MPI conference series is the premier research event for high-performance parallel programming in the message-passing paradigm. Applications using parallel message-passing programming, pioneered in this research community, are having significant impact in the areas of computational science, such as bioinformatics, atmospheric science, chemistry, physics, astronomy, medicine, banking and finance, energy, etc.

EuroPVM/MPI is a flagship conference for this community, established as the premier international forum for researchers, users and vendors to present their latest advances in MPI and PVM. EuroPVM/MPI is the forum where fundamental aspects of message passing, implementations, standards, benchmarking, performance and new techniques are presented and discussed by researchers, developers and users from academia and industry.

The 16th European PVM/MPI Users' Group Meeting was held in Espoo during September 7–10, 2009. The conference was organized by the CSC- IT Center for Science and the Department of Information Technology at Åbo Akademi University. The previous conferences were held in Dublin (2008), Paris (2007), Bonn (2006), Sorrento (2005), Budapest (2004), Venice (2003), Linz (2002), Santorini (2001), Balatonfured (2000), Barcelona (1999), Liverpool (1998), Krakow (1997), Munich (1996), Lyon (1995) and Rome (1994).

The main topics of the meeting were message-passing interface (MPI) performance issues in very large systems, MPI program verification and MPI on multi-core architectures.

The Program Committee invited six outstanding researchers to present lectures on different aspects of the message-passing and multithreaded paradigms: Martin Burtscher presented “Real-Time Message Compression in Software,” Richard Graham presented “The MPI 2.2 Standard and the Emerging MPI 3 Standard,” William Gropp presented “MPI at Exascale: Challenges for Data Structures and Algorithms,” Alexey Lastovetsky presented “Model-Based Optimization of MPI Collective Operations for Computational Clusters,” Ewing Lusk presented “Using MPI to Implement Scalable Libraries” and Stephen Siegel “Formal Verification for Scientific Computing: Trends and Progress.”

The conference also included the full-day tutorial on “Practical Formal Verification of MPI and Thread Programs” by Ganesh Gopalakrishnan and Robert

M. Kirby, and the eighth edition of the special session “ParSim 2009—Current Trends in Numerical Simulation for Parallel Engineering Environments”.

In all, 48 full papers were submitted to EuroPVM/MPI. Out of these, 27 were selected for presentation at the conference and 5 as posters. Each submitted paper was assigned to three members of the Program Committee(PC). The review process was smooth and provided a solid base for the Program Chairs to make the final decision for the conference program. The program provided a balanced and interesting view on current developments and trends in the parallel virtual machine and message passing interface areas. Four papers were selected as outstanding contributions to EuroPVM/MPI 2009 and were presented at a special plenary session:

- “The Design of Seamless MPI Computing Environment for Commodity-Based Clusters” by Shinji Sumimoto, Kohta Nakashima, Akira Naruse, Kouichi Kumon, Takashi Yasui, Yoshikazu Kamoshida, Hiroya Matsuba, Atsushi Hori and Yutaka Ishikawa
- “MPI on a Million Processors” by Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, Sameer Kumar, Ewing Lusk, Rajeev Thakur and Jesper Larsson Träff
- “Scalable Detection of MPI-2 Remote Memory Access Inefficiency Patterns” by Marc-André Hermanns, Markus Geimer, Bernd Mohr and Felix Wolf
- “Processing MPI Datatypes Outside MPI” by Robert Ross, Robert Latham, William Gropp, Ewing Lusk and Rajeev Thakur

The Program and General Chairs would like to sincerely thank everybody who contributed to making EuroPVM/MPI 2009 a stimulating and fertile meeting, be they technical paper or poster authors, PC members, external referees, participants or sponsors. We are particularly grateful to all the members of the PC and the additional reviewer, who ensured the high quality of EuroPVM/MPI 2009 with their careful work.

September 2009

Jan Westerholm
Matti Ropo
Jack Dongarra

Organization

EuroPVM/MPI 2009 was organized by the CSC- IT Center for Science and Department of Information Technology in Åbo Akademi University.

General Chair

Jack J. Dongarra University of Tennessee, Knoxville, USA

Program Chairs

Jan Westerholm Åbo Akademi University, Finland

Program Committee

Richard Barret	ORNL, USA
Gil Bloch	Mellanox, USA
George Bosilca	University of Tennessee, USA
Franck Cappello	INRIA, France
Barbara Chapman	University of Houston, USA
Jean-Christophe Desplat	Irish Centre for High-End Computing (ICHEC), Ireland
Frederic Desprez	INRIA, France
Erik D'Hollander	Ghent University, Belgium
Edgar Gabriel	University of Houston, USA
Al Geist	Oak Ridge National Laboratory, USA
Patrick Geoffray	Myricom, Inc., USA
Michael Gerndt	Technische Universität München, Germany
Ganesh Gopalakrishnan	School of Computing, USA
Sergei Gorlatch	Universität Münster, Germany
Andrzej Goscinski	Deakin University, Australia
Richard L.Graham	Oak Ridge National Laboratory ,USA
William Gropp	Argonne National Laboratory, USA
Thomas Herault	Université Paris-Sud / INRIA, France
Torsten Hoefler	Indiana University, USA
Yutaka Ishikawa	University of Tokyo, Japan
Tahar Kechadi	University College Dublin, Ireland
Rainer Keller	HLRS, Germany
Stefan Lankes	RWTH Aachen University, Germany
Alexey Lastovetsky	University College Dublin, Ireland
Ewing Rusty Lusk	Argonne National Laboratory, USA

VIII Organization

Tomas Margalef
Jean-François Méhaut
Bernd Mohr
Raymond Namyst
Rolf Rabenseifner
Casiano Rodriguez-Leon
Martin Schulz

Stephen F. Siegel
Jeffrey Squyres
Rajeev Thakur
Carsten Trinitis
Jesper Larsson Träff

Roland Wismueller
Joachim Worringen

Catedràtic d'Universitat, Spain
Laboratoire LIG, France
Juelich Supercomputing Center, Germany
INRIA, France
HLRS, University of Stuttgart, Germany
Universidad de La Laguna, Spain
Lawerence Livermore National Laboratory,
USA
University of Delaware, USA
Cisco, inc., USA
Argonne National Laboratory, USA
TU München, Germany
NEC Laboratories Europe, NEC Europe Ltd.,
Germany
University of Siegen, Germany
Dolphin ICS, Germany

External Referees

Martin Swany

Local Organizers

Juha Fagerholm
Jussi Heikonen
Tiina Leiponen
Per Öster
Matti Ropo

CSC- IT Center for Science, Finland
Åbo Akademi University, Finland

Table of Contents

Invited Talks (Abstracts)

Real-Time Message Compression in Software	1
<i>Martin Burtscher</i>	
The MPI 2.2 Standard and the Emerging MPI 3 Standard	2
<i>Richard L. Graham</i>	
MPI at Exascale: Challenges for Data Structures and Algorithms	3
<i>William Gropp</i>	
Model-Based Optimization of MPI Collective Operations for Computational Clusters	4
<i>Alexey Lastovetsky</i>	
Using MPI to Implement Scalable Libraries	6
<i>Ewing Lusk</i>	
Formal Verification for Scientific Computing: Trends and Progress	7
<i>Stephen F. Siegel</i>	

Tutorial

Practical Formal Verification of MPI and Thread Programs (Abstract)	8
<i>Ganesh Gopalakrishnan and Robert M. Kirby</i>	

Outstanding Papers

The Design of Seamless MPI Computing Environment for Commodity-Based Clusters	9
<i>Shinji Sumimoto, Kohta Nakashima, Akira Naruse, Kouichi Kumon, Takashi Yasui, Yoshikazu Kamoshida, Hiroya Matsuba, Atsushi Hori, and Yutaka Ishikawa</i>	
MPI on a Million Processors	20
<i>Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, Sameer Kumar, Ewing Lusk, Rajeev Thakur, and Jesper Larsson Träff</i>	
Scalable Detection of MPI-2 Remote Memory Access Inefficiency Patterns	31
<i>Marc-André Hermanns, Markus Geimer, Bernd Mohr, and Felix Wolf</i>	

Processing MPI Datatypes Outside MPI	42
<i>Robert Ross, Robert Latham, William Gropp, Ewing Lusk, and Rajeev Thakur</i>	

Applications

Fine-Grained Data Distribution Operations for Particle Codes	54
<i>Michael Hofmann and Gudula Rünger</i>	
Experiences Running a Parallel Answer Set Solver on Blue Gene	64
<i>Lars Schneidenbach, Bettina Schnor, Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub</i>	

Fault Tolerance

Challenges and Issues of the Integration of RADIC into Open MPI	73
<i>Leonardo Fialho, Guna Santos, Angelo Duarte, Dolores Rexachs, and Emilio Luque</i>	
In-Memory Checkpointing for MPI Programs by XOR-Based Double-Erasure Codes	84
<i>Gang Wang, Xiaoguang Liu, Ang Li, and Fan Zhang</i>	

Library Internals

MPC-MPI: An MPI Implementation Reducing the Overall Memory Consumption	94
<i>Marc Pérache, Patrick Carribault, and Hervé Jourdren</i>	
Towards an Efficient Process Placement Policy for MPI Applications in Multicore Environments	104
<i>Guillaume Mercier and Jérôme Clet-Ortega</i>	

Dynamic Communicators in MPI	116
<i>Richard L. Graham and Rainer Keller</i>	

VolpexMPI: An MPI Library for Execution of Parallel Applications on Volatile Nodes	124
<i>Troy LeBlanc, Rakhi Anand, Edgar Gabriel, and Jaspal Subhlok</i>	

MPI I/O

Using Non-blocking I/O Operations in High Performance Computing to Reduce Execution Times	134
<i>David Buettner, Julian Kunkel, and Thomas Ludwig</i>	

Conflict Detection Algorithm to Minimize Locking for MPI-IO Atomicity	143
<i>Saba Sehrish, Jun Wang, and Rajeev Thakur</i>	
Exploiting Efficient Transpacking for One-Sided Communication and MPI-IO	154
<i>Faisal Ghias Mir and Jesper Larsson Träff</i>	
Multiple-Level MPI File Write-Back and Prefetching for Blue Gene Systems	164
<i>Javier García Blas, Florin Isailă, J. Carretero, Robert Latham, and Robert Ross</i>	

OpenMP

Performance Evaluation of MPI, UPC and OpenMP on Multicore Architectures	174
<i>Damián A. Mallón, Guillermo L. Taboada, Carlos Teijeiro, Juan Touriño, Basilio B. Fraguela, Andrés Gómez, Ramón Doallo, and J. Carlos Mourino</i>	
Automatic Hybrid MPI+OpenMP Code Generation with <code>l1c</code>	185
<i>Ruymán Reyes, Antonio J. Dorta, Francisco Almeida, and Francisco de Sande</i>	

Performance

Optimizing MPI Runtime Parameter Settings by Using Machine Learning	196
<i>Simone Pellegrini, Jie Wang, Thomas Fahringer, and Hans Moritsch</i>	
CoMPI: Enhancing MPI Based Applications Performance and Scalability Using Run-Time Compression	207
<i>Rosa Filgueira, David E. Singh, Alejandro Calderón, and Jesús Carretero</i>	
A Memory-Efficient Data Redistribution Algorithm	219
<i>Stephen F. Siegel and Andrew R. Siegel</i>	
Impact of Node Level Caching in MPI Job Launch Mechanisms	230
<i>Jaidev K. Sridhar and Dhabaleswar K. Panda</i>	

Programming Paradigms and Collective Operations

Towards Efficient MapReduce Using MPI	240
<i>Torsten Hoefler, Andrew Lumsdaine, and Jack Dongarra</i>	

Process Arrival Pattern and Shared Memory Aware Alltoall on InfiniBand	250
<i>Ying Qian and Ahmad Afshahi</i>	

Verification of MPI Programs

How Formal Dynamic Verification Tools Facilitate Novel Concurrency Visualizations	261
<i>Sriram Aananthakrishnan, Michael DeLisi, Sarvani Vakkalanka, Anh Vo, Ganesh Gopalakrishnan, Robert M. Kirby, and Rajeev Thakur</i>	
Sound and Efficient Dynamic Verification of MPI Programs with Probe Non-determinism	271
<i>Anh Vo, Sarvani Vakkalanka, Jason Williams, Ganesh Gopalakrishnan, Robert M. Kirby, and Rajeev Thakur</i>	
Bringing Reverse Debugging to HPC	282
<i>Chris Gottbrath</i>	

ParSim

8 th International Special Session on Current Trends in Numerical Simulation for Parallel Engineering Environments – New Directions and Work-in-Progress (ParSim 2009)	292
<i>Carsten Trinitis, Michael Bader, and Martin Schulz</i>	
A Parallel Simulator for Mercury (Hg) Porosimetry	294
<i>C.H. Moreno-Montiel, F. Rojas-González, G. Román-Alonso, S. Cordero-Sánchez, M.A. Castro-García, and M. Aguilar-Cornejo</i>	
Dynamic Load Balancing Strategies for Hierarchical p -FEM Solvers	305
<i>Ralf-Peter Mundani, Alexander Düster, Jovana Knežević, Andreas Niggl, and Ernst Rank</i>	
Simulation of Primary Breakup for Diesel Spray with Phase Transition	313
<i>Peng Zeng, Samuel Sarholz, Christian Iwainsky, Bernd Binniger, Norbert Peters, and Marcus Herrmann</i>	

Posters Abstracts

Implementing Reliable Data Structures for MPI Services in High Component Count Systems	321
<i>Justin M. Wozniak, Bryan Jacobs, Robert Latham, Sam Lang, Seung Woo Son, and Robert Ross</i>	

Parallel Dynamic Data Driven Genetic Algorithm for Forest Fire Prediction	323
<i>Mónica Denham, Ana Cortés, and Tomás Margalef</i>	
Hierarchical Collectives in MPICH2	325
<i>Hao Zhu, David Goodell, William Gropp, and Rajeev Thakur</i>	
An MPI-1 Compliant Thread-Based Implementation	327
<i>J.C. Díaz Martín, J.A. Rico Gallego, J.M. Álvarez Llorente, and J.F. Perogil Duque</i>	
Static-Analysis Assisted Dynamic Verification of MPI Waitany Programs (Poster Abstract)	329
<i>Sarvani Vakkalanka, Grzegorz Szubzda, Anh Vo, Ganesh Gopalakrishnan, Robert M. Kirby, and Rajeev Thakur</i>	
Author Index	331

Real-Time Message Compression in Software

Martin Burtscher

The University of Texas at Austin, USA

Many parallel applications spend a substantial amount of their total execution time exchanging data between processes. As a result, much work has been done on enhancing the performance of messaging libraries. However, little attention has been paid to improving the achievable bandwidth of large messages because network utilization is relatively good with large message sizes. Yet, large messages dominate the overall message makeup and communication time in many applications. For example, in simulation codes it is typical to transfer a significant amount of floating-point data between compute nodes after each time step.

Our work focuses on this often overlooked area. In particular, we investigate the idea of employing fast data compression to reduce the communication time of large messages and to improve the overall bandwidth achievable by the system during periods of heavy communication. While compression can reduce the number of bits that need to be transferred, the runtime overhead due to compression has previously been considered prohibitive in high-performance settings. Thus, the challenge for us was to devise an approach that combines a low overhead and high throughput with a good compression ratio so that a net reduction in execution time can be obtained.

To this end, we have developed FPC, a lossless compression algorithm that targets double-precision floating-point data as are typically found in the messages of scientific computations. It employs a value predictor to forecast message entries based on earlier entries. The compression is performed one double at a time by predicting each value and then transmitting only the difference between the predicted and the true value. If the prediction is close to the true value, the difference can be encoded in just a few bits, significantly shrinking the size of the message. Thus, the predictor has to be both fast and accurate.

FPC delivers a good average compression ratio on hard-to-compress numeric data. Moreover, it compresses and decompresses one to two orders of magnitude faster than other algorithms and is, in fact, fast enough to provide real-time message compression and decompression on modern multicore compute nodes. Consequently, software-based data compression can be used to speed up message passing and accelerate high-performance computing clusters.

The MPI 2.2 Standard and the Emerging MPI 3 Standard

Richard L. Graham

Oak Ridge National Laboratory,
USA

The Message Passing Interface (MPI) 2.1 standard (<http://www.mpi-forum.org>) has served the parallel technical and scientific applications community well over the last decade, and has become the ubiquitous communications application programming interface for this community. However, this extensive use has also exposed areas where this standard can be strengthened to meet the challenges of emerging ultra-scale computing. This talk will describe the changes introduced to the standard in latest version of the MPI standard, 2.2, and current work targeting version 3 of this standard.

The latest version of the MPI standard, version 2.2, is the culmination of a year and a half effort to modernize the standard. This includes updating the support for Fortran (2003) and C (99) language specifications, consolidating the standard into a single document, and many textual changes to correct errors, and clarify the standard.

Work on MPI 3 is aiming to modify the standard to better meet the needs of the emerging ultra-scale computer systems, and is expected to introduce large changes to the standard. The addition of support for nonblocking collectives to this standard has already won preliminary approval from the forum, and is awaiting final approval of the full MPI 3 standard. The other areas gaining a lot of attention include (1) adding support for MPI fault tolerance, (2) improved support for remote direct memory access, and (3) better standard support for debugging MPI based applications and extracting performance characteristics of such libraries.

MPI at Exascale: Challenges for Data Structures and Algorithms

William Gropp

University of Illinois at Urbana-Champaign, USA

Petascale computing is here and MPI continues to succeed as a effective, scalable programming model, despite previous predictions that MPI could not work at the Petascale. As the high-performance computing community considers Exascale systems, will MPI continue to be suitable? Already, a number of studies have looked at the behavior of MPI as the number of tasks approaches one million, and have identified scaling problems in the current state-of-the-art. In addition, the MPI Forum has begun to consider the scalability of the *definition* of some of the routines in MPI, based on the sizes of their input arguments. Concerns have also been raised about other routines, based on the scalability of current implementations.

In this talk, I will look at some of the challenges of implementing MPI efficiently on an Exascale system and identify places where new implementation approaches, for both data structures and algorithms, may allow MPI to scale effectively to exascale systems. New hardware capabilities expected on the next generation of Petascale systems as well as on Exascale systems may also offer new options for the efficient implementation of MPI. These results also suggest fundamental limitations in aspects of the current MPI programming model that can guide the evolution of MPI.

Model-Based Optimization of MPI Collective Operations for Computational Clusters

Alexey Lastovetsky

School of Computer Science and Informatics
University College Dublin, Ireland

Computational clusters based on switched networks are widely used by the academic research community for high performance computing, with MPI being the primary programming tool for development of parallel applications. Contribution of communication operations into the execution time can be quite significant for many parallel applications on this platform. Therefore, minimization of the execution time of communication operations is an important optimization technique for high-performance computing on computational clusters.

In theory, analytical communication performance models can play an important role in optimization of communication operations. Their predictions could be used at run-time for selection of the optimal algorithm of one or the other collective operation depending on the message size and the computing nodes involved in the operation. In practice, this approach does not work well when traditional models are employed. The analytical predictions given by these models appear rather inaccurate, often leading to wrong optimization decisions.

In this talk, we analyze the restrictions of the traditional models affecting the accuracy of analytical prediction of the execution time of different algorithms of collective communication operations. The most important restriction is that the constant and variable contributions of processors and network are not fully separated in these models. We show that the full separation of the contributions that have different nature and arise from different sources would lead to more intuitive and accurate models of switched computational clusters. At the same time, we demonstrate that the traditional estimation methods based on point-to-point communication experiments cannot estimate parameters of such models because the total number of independent point-to-point experiments will always be less than the number of parameters. Thus, this is the traditional estimation methods that limit the design of the traditional communication models and make them less intuitive and less accurate than they might be.

We describe a recent solution to this problem, which is a new estimation method based on a set of independent communication experiments including not only point-to-point but also point-to-two operations. We also outline one non-traditional intuitive communication model, the LMO model, detail communication experiments for estimation of its parameters and demonstrate how this model can be used for accurate prediction of the execution time of collective algorithms.

In the second part of the talk, we extend our analysis to computational clusters consisting of heterogeneous processors, probably the most common parallel platform available to the academic research community. We consider and compare three different types of communication performance models that can be used for the heterogeneous

clusters: the traditional (homogeneous) models, heterogeneous extensions of the traditional models, and a heterogeneous extension of the non-traditional LMO model. We demonstrate that while the heterogeneous extensions of the traditional models are more accurate than their original homogeneous versions, they are not accurate enough to compete with the heterogeneous LMO model, which much more accurately predicts the execution time of collective algorithms. We also show that the optimised collective operations based on the heterogeneous LMO model outperform those based on the traditional models, whether homogeneous or heterogeneous.

Using MPI to Implement Scalable Libraries

Ewing Lusk

Mathematics and Computer Science Division
Argonne National Laboratory, USA

MPI is an instantiation of a general-purpose programming model, and high-performance implementations of the MPI standard have provided scalability for a wide range of applications. Ease of use was not an explicit goal of the MPI design process, which emphasized completeness, portability, and performance. Thus it is not surprising that MPI is occasionally criticized for being inconvenient to use and thus a drag on software developer productivity. One approach to the productivity issue is to use MPI to implement simpler programming models. Such models may limit the range of parallel algorithms that can be expressed, yet provide sufficient generality to benefit a significant number of applications, even from different domains. We illustrate this concept with the ADLB (Asynchronous, Dynamic Load-Balancing) library, which can be used to express manager/worker algorithms in such a way that their execution is scalable, even on the largest machines. ADLB makes sophisticated use of MPI functionality while providing an extremely simple API for the application programmer. We will describe it in the context of solving Sudoku puzzles and a nuclear physics Monte Carlo application currently running on tens of thousands of processors.

Formal Verification for Scientific Computing: Trends and Progress

Stephen F. Siegel

Verified Software Laboratory, Department of Computer and Information Sciences,
University of Delaware, Newark, DE 19716, USA, siegel@cis.udel.edu

Science has been transformed by computation. Many observers now consider simulation a “third approach” to scientific discovery, on par with experimentation and theory. But scientists have long-established methods for verifying conclusions based on those traditional approaches—the same is not true for simulation. Reports on simulation-based research often say little about the qualities of the software used, or what was done to ascertain its correctness. Sometimes the software is not even made available to the public; but even when it is, how many would bother to check it, and how would they go about doing so?

Unfortunately, there is substantial evidence that the software used in science is as unreliable as any other type of software. The extensive case studies conducted by Les Hatton, for example, showed that scientific programs are often riddled with undiscovered defects. The most insidious defects do not cause the software to fail catastrophically—they lead to erroneous yet believable results.

Testing is still the most widely-used approach for verifying correctness. Yet Dijkstra warned long ago that testing can only show that a program is wrong, never that a program is right. There are other limitations to testing. First, for nondeterministic programs, including many parallel programs, a correct test result does not even guarantee a program will produce the correct result if run again on the same test. Second, for scientific programs, there is often no way to know what the correct result is supposed to be. Third, testing is expensive—typically consuming 50% of development effort—and in science often requires long run-times and access to expensive hardware.

Into this challenging landscape step *formal methods*, which use techniques grounded in formal logic to reason about program correctness. The ideas go back to the dawn of computer science, but recent advances are now starting to make these approaches practical. *Theorem proving* approaches are the most well-known; they have steadily improved, but still require enormous skill and significant user interaction. More recent *finite-state* techniques (often referred to as *model checking*) trade some of the completeness of theorem-proving for automation and usability. Finite-state verification tools for MPI-based programs include MPI-SPIN and ISP. Finally, new approaches using *symbolic execution* deal with some of the most difficult issues, including reasoning about floating-point computations and determining whether two programs are functionally equivalent.

Like fundamental problems in other scientific disciplines, the verification problem will never have a complete solution. This only makes the problem more interesting, and the steady progress and explosion of new ideas testifies to its continuing importance and fascination.

Practical Formal Verification of MPI and Thread Programs

Ganesh Gopalakrishnan and Robert M. Kirby

University of Utah, USA

Large-scale simulation codes in science and engineering are written using the Message Passing Interface (MPI). Shared memory threads are widely used directly, or to implement higher level programming abstractions. Traditional debugging methods for MPI or thread programs are incapable of providing useful formal guarantees about coverage. They get bogged down in the sheer number of interleavings (schedules), often missing shallow bugs. In this tutorial we will introduce two practical formal verification tools: ISP (for MPI C programs) and Inspect (for Pthread C programs). Unlike other formal verification tools, ISP and Inspect run directly on user source codes (much like a debugger). They pursue only the relevant set of process interleavings, using our own customized Dynamic Partial Order Reduction algorithms. For a given test harness, DPOR allows these tools to guarantee the absence of deadlocks, instrumented MPI object leaks and communication races (using ISP), and shared memory races (using Inspect). ISP and Inspect have been used to verify large pieces of code: in excess of 10,000 lines of MPI/C for ISP in under 5 seconds, and about 5,000 lines of Pthread/C code in a few hours (and much faster with the use of a cluster or by exploiting special cases such as symmetry) for Inspect. We will also demonstrate the Microsoft Visual Studio and Eclipse Parallel Tools Platform integrations of ISP (these will be available on the LiveCD).

The attendees of this tutorial will be given a LiveCD containing ISP and Inspect that they can boot into on their laptops (Win or Mac). In the forenoon session, they will be given a sufficient understanding of practical dynamic analysis methods and DPOR methods to practice them on simple examples. In the afternoon session, they will be able to work through larger examples, and also learn in depth the details of ISP and Inspect algorithms. This is joint work with our students Sarvani Vakkalanka, Yu Yang, Anh Vo, Michael DeLisi, Sriram Aananthakrishnan, Subodh Sharma, Simone Atnezi, Greg Szubzda, Jason William, Alan Humphrey, Chris Derrick, Wei-Fan Chiang, Guodong Li, and Geof Sawaya. The work is supported by Microsoft, NSF CNS 0509379 and CCF-0811429, and SRC Task ID 1847.001.

The Design of Seamless MPI Computing Environment for Commodity-Based Clusters

Shinji Sumimoto¹, Kohta Nakashima¹, Akira Naruse¹, Kouichi Kumon¹,
Takashi Yasui², Yoshikazu Kamoshida³, Hiroya Matsuba³, Atsushi Hori³,
and Yutaka Ishikawa³

¹ Fujitsu Laboratories, Ltd, 4-1-1, Kamikodanaka Nakahara-ku,
Kawasaki, Kanagawa, 211-8588, Japan

² Hitachi Ltd. 1-280 Higashi-Koigakubo,
Kokubunji, Tokyo 185-8601, Japan

³ The University of Tokyo, 2-11-16 Yayoi
Bunkyo-ku, Tokyo, 113-8658, Japan

Abstract. This paper describes the design and implementation of a seamless MPI runtime environment, called MPI-Adapter, that realizes MPI program binary portability in different MPI runtime environments. MPI-Adapter enables an MPI binary program to run on different MPI implementations. It is implemented as a dynamic loadable module so that the module dynamically captures all MPI function calls and invokes functions defined in a different MPI implementation using the data type translation techniques. A prototype system was implemented for Linux PC clusters to evaluate the effectiveness of MPI-Adapter. The results of an evaluation on a Xeon Processor (3.8GHz) based cluster show that the MPI translation overhead of MPI sending (receiving) is around $0.028\mu s$, and the performance degradation of MPI-Adapter is negligibly small on the NAS parallel benchmark IS.

Keywords: MPI ABI Translation, Portable MPI programs, Dynamic linked library, MPICH2, and Open MPI.

1 Introduction

Commodity-based clusters are widely used for high-performance computing in universities, research laboratories, and companies around the world. In particular, more and more computational centers in universities and research laboratories in Japan have been introducing commodity-based clusters as typified by T2K Open Supercomputers[1], Tsubame[2], and the Riken Super Combined Cluster (RSCC)[3] listed in the TOP 500[4].

In such an environment with many large or small-scale clusters connected to the Internet, the users may obtain advanced computational resources through the Internet. For example, small commodity-based clusters at a research laboratory can be used in the parallel program development phase, whereas large commodity-based clusters at computation centers can be used in the production phase. Users could select one of those clusters according to the characteristics

that best suit their needs, for example, large computing resource, shortest waiting time, low execution cost, etc. If the clusters are built on the same CPU architecture and operating system, e.g., the x86 architecture and Linux, they should be able to be used seamlessly in the sense that the user binary programs will be able to run on every machine without recompilation.

The problem in providing such a seamless environment is that MPI runtime environments are not portable. For example, the three T2K open supercomputers, installed at the University of Tsukuba, University of Tokyo, and Kyoto University, have the same hardware specifications and operating system, but use different MPI runtimes from different system vendors. So, for example, an MPI binary program built at the University of Tokyo cannot be executed on the system at Kyoto University.

This issue is due to the MPI standard [5] not defining the application binary interface (ABI). Recent MPI standardization activity[6] are aimed at specifying the MPI ABI, and the ABI will hopefully be defined in a future. However, even when the MPI ABI is defined, it will take a long time before the MPI runtime with a unified ABI can be used on existing commodity-based clusters. Thus, instead of waiting for the unified ABI, it would be better to develop a runtime environment, that translates the ABI of the original MPI execution binary into the ABI of another MPI environment. The Seamless MPI environment that the authors have been developing provides such a seamless environment.

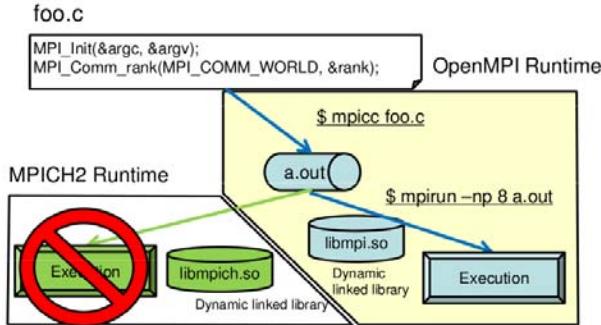
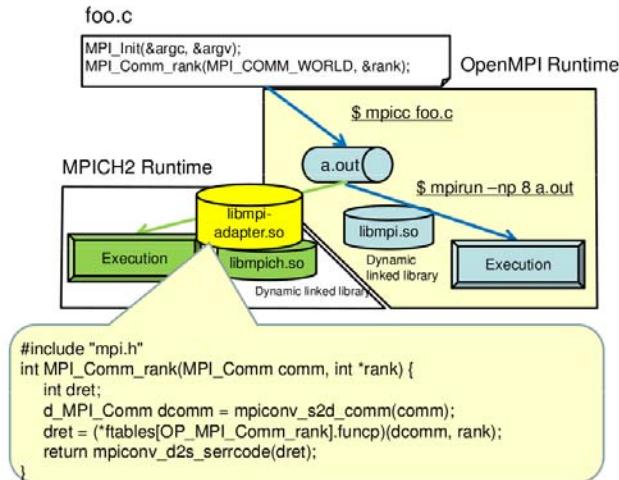
This paper describes the design and implementation of an ABI translating library, called MPI-Adapter, to enable an Seamless MPI environment for commodity-based clusters. The MPI-Adapter has already implemented, which translates the ABI of Open MPI into the ABI of MPICH2. A preliminary evaluation of the MPI-Adapter shows that its translation overhead of MPI-Adapter is $0.028\mu s$ per MPI call, and its performance on the NAS parallel benchmark IS is good. These results prove that the MPI-Adapter can be facilitate the Seamless MPI environment.

This paper is organized as follows: Section 2 presents our motivation and issues regarding the seamless MPI computing environment, and Section 3 presents its design and implementation. The overhead of the MPI-Adapter environment is evaluated in Section 4, and related work is described in Section 5. Section 6 concludes the paper.

2 Seamless MPI Environment

As mentioned in the previous section, an MPI binary program cannot run on any commodity-based clusters because of the lack of an ABI standard. For example, in Figure 1, the binary `a.out` of the `foo.c` program is compiled in the Open MPI environment. Of course, `a.out` runs with `libmpi.so` provided in the Open MPI environment. However, the binary cannot run with `libmpich.so` provided in the MPICH2 environment.

There are at least three use cases when we provide a portable MPI runtime environment to resolve this issue. First, as described in the introduction section,

**Fig. 1.** MPI Runtime Issues**Fig. 2.** MPI-Adapter Processing Image

users may want to migrate from small PC cluster environments to supercomputer center environments without their programs having to be recompiled. Second, the independent software vendors (ISV) do not need to provide many binaries for different MPI implementations and do not need to ask their users which MPI runtime environment is being used; that is, end users usually do not know. The third use case is that users may want to change the MPI implementation without recompilation in a supercomputer center that provides several MPI implementations because of functionality and performance issues.

One possible resolution of these issues is to provide a unified MPI ABI. However, even if the MPI ABI is defined, it will take a long time before MPI runtimes with a unified ABI can be used on existing machines. Therefore, we took the approach of translating the ABI of the original MPI execution binary into the ABI of the running cluster. We have developed MPI-Adapter to translate the MPI ABI.

Figure 2 shows an example of the processing image in MPI-Adapter. A program `foo.c` is compiled and executed in an Open MPI runtime environment (the right side of Figure 2). In the MPI-Adapter environment, the binary compiled on the Open MPI can be executed in the MPICH2 runtime environment by using the translation library `libmpi-adapter.so` (the left side of Figure 2). The library defines all MPI functions so that all MPI function calls are captured. The function defined in the library translates all arguments into representations for the MPICH2 runtime environment. Then, the MPICH2 function defined in the `libmpich.so` is invoked. The return value of the function, which is in the MPICH2 runtime environment, is translated into the value in the Open MPI runtime environment, and the value is returned to the user code.

The detailed design and implementation will be given in Section 3. In the rest of this section, the issues of the data type representations are discussed.

2.1 Representations of Data Types

The MPI standard[5] defines several objects and types in the API, some examples are shown in Table 1. These objects are implemented in MPI runtime systems, and the MPI developer chooses how to implement MPI objects with respect to particular values or data structures. This subsection examines MPI ABIs of Open MPI[7] and MPICH2[8], which are open source MPI runtime environments.

Table 2 shows some ABI differences between Open MPI[7] and MPICH2[8] in the C language. The MPI objects of MPICH2 are defined as 32-bit integer values, whereas those of the Open MPI are defined as pointer values to C data structures. These differences are a problematic in the 64-bit processor case. Moreover, certain pre-defined values such as `MPI_ERR_TRUNCATE` differ between MPICH2 and Open MPI.

Table 3 shows the differences between Open MPI[7] and MPICH2[8] in their MPI ABI implementations in the language FORTRAN. The MPI objects of MPICH2 and Open MPI are defined as 32-bit integer values. However, certain pre-defined values such as `MPI_COMM_WORLD` and `MPI_ERR_TRUNCATE` differ between MPICH2 and Open MPI. Moreover, the FORTRAN definitions of `MPI_SUCCESS` and `MPI_ERR_TRUNCATE` differ between MPICH2 and Open MPI.

Figures 3 and 4 show the differences in the `MPI_Status` structures of Open MPI and MPICH2. These differences mean that the structure of `MPI_Status` must be translated and copied.

Table 1. Objects Defined by MPI Standard Specification

Objects	Types	Objects	Types
Communicator	<code>MPI_Comm</code>	Group	<code>MPI_Group</code>
Request	<code>MPI_Request</code>	Status	<code>MPI_Status</code>
Data type	<code>MPI_Datatype</code>	Operation	<code>MPI_Op</code>
Window	<code>MPI_Win</code>	File	<code>MPI_File</code>
Info	<code>MPI_Info</code>	Difference between Pointers	<code>MPI_Aint</code>
Offset	<code>MPI_Offset</code>	Error Handler	<code>MPI_Errhandler</code>

Table 2. Difference Examples between MPI Runtimes(C Language)

MPI Pre-defined Value	MPICH2	Open MPI
<code>MPI_COMM_WORLD</code>	0x44000000	<code>&ompi_mpi_comm_world</code>
<code>MPI_INT</code>	0x4c000405	<code>&ompi_mpi_int</code>
<code>MPI_INTEGER</code>	0x4c00041b	<code>&ompi_mpi_intger</code>
<code>MPI_SUCCESS</code>	0	0
<code>MPI_ERR_TRUNCATE</code>	14	15

```
struct ompi_status_public_t {
    int MPI_SOURCE;
    int MPI_TAG;
    int MPI_ERROR;
    int _count;
    int _cancelled;
};
```

```
typedef struct MPI_Status {
    int count;
    int cancelled;
    int MPI_SOURCE;
    int MPI_TAG;
    int MPI_ERROR;
} MPI_Status;
```

Fig. 3. MPI_Status Structure(Open MPI)**Fig. 4.** MPI_Status Structure(MPICH2)**Table 3.** Difference Examples between MPI Runtimes(FORTRAN Language)

MPI Pre-defined Values	MPICH2	Open MPI
<code>MPI_COMM_WORLD</code>	0x44000000	0
<code>MPI_INTEGER</code>	0x4c00041b	7
<code>MPI_SUCCESS</code>	0	0
<code>MPI_ERR_TRUNCATE</code>	14	15

The significant work was done by the ABI working group[9] in the MPI forum[6]. The group identified the differences among several MPI ABIs, including vendor MPIS. This work has yielded useful information as follows:

- The Intel MPI, Cray MPI, and Microsoft MPI (MSMPI) are based on the MPICH2 implementation. Therefore, 32-bit integer values are used in their C and FORTRAN MPI runtimes.
- HP MPI is a proprietary implementation; however, like Open MPI, it uses a pointer for MPI objects. The HP MPI also uses 32-bit integer values in FORTRAN.

The examination, so far, would indicate that the MPI-Adapter should be designed as follows:

1. Regarding C, the 64-bit processor environment, which is mainly used in modern clusters, needs a translation mechanism between 64-bit pointers and 32-bit integers. For example, suppose that a program has a variable whose data type is `MPI_Comm`, to keep an `MPI_Comm` object generated by the `MPI_Comm_create` function. The variable only has a 32-bit length in the MPICH environment because the data type is represented by a 32-bit integer

- value. Thus, the variable cannot keep the `MPI_Comm` object in the Open MPI environment.
2. Regarding FORTRAN, only a translation mechanism between 32 bit integers is needed.
 3. The structure of `MPI_Status` must be translated and copied in the case of a different structure.

3 Design and Implementation

3.1 Dynamic Linked Library Based

MPI-Adapter assumes that MPI program binaries are compiled and linked with the dynamic libraries so that we may replace any dynamic library. In the Linux environment, dynamic linked libraries are switched using `LD_LIBRARY_PATH`.

In the most MPI runtimes, all MPI functions are provided by a single dynamic link library, such as the `libmpich.so` file in the MPICH2 runtime. So the MPI-Adapter runtime replaces such a file, which we call the original `libmpi.so` hereafter, with another `limpi.so` for the MPI-Adapter runtime, which we call the `libmpi-adapter.so` hereafter. The `libmpi-adapter.so` defines all MPI functions to translate the original MPI ABI into the target MPI ABI and invoke the MPI functions defined in the `libmpi.so` of the target MPI runtime, which we call the target `libmpi.so` file. The `libmpi-adapter.so` is separated from the target MPI runtime library in order to isolate the same symbols of the target `libmpi.so`.

It should be noted that some runtimes provide additional dynamic library files in addition to the `libmpi.so`, e.g., `libopen-rte.so` and `libopen-pal.so` in the Open MPI runtime. Such extra dynamic library files used in the original one must also be provided in the MPI-Adapter runtime because those files are dynamically loaded at runtime, although they are never used. MPI-Adapter provides those files as dummy files.

In the rest of the paper, we will consider that the original MPI runtime is an Open MPI and the target MPI runtime is MPICH2. In other words, an application has been compiled in the Open MPI environment, and the user wants to run the application in the MPICH2 environment.

When the application program invokes the `MPI_Init` function, the function defined in `libmpi-adapter.so` is invoked, and it processes the following procedures: (1) the target `libmpich.so` file is opened using `dlopen()` because the target MPI runtime is MPICH2; (2) all MPI function addresses are gathered by `dlsym()`; (3) those addresses are stored in a function pointer table called `function_tables`; (4) the target `MPI_Init` function is invoked and the resulting value of the function is converted into the original MPI ABI; and (5) the converted value is returned to the application.

The `MPI_Comm_rank` function, that is an example of a function defined in `libmpi-adapter.so`, is shown as follows:

```
#include "mpi.h" int MPI_Comm_rank(MPI_Comm comm, int *rank) {
    int dret;
    d_MPI_Comm dcomm = mpiconv_s2d_comm(comm);
    dret = (*function_tables[OP_MPI_Comm_rank].funcp)
        (dcomm, rank);
    return mpiconv_d2s_serrcode(dret);
}
```

In this conversion, (1) the `MPI_Comm` data type is converted; (2) the target `MPI_Comm_rank` function, whose address has been registered in the `function_tables`, is invoked; and (3) the return value is converted and the converted value is returned.

There are two issues with the implementation of MPI-Adapter: (1) how do we extract the MPI object and type information in the original and target MPI runtimes? And (2) how do we define the type conversion function for the combination of the original and target MPI runtimes? These issues are resolved in the rest of this section.

3.2 MPI Object Information Collection Tool

To provide conversion functions, each of which converts an MPI object type from the original MPI object definition into the target definition, the object definitions in the MPI header files, i.e., `mpi.h` and `mpif.h` must be collected. MPI-Adapter provides a tool to collect MPI object information using the `mpicc` and `mpif` compilers and provides filter functions from `mpi.h` and `mpif.h`. Figures 5 and 6 show generated examples of Open MPI and MPICH2 definitions in which the Open MPI is the original and MPICH2 is the target MPI runtime.

```
#define s_MPI_COMM_WORLD (&ompi_mpi_comm_world)
#define s_MPI_COMM_NULL (&ompi_mpi_comm_null)
#define s_MPI_COMM_SELF (&ompi_mpi_comm_self)
```

Fig. 5. Samples of Original ABI Definition (Open MPI)

```
#define d_MPI_COMM_WORLD ((d_MPI_Comm)0x44000000)
#define d_MPI_COMM_NULL ((d_MPI_Comm)0x04000000)
#define d_MPI_COMM_SELF ((d_MPI_Comm)0x44000001)
```

Fig. 6. Samples of Target ABI Definition (MPICH2)

3.3 Conversion Skeleton

The Conversion Skeleton defines conversion function skeletons between the original and target MPI ABIs. Here is a sample conversion skeleton, which converts `MPI_Comm` object defined in the original MPI runtime into the one defined in the target MPI runtime.

```
static inline void mpiconv_s2d_comm(d_MPI_Comm *dcomm, s_MPI_Comm
comm) {
    if(comm == s_MPI_COMM_WORLD) *dcomm = d_MPI_COMM_WORLD;
    else if(comm == s_MPI_COMM_NULL) *dcomm = d_MPI_COMM_NULL;
```

```

else if(comm == s_MPI_COMM_SELF) *dcomm = d_MPI_COMM_SELF;
else {
    if(sizeof(s_MPI_Comm) <= sizeof(d_MPI_Comm)) {
        *dcomm = (d_MPI_Comm) comm;
    } else {
        *dcomm = mpiconv_s2d_comm_hash(comm);
    }
}
}

```

In the above skeleton, if the size of the MPI_Comm type in the original MPI runtime, denoted by `s_MPI_Comm`, is not larger than the target MPI runtime, denoted by `d_MPI_Comm`, the original object is used as the target object. Otherwise, the original object is stored in a hash table, and the hash key generated in the hash function is used as the target object.

All conversion functions are generated by using the Conversion Skeleton with the original and target MPI object definitions. The following is an example of code generated using the above definition for the objects shown in Figures 5 and 6.

```

/* automatically generated using skeleton codes */ static inline
void mpiconv_s2d_comm(d_MPI_Comm *dcomm, MPI_Comm comm) {
if(comm == (&ompi_mpi_comm_world)) *dcomm = ((d_MPI_Comm)0x44000000);
else if(comm == (&ompi_mpi_comm_null)) *dcomm = ((d_MPI_Comm)0x04000000);
else if(comm == (&ompi_mpi_comm_self)) *dcomm = ((d_MPI_Comm)0x44000001);
else {
    if(sizeof(MPI_Comm) <= sizeof(d_MPI_Comm)) {
        *((d_MPI_Comm *)dcomm) = (d_MPI_Comm)comm;
    }
    else {
        *((d_MPI_Comm *)dcomm) = mpiconv_s2d_comm_hash(comm);
    }
}
}

```

The C language version of MPI-Adapter, which translates the Open MPI ABI into the MPICH2 ABI, has been implemented and tested. The main translation functions (305 MPI functions) were developed in which the original MPI object is translated into the target. Regarding `MPI_Comm_rank`, the `d_MPI_Comm` is the target `MPI_Comm` object, the `mpiconv_s2d_comm()` translates `MPI_Comm` into the target MPI runtime, and the `mpiconv_d2s_serrcode()` translates the target return code into that of the original.

The IMB(Intel MPI Benchmarks) program worked well and finished without any problems using the current version of MPI-Adapter/Open MPI-MPICH2. `PingPong`, `PingPing`, `Sendrecv`, `Exchange`, `Allreduce`, `Reduce`, `Reduce_scatter`, `Allgather`, `Allgatherv`, `Gather`, `Gatherv`, `Scatter`, `Scatterv`, `Alltoall`, `Alltoallv`, `Bcast`, and `Barrier` functions all worked well on an 8 node (16 processor) cluster including `MPI_Comm_split`.

4 Evaluation of MPI-Adapter

This section evaluates the overhead of the MPI-Adapter. Table 4 describes the evaluation environments. The environments included the MPICH2/SCore[10] cluster; therefore we compared MPICH2/SCore and Open MPI binary using

Table 4. Evaluation Environments

Computation Node	DUAL Xeon 3.8GHz Primargy RX200S2
Ethernet (1Gbps)	Intel E1000 NIC Netgear 48port GigE Switch
OS	CentOS 5.1 x86_64 (2.6.18-8.el5 kernel) SCore7.0

MPI-Adapter and MPICH2/SCore. The communication layer of MPICH2/SCore was PMX/Etherhxb¹[11][10], not TCP/IP socket on Gigabit Ethernet.

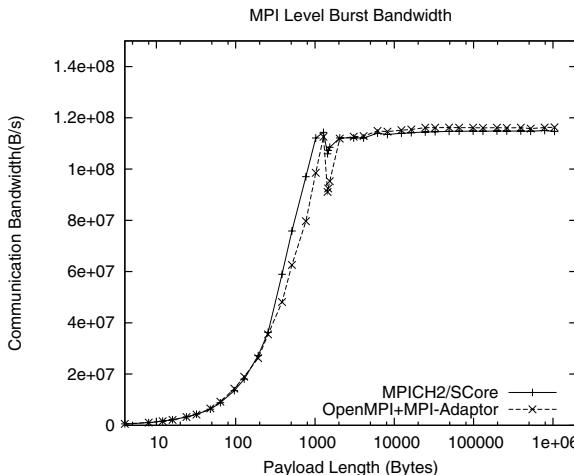
4.1 Evaluation of MPI Communication Performance

To evaluate MPI-Adapter's processing overhead, we measured the MPI round trip latency and MPI burst communication bandwidth and compared the measured values with those of the original MPICH2/SCore.

Table 5 and Figure 7 show the results. Table 5 shows that the processing overhead of MPI-Adapter is only 0.2%, while Figure 7 shows that maximum bandwidth degradation is less than 10%. The MPI ping-pong program uses `MPI_Send()` and `MPI_Recv()` for each side; therefore, the MPI translation overhead is $0.028\mu s$ per

Table 5. MPI Round Trip Time

	RTT	Ratio
MPICH2/SCore	$43.328\mu s$	100.0%
Open MPI+MPI-Adapter	$43.440\mu s$	100.2%

**Fig. 7.** MPI Burst Communication Performance

¹ PMX/Etherhxb is the new version of PM/Ethernet-HXB.

MPI call on the Xeon 3.8-GHz processor. This shows that the overhead is quite small. When the communication network is a $1\mu\text{s}$ latency InfiniBand, the MPI-Adapter overhead becomes around 5%, and it is not too high.

4.2 Evaluation Using NPB

We evaluated the performance differences between the original MPICH2/SCore and target Open MPI with MPI-Adapter and MPICH2/SCore.

Table 6. NPB IS 8 Node

Mops Total	Class A	Class B	Class C
MPICH2/SCore	45.90	52.27	70.20
Open MPI+MPI-Adapter	46.10	49.77	70.02

Figure 6 shows the results; Open MPI+MPI-Adapter is faster than the original MPICH2/SCore for IS Class A, and it is 4.8 % slower for Class B and 0.25 % slower for Class C. The reason for Class A being slower than Class B or Class C is a message timing problem based on message re-transmission.

5 Related Work

The MPI ABI problem can be dealt with by putting a single MPI ABI on multiple MPI runtime systems. The research and standardization activities on this topic are as follows.

- Morph MPI[12] provides unified MPI headers, which means that users must re-compile.
- GMPI[13] also provides a generic mpi header, so in this case too, users must re-compile.
- The ABI working group [14] in the MPI forum [6] is now discussing how to provide unified ABI specifications. However, it will take more time to decide and even more time to implement their findings.

Compared with these studies, MPI-Adapter does not need to re-compile of the programs. This feature can be easily exploited on existing PC clusters, and applied to old MPI binaries without the original source code.

6 Conclusion and Future Work

This paper described the design and implementation of the MPI-Adapter to enable commodity-based clusters to run in an Seamless MPI environment. That is, MPI program binaries compiled in certain MPI runtime environments can be run in another MPI runtime environment without recompilation if the CPU architecture and operating system are the same in both environments including related libraries such as the BLAS numerical library. The MPI-Adapter is

generated by using the translation skeleton with the original and target ABI definitions collected by the MPI object information collection tool. Thus, it is easy to create a new MPI-Adapter for any MPI implementation.

The current implementation of MPI-Adapter translates the ABI of Open MPI into the ABI of MPICH2. The evaluation shows that its translation overhead is $0.028\mu s$ per MPI call, and the performance degradation of MPI-Adapter is negligibly small on the NAS parallel benchmark IS. This proves that MPI-Adapter is an effective way to facilitate Seamless MPI.

The use of MPI-Adapter will make it possible for MPI users to distribute binaries of MPI programs, and it will lead to an effective Seamless MPI environment for commodity-based clusters. We are developing a FORTRAN version of the MPI-Adapter skeleton code. The FORTRAN version will be simpler than the C language version. We will develop several MPI-Adapters and evaluate the Seamless MPI environment at several Japanese supercomputer centers including the University of Tsukuba, University of Tokyo, and Kyoto University. After the evaluation, MPI-Adapter will be released as open source code.

Acknowledgement. This research was partially supported by the “eScience project” of the Ministry of Education, Culture, Sports, Science, and Technology (MEXT), Japan.

References

1. T2K Open Supercomputer Alliance, <http://www.open-supercomputer.org/>
2. GSIC, <http://www.gsic.titech.ac.jp/index.html.en>
3. RSCC: RIKEN Super Combined Cluster System,
<http://w3cic.riken.go.jp/rscc/>
4. Super Computer TOP500, <http://www.top500.org/>
5. The Message Passing Interface (MPI) standard,
<http://www.mpi-forum.org/docs/docs.html>
6. The Message Passing Interface (MPI) Forum, <http://www.mpi-forum.org>
7. OpenMPI, <http://www.open-mpi.org/>
8. MPICH2, <http://www.mcs.anl.gov/research/projects/mpich2/>
9. MPI ABI OpenMPI + MPICH2 + HPMPI + LAMPI + NEC + vendors.xls,
<https://svn.mpi-forum.org/trac/mpi-forum-web/attachment/wiki/abiwikipage/mpi%20abi%20openmpi%20%2b%20mpich2%20%2b%20hpm MPI%20%2b%20lampi%20%2b%20nec%20%2b%20vendors.xls>
10. SCore Cluster System Software, <http://www.pccluster.org/>
11. Sumimoto, S., Ooe, K., Kumon, K., Boku, T., Sato, M., Ukawa, A.: A Scalable Communication Layer for Multi-Dimensional Hyper Crossbar Network Using Multiple Gigabit Ethernet. In: The International Conference on Supercomputing 2006 (ICS 2006). ACM Press, New York (2006)
12. MorphMPI, <http://morphmpi.sourceforge.net/>
13. Gropp, W.: Building library components that can use any mpi implementation. In: Kranzlmüller, D., Kacsuk, P., Dongarra, J., Volkert, J. (eds.) PVM/MPI 2002. LNCS, vol. 2474, pp. 280–287. Springer, Heidelberg (2002)
14. Application Binary Interface Working Group,
<https://svn.mpi-forum.org/trac/mpi-forum-web/wiki/abiwikipage>

MPI on a Million Processors

Pavan Balaji¹, Darius Buntinas¹, David Goodell¹, William Gropp²,
Sameer Kumar³, Ewing Lusk¹, Rajeev Thakur¹, and Jesper Larsson Träff⁴

¹ Argonne National Laboratory, Argonne, IL 60439, USA

² University of Illinois, Urbana, IL, 61801, USA

³ IBM T.J. Watson Research Center, Yorktown Heights, NY 10598, USA

⁴ NEC Laboratories Europe, Sankt Augustin, Germany

Abstract. Petascale machines with close to a million processors will soon be available. Although MPI is the dominant programming model today, some researchers and users wonder (and perhaps even doubt) whether MPI will scale to such large processor counts. In this paper, we examine this issue of how scalable is MPI. We first examine the MPI specification itself and discuss areas with scalability concerns and how they can be overcome. We then investigate issues that an MPI implementation must address to be scalable. We ran some experiments to measure MPI memory consumption at scale on up to 131,072 processes or 80% of the IBM Blue Gene/P system at Argonne National Laboratory. Based on the results, we tuned the MPI implementation to reduce its memory footprint. We also discuss issues in application algorithmic scalability to large process counts and features of MPI that enable the use of other techniques to overcome scalability limitations in applications.

1 Introduction

We are fast approaching an era where the largest supercomputers in the world will have on the order of a million processor cores. MPI is the predominant model for programming the largest parallel machines today. As these machines scale to a million cores, many users and researchers are wondering whether MPI (and applications written in MPI) will scale to that level. In reality, there are multiple aspects to the scalability issue. First, is the MPI *specification* scalable, or are there aspects of the interface that may have issues at large scale? Second, is the MPI *implementation* scalable, and what do implementations need to address to improve their scalability? Third, are the parallel algorithms that MPI applications use themselves scalable to a million processes? We examine these issues in this paper.

Factors affecting scalability include performance and memory consumption. A *nonscalable* MPI function is one whose time or memory consumption *per process* increases linearly (or worse) with the number of processes, all other things being equal. For example, if the time taken by `MPI_Comm_spawn` increases linearly with the number of processes being spawned, it indicates a nonscalable implementation of the function. Similarly, if the memory consumption of `MPI_Comm_dup`

increases linearly with the number of processes, it is not scalable. Such examples of nonscalability need to be identified and fixed, both in the MPI specification and in implementations. A goal should be to use constructs that require only constant space per process.

2 Scalability Issues in the MPI Specification

Although the developers of MPI may not have envisioned million-core systems when MPI was first designed, it was nonetheless intended and designed with scalability in mind. For example, MPI tries to maintain very little global state per process. MPI also defines many operations as collective (called by a group of processes), which enables them to be implemented scalably and efficiently. Nonetheless, examination of the MPI specification reveals that some parts of it may have issues at large scale, particularly with respect to memory consumption.

2.1 Irregular Collectives

Many collectives in MPI have an irregular (or “v”) version that allows users to transfer unequal amounts of data among processes. These collectives take one or more arguments that are arrays of size equal to the number of processes in the communicator, e.g., arrays of counts and displacements in `MPI_Gatherv` and `MPI_Scatterv`. An extreme case is `MPI_Alltoallw`, which takes *six* such arrays as arguments: counts, displacements, and datatypes for both sends and receives. Using such parameters is nonscalable: on a million processes, each array will consume 4 MB on each process.

Irregular collectives are often used in applications because MPI lacks any other way to express communication within a sparse subset of processes in a communicator. For example, in many applications that require nearest-neighbor communication in a Cartesian grid, each process performs an `MPI_Alltoallv` on `MPI_COMM_WORLD` and specifies 0 bytes for all processes other than its neighbors.¹ The PETSc library [13], for example, uses `MPI_Alltoallv` in this manner. While most MPI implementations optimize this pattern by communicating only with processes that have non-zero data, the MPI implementation must still scan through the entire array of data sizes to know which processes have non-zero data, and the user must allocate and initialize this array. On large numbers of processes, the time to read the entire array itself can be large and may increase linearly with system size, even though the number of neighbors a process communicates with remains fixed. Figure 1 shows this effect on an IBM Blue Gene/P for calling `MPI_Alltoallv` with zero-byte messages (no actual communication).

To avoid this problem, some computational libraries, such as PETSc, disable `MPI_Alltoallv`-based communication by default and instead perform direct

¹ This communication cannot be done easily by using subcommunicators because each process may belong to many subcommunicators and the collectives would have to be carefully ordered to avoid deadlocks. Such a scheme would also serialize much of the communication.

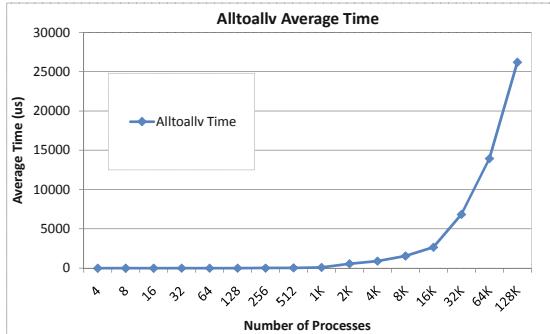


Fig. 1. Zero-byte Alltoallv time on IBM Blue Gene/P (no actual communication)

point-to-point communication among nearest neighbors, which is not as efficient as a concisely represented collective operation could be. The MPI Forum is working on fixing this issue in MPI-3. A proposal for sparse collectives has already been put forth [8].

2.2 Graph Topology

One of the most nonscalable constructs in MPI (memory wise) is the general graph topology. An MPI program can specify the communication pattern in the program as a graph, with edges of the graph representing the communication between nodes. This allows the MPI implementation to optimize communication by appropriate reordering or placement of processes. The problem with the specification is that it requires the *entire* communication graph to be supplied on each process. It therefore requires $O(p + e)$ space per process where e is the number of edges in the graph, and $O(p^2 + pe)$ in total (across all processes). Other limitations of this interface are discussed in [20].

The MPI Forum is currently discussing alternatives in which the graph is specified in a distributed form, requiring only $O(d)$ space per process (where d is the average degree of the communication graph) and $O(p + e)$ space in total across all processes [15]. It is then up to the MPI implementation to work in a distributed fashion and not construct the whole graph for each process.

2.3 One-Sided Communication

Many applications have been shown to benefit from one-sided communication, where a process directly accesses the memory of another process instead of sending and receiving messages. For this reason, MPI-2 also defined an interface for one-sided communication that uses put, get, and accumulate calls and three different synchronization methods. This interface, however, has not been widely used for a number of reasons, the main being that its performance is often worse than regular point-to-point communication. The culprit is often the synchronization associated with one-sided communication. The fence synchronization is

collective across the entire communicator associated with the window object, even if only small subsets of processes communicate with each other. The post-start-complete-wait method synchronizes over a smaller set, but it takes MPI process groups as arguments, which are somewhat cumbersome to create. The third method, lock-unlock, does not need synchronization with the target, but the origin must lock-unlock each target separately and the target does not know when the one-sided operation has completed. Other issues with this interface are discussed in [3].

Solutions to this problem are being considered by the MPI Forum for inclusion in MPI-3 as part of a new and better RMA interface for MPI [12].

2.4 All-to-All Communication

All-to-all communication is not a scalable communication pattern. Each process has a unique data item to send to every other process, which leads to limited opportunities for optimization compared with other collectives. This is not really a problem with the MPI specification but is something applications should be aware of and avoid as far as possible. Avoiding the use of all-to-all may require new algorithms.

2.5 Representation of Process Ranks

One nonscalable aspect of MPI is the explicit representation of process ranks, such as in the group routines `MPI_Group_incl` and `MPI_Group_excl`. While concise representations of collections of processes are possible (for example, some group routines support ranges), the MPI specification encourages the sort of unstructured enumeration that is difficult to scale. Eliminating the explicit enumeration should be considered as an option for large scale.

2.6 Fault Tolerance

On systems with a million cores, the probability of failure or unrecoverable error in some part of the system becomes very high. As a result, greater resilience against failure is needed from all components of the software stack, from low-level system software to libraries and applications. The MPI specification already provides some support to enable users to write programs that are resilient to failure, given appropriate support from the implementation [7]. For example, when a process dies, an implementation, instead of aborting the whole job, can return an error to any other process that tries to communicate with the failed process. It is then up to the application to decide what to do at that point.

However, more support is needed for true fault tolerance. For example, the current set of error classes and codes need to be extended to indicate process failure and other failure modes. The MPI Forum has a subgroup on fault tolerance that is actively working on adding fault-tolerance capabilities to MPI-3 [11]. A number of research efforts in fault-tolerant MPI implementation also exist [4,6,9].

3 MPI Implementation Scalability

In terms of scalability, MPI implementations must pay attention to two aspects as the number of processes is increased: memory consumption of any function and the performance of *all* collective functions (including functions such as `MPI_Init` and `MPI_Comm_split`).

3.1 Process Mappings

MPI communicators usually contain a mapping from MPI process ranks to processor id's. This mapping is usually implemented by an array of p entries (where p is the number of processes in a communicator) for direct, constant-time lookup, possibly with shortcuts for particular mappings. A number of other mappings are often maintained, for instance, to enable fast navigation within and across the nodes of an SMP cluster. Although convenient and very fast, this solution, which requires linear space per process per communicator and quadratic space over the system, is clearly not scalable. Instead, communicators with the same process-to-processor mapping must share mappings. For example, if a communicator is dup'ed with `MPI_Comm_dup`, the new communicator must share the mapping with the original communicator. (At least MPICH2 and vendor implementations derived from it do so.) However, for random communicators, such as those created with `MPI_Comm_split`, mappings cannot be shared.

A solution to this problem is needed. Many mappings can be represented easily by simple linear functions, $ia + b \bmod p$. The identity mapping is often all that is needed for `MPI_COMM_WORLD`. Such linear representations, when possible, can be easily detected and cover many common cases, e.g., subcommunicators that form a consecutive segment from `MPI_COMM_WORLD`. However, this simple mapping covers only a very small fraction of the order of $p!$ possible communicators. More systematic approaches to compact representations of permutations have recently been explored in [2].

3.2 Memory Overheads in Communicator Creation

We ran some experiments on the IBM Blue Gene/P at Argonne to measure the memory overheads of creating new communicators. Figure 2 shows the results of a simple experiment to determine, for different numbers of processes, how many communicators can be created by calling `MPI_Comm_dup` of `MPI_COMM_WORLD` in a loop until it fails. Note that the maximum number of communicators supported by the implementation by default is 8189 (independent of `MPI_Comm_dup`) because of a limit on the number of available context ids.

With the default settings, the number of new communicators that can be created drops sharply starting at about 2048 processes. For 128K processes, the number drops to as low as 264. Although the MPI implementation does not duplicate the process-to-processor mapping in `MPI_Comm_dup`, it allocates some memory for optimizing collective communication. For example, it allocates memory to store “metadata” (such as counts and offsets) needed to optimize

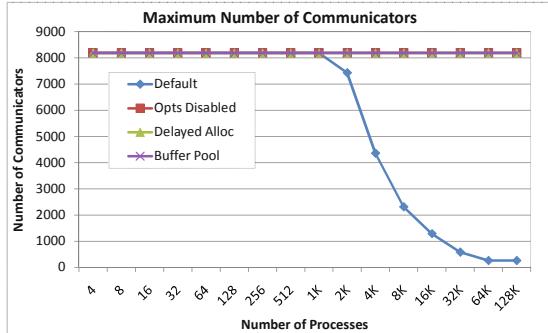


Fig. 2. Maximum number of communicators that can be created with `MPI_Comm_dup` of `MPI_COMM_WORLD` on IBM BG/P for different sizes of `MPI_COMM_WORLD`

`MPI_Alltoall` and its variants. This memory is of size proportional to the number of processes in the communicator. Having such metadata per communicator is useful as it allows different threads to perform collective operations on different communicators in parallel. However, the per-communicator memory usage increases with system size. Since the amount of memory per process is very limited on the Blue Gene/P (512 MB), this optimization also limits the total number of communicators that can be created with `MPI_Comm_dup`.

This scalability problem can be avoided in a number of ways. The simplest way is to use an environment variable to disable collective optimizations (`DCMF_COLLECTIVES=0`), which eliminates the extra memory allocation. However, it has the undesirable impact of reducing the performance of all collectives. Another approach is to use an environment variable (`DCMF_ALLTOALL_PREMALLOC=N`) that delays the allocation of memory until the user actually calls `MPI_Alltoall` on the communicator. This approach helps only those applications that do not perform `MPI_Alltoall`. A third approach that we have implemented is to allocate memory based on the amount of required parallelism rather than the number of communicators. Since the per-communicator metadata buffers are intended to enable different threads to perform collective operations on different communicators in parallel, a fixed number of buffers equal to the maximum number of threads allowed on a node of the Blue Gene/P (four) would be sufficient.

Figure 3 shows the memory consumption in all these cases after 32 calls to `MPI_Comm_dup`. The fixed buffer pool enables all optimizations for all collectives and takes up only a small amount of memory.

3.3 Scalability of `MPI_Init`

Since the performance of `MPI_Init` is rarely measured, implementations may neglect scalability issues in `MPI_Init`. On large numbers of processes, however, a nonscalable implementation of `MPI_Init` may result in `MPI_Init` itself taking several minutes. For example, on connection-oriented networks where a process needs to establish a connection with another process before communication, it is

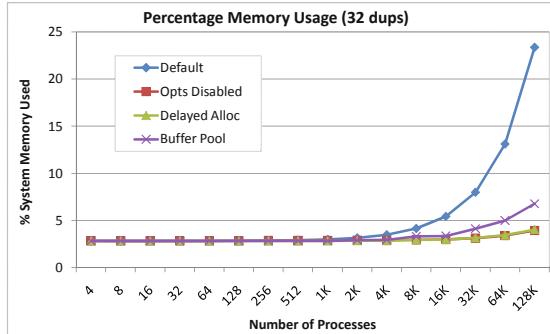


Fig. 3. MPI memory usage on BG/P (after 32 calls to `MPI_Comm_dup` of `MPI_COMM_WORLD`)

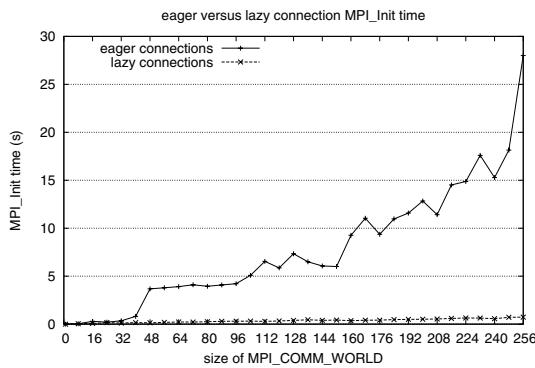


Fig. 4. Duration of `MPI_Init` with eager versus lazy connection establishment. Results are from an eight-core per node cluster using TCP/IP as the communication protocol.

tempting for an MPI implementation to set up all-to-all connections in `MPI_Init` itself. This is however an $O(p^2)$ operation and hence inherently nonscalable. A better approach is to establish no connections in `MPI_Init` and instead establish a connection when a process needs to communicate with another. This method does make the first communication more expensive, but only those connections that are really needed are set up. It also minimizes the number of connections as applications written for scalability are not likely to have communication patterns where all processes directly communicate with all other processes.

Figure 4 shows the time taken by `MPI_Init` on a Linux cluster with TCP when all connections are set up eagerly in `MPI_Init` and when they are set up lazily. The eager method is clearly not scalable.

3.4 Scalable Algorithms for Collective Communication

MPI libraries typically use $O(\log p)$ algorithms (such as a binomial tree) for short-message collectives and $O(m)$ algorithms, where m is the total message

size, for large-message collectives (e.g., a broadcast implemented as a scatter followed by an allgather) [19]. On a million-processor system, we can continue to use $O(\log p)$ algorithms for short messages. For large messages, however, an $O(m)$ broadcast algorithm may not scale, as the message size in the allgather phase will be very small. For example, for a 1 MB broadcast on one million processes, the allgather phase may involve one byte messages. Hybrid algorithms [21] can first do a logarithmic broadcast to a subset of nodes and then a scatter/allgather on many subsets at the same time. Since a million-processor system will likely have several large multicore nodes connected by a direct or switched network, a hybrid collective that first does a network broadcast on the different nodes followed by an intranode broadcast may be a more scalable solution depending on the scalability of the memory hierarchy within a multicore node.

Topology-specific optimizations may also be useful. For easy assembly, most interconnects have smaller diameters than the size of the network ($O(\log p)$ on switched networks and $O(\sqrt[3]{p})$ on 3D torus networks). A pipelined algorithm that streams data on a spanning tree embedded in the network topology will provide more scalable performance because the throughput of the collective is determined by $\frac{\text{message-size}}{\text{diameter}}$. For example, on the BG/P, the six-color torus algorithm can keep 95% of all the links busy during an 8 MB broadcast operation [10].

Global collective acceleration supported by many networks such as Quadrics, InfiniBand, and Blue Gene may be another solution for collectives on MPI_COMM_WORLD. On the Blue Gene/P, for example, broadcast, reduce, allreduce, scatter, scatterv, and allgather collectives take advantage of the combine and broadcast features of the tree network.

4 Enabling Application Scalability

As emerging hardware architectures make greater degrees of parallelism available, even necessary, existing applications are facing the problem of scaling up. The complexity of solving this problem depends entirely on the basic algorithms used by the application, and so no completely general approach will do. In this section, we describe some ways in which features of MPI, perhaps not being used in the current version of a particular application, can play an important role in enabling that application to run effectively on more processors. In many cases, it may be possible to retain most of the existing application code, which is of course extremely desirable from the application's point of view.

4.1 Higher-Dimensional Decompositions with MPI

One relatively straightforward case occurs when the application consists of calculations carried out on a rectangular two- or three-dimensional mesh with nearest-neighbor communication, but the application has parallelized the computation with a one-dimensional decomposition of the mesh. This approach results in contiguous buffers for the MPI sends and receives, which simplifies the application. Straightforward arithmetic shows that as the number of processors and mesh

cells scales up, it becomes more efficient to use a two- or three-dimensional decomposition of the mesh. This results in noncontiguous communication buffers for sending and receiving edge or face data. MPI can help by providing the functions for assembling MPI datatypes that describe these noncontiguous areas of memory. Modern MPI implementations then use particularly efficient algorithms for communicating these areas [18].

4.2 Use of Threads with MPI

In the earlier parts of this paper, we have been treating “MPI on a million processors” as if it meant that the application would see one million separate MPI ranks. This is unlikely to be the case in practice. As the amount of memory per core decreases, applications will be increasingly motivated to use a shared-memory programming model on multicore nodes, while continuing to use MPI for communication among address spaces. MPI supports this transition by having clear semantics for interoperation with threads, based on four levels of thread safety that can be required by an application and provided by an MPI implementation. Although no particular thread system is mentioned in the MPI standard, the MPI specification of levels of thread safety meshes particularly well with the OpenMP standard. This feature has made OpenMP+MPI the current most widely used hybrid programming method [5,16,17].

4.3 Use of MPI-Based Libraries to Hide Complexity

We describe an example of how MPI enables the development of libraries that make it easier to write applications.

One of the most obviously nonscalable approaches to parallel programming is the “manager-worker” algorithm, which achieves good load balancing at the expense of having a single manager process to coordinate the dispensing of work to the worker processes, collection of results, and perhaps addition of new work to the work queue. We recently worked with a Monte Carlo application in nuclear physics [14], which used a variation of this approach and was stuck at about 2000 processors, with the ambition of going to tens of thousands. MPI helped solve this problem by enabling the construction of a general-purpose library called ADLB (Asynchronous Dynamic Load Balancing) [1] that eliminated the single manager as a bottleneck by providing a simple put/get interface to a distributed work queue. The application actually became simpler than before as the MPI communication disappeared; any application process simply puts new work to the queue or retrieves work from it. The ADLB implementation, however, is relatively complex and for scalability and efficiency requires a full range of MPI features, including thread safety, multiple communicators, derived datatypes, asynchronous sends and receives, the “ready” send operation, and other features, all of which are hidden from the application. In this way, MPI supports application scalability while actually simplifying application code.

5 Conclusions

MPI is ready for scaling to a million processors barring a few issues that can be (and are being) fixed. Nonscalable parts of the MPI standard include irregular collectives and virtual graph topology. There is also a need for investigating systematic approaches to compact, adaptive representations of process groups. MPI implementations must pay careful attention to the memory requirements of functions and systematically root out data structures whose size grows linearly with the number of processes. To obtain scalable performance for collective communication, MPI implementations may need to become more topology aware or rely on global collective acceleration support. MPI also provides other features, such as support for building complex libraries and clear semantics for interoperation with threads, that enable applications to use other techniques to scale when limited by memory or data-size constraints.

Acknowledgments

This work was supported in part by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357 and award DE-FG02-08ER25835, and in part by the National Science Foundation award 0837719.

References

1. ADLB library, <http://www.cs.mtsu.edu/~rbutler/adlb/>
2. Barbay, J., Navarro, G.: Compressed representations of permutations, and applications. In: Proc. of 26th Int'l Symposium on Theoretical Aspects of Computer Science (STACS), pp. 111–122 (2009)
3. Bonachea, D., Duell, J.: Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations. In: 2nd Workshop on Hardware-/Software Support for High Performance Sci. and Eng. Computing (2003)
4. Bosilca, G., Bouteiller, A., Cappello, F., Djilali, S., Fedak, G., Germain, C., Herault, T., Lemarinier, P., Lodygensky, O., Magniette, F., Neri, V., Selikhov, A.: MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes. In: Proc. of SC 2002. IEEE, Los Alamitos (2002)
5. Chapman, B., Jost, G., van der Pas, R.: Using OpenMP: Portable Shared Memory Parallel Programming. The MIT Press, Cambridge (2007)
6. Fagg, G.E., Dongarra, J.J.: FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. In: Dongarra, J., Kacsuk, P., Podhorszki, N. (eds.) PVM/MPI 2000. LNCS, vol. 1908, pp. 346–353. Springer, Heidelberg (2000)
7. Gropp, W.D., Lusk, E.: Fault tolerance in MPI programs. Int'l Journal of High Performance Computer Applications 18(3), 363–372 (2004)
8. Hoefler, T., Träff, J.L.: Sparse collective operations for MPI. In: Proc. of 14th Int'l Workshop on High-level Parallel Programming Models and Supportive Environments at IPDPS (2009)

9. Jitsumoto, H., Endo, T., Matsuoka, S.: ABARIS: An adaptable fault detection/recovery component framework for MPIs. In: Proc. of 12th IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems (DPDNS 2007) in conjunction with IPDPS 2007 (March 2007)
10. Kumar, S., Dozsa, G., Berg, J., Cernohous, B., Miller, D., Ratterman, J., Smith, B., Heidelberger, P.: Architecture of the component collective messaging interface. In: Lastovetsky, A., Kechadi, T., Dongarra, J. (eds.) EuroPVM/MPI 2008. LNCS, vol. 5205, pp. 23–32. Springer, Heidelberg (2008)
11. MPI Forum fault tolerance working group,
<https://svn.mpi-forum.org/trac/mpi-forum-web/wiki/FaultToleranceWikiPage>
12. MPI Forum RMA working group,
<https://svn.mpi-forum.org/trac/mpi-forum-web/wiki/RmaWikiPage>
13. PETSc library, <http://www.mcs.anl.gov/petsc>
14. Pieper, S.C., Wiringa, R.B.: Quantum Monte Carlo Calculations of Light Nuclei. *Annu. Rev. Nucl. Part. Sci.* 51, 53 (2001)
15. Proposal for distributed graph topology,
<https://svn.mpi-forum.org/trac/mpi-forum-web/ticket/33>
16. Rabenseifner, R., Hager, G., Jost, G.: Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. In: Proc. of 17th Euromicro Int'l Conference on Parallel, Distributed, and Network-Based Processing (PDP 2009), February 2009, pp. 427–236 (2009)
17. Rane, A., Stanzione, D.: Experiences in tuning performance of hybrid MPI/OpenMP applications on quad-core systems. In: Proc. of 10th LCI Int'l Conference on High-Performance Clustered Computing (March 2009)
18. Ross, R., Miller, N., Gropp, W.: Implementing fast and reusable datatype processing. In: Dongarra, J., Laforenza, D., Orlando, S. (eds.) EuroPVM/MPI 2003. LNCS, vol. 2840, pp. 404–413. Springer, Heidelberg (2003)
19. Thakur, R., Rabenseifner, R., Gropp, W.: Optimization of collective communication operations in MPICH. *Int'l Journal of High-Performance Computing Applications* 19(1), 49–66 (spring 2005)
20. Träff, J.L.: SMP-aware message passing programming. In: Proc. of 8th Int'l Workshop on High-level Parallel Programming Models and Supportive Environments at IPDPS 2003, pp. 56–65 (2003)
21. Träff, J.L.: A simple work-optimal broadcast algorithm for message-passing parallel systems. In: Kranzlmüller, D., Kacsuk, P., Dongarra, J. (eds.) EuroPVM/MPI 2004. LNCS, vol. 3241, pp. 173–180. Springer, Heidelberg (2004)

Scalable Detection of MPI-2 Remote Memory Access Inefficiency Patterns

Marc-André Hermanns¹, Markus Geimer¹, Bernd Mohr¹, and Felix Wolf^{1,2}

¹ Jülich Supercomputing Centre

Forschungszentrum Jülich, Germany

{m.a.hermanns,m.geimer,b.mohr,f.wolf}@fz-juelich.de

² Department of Computer Science

RWTH Aachen University, Germany

Abstract. Wait states in parallel applications can be identified by scanning event traces for characteristic patterns. In our earlier work, we have defined such patterns for MPI-2 one-sided communication, although still based on a trace-analysis scheme with limited scalability. Taking advantage of a new scalable trace-analysis approach based on a parallel replay, which was originally developed for MPI-1 point-to-point and collective communication, we show how wait states in one-sided communications can be detected in a more scalable fashion. We demonstrate the scalability of our method and its usefulness for the optimization cycle with applications running on up to 8,192 cores.

Keywords: MPI-2, remote memory access, performance analysis, scalability, pattern search.

1 Introduction

Remote memory access (RMA) describes the ability of a process to access all or parts of the memory belonging to a remote process directly, without explicit participation of the remote process in the data transfer. Since all parameters for the data transfer are determined by a single process, it is also called one-sided communication. This programming model is made available to the programmer often in the form of platform- or vendor-specific libraries, such as SHMEM (Cray/SGI) or LAPI (IBM). In 1997, one-sided communication was added to the portable MPI standard with version 2 [1], and since then has been adopted by the majority of the available MPI implementations.

Although it has been shown that the use of MPI-2 RMA can improve application performance [2], it has not yet been widely adopted among the MPI user community. But we believe that the availability of suitable programming tools, in particular for performance analysis, can encourage more developers to exploit the benefits of this model. However, since increasing demand for compute power in combination with recent trends in microprocessor design towards multicore chips forces applications to scale to much higher processor counts, such tools must be scalable to be useful.

A non-negligible fraction of the execution time of MPI applications can often be attributed to wait states, which occur when processes fail to reach synchronization points in a timely manner, for example, due to load imbalance. Especially when trying to scale communication-intensive applications to large processor counts, such wait states can

present severe challenges to achieving good performance. In our earlier work [3], we have shown how wait states related to MPI-2 RMA can be identified by searching event traces for characteristic patterns. However, the search algorithm applied was sequential and intended to operate on a single global trace file, offering only limited scalability. In the meantime, we developed a general framework to make pattern search in event traces more scalable [4]. Instead of sequentially analyzing a single global trace file, the framework analyzes multiple process-local trace files in parallel while performing a parallel replay of the target application’s communication behavior. In this paper, we present a synthesis of the two approaches, making the search for wait states in the context of MPI-2 RMA more scalable by enacting a parallel replay of one-sided operations, which had previously only been tried for two-sided and collective operations. The new scalable detection scheme for one-sided communication has been integrated into Scalasca [5], a performance analysis toolset specifically designed for large-scale systems.

The remainder of this paper is organized as follows. Section 2 gives a brief overview of the work done on this topic so far. Afterwards, the semantics of the MPI RMA programming model are explained in Section 3, before specifying the supported MPI RMA inefficiency patterns and their replay-based detection algorithms in Section 4. Moreover, results with two RMA-based applications running on up to 8,192 cores demonstrate the scalability of our method and its usefulness for the optimization cycle in Section 5. Finally, Section 6 concludes the paper and gives a brief outlook on future work.

2 Related Work

The number of portable performance-analysis tools supporting MPI-2 RMA is quite limited. The Paradyn tool, which conducts an automatic on-line bottleneck search, supports several major features of MPI-2 [6]. To analyze RMA operations, it collects process-local statistical data (i.e., transfer counts and time spent in RMA functions). Yet, it does not take inter-process relationships into account. By contrast, the TAU performance system [7] supports profiling and tracing of MPI-2 one-sided communication, though only by monitoring the entry and exit of RMA functions. Therefore, it does not provide RMA transfer statistics nor does it record the transfers in tracing mode. Recently, the trace collection and visualization toolset VampirTrace/Vampir [8] was extended to provide experimental support for MPI-2 one-sided communication [9].

In our previous work [3], we defined a formal event model as well as a number of characteristic patterns of inefficient behavior that can arise in the context of MPI-2 RMA communication. The detection of these patterns was implemented as an extension of the serial trace analyzer KOJAK [10] and constitutes the foundation for our new, scalable bottleneck detection algorithm.

The *Parallel Performance Wizard* (PPW) [11] is an automatic performance tool specifically designed for *partitioned global address space* (PGAS) languages, which provide the abstraction of shared memory to the user while internally converting all remote accesses to one-sided communication calls. Some PGAS languages, such as UPC, also support explicit one-sided communication. PPW supports the analysis of programs written in such languages by providing so-called generic operation types that are defined on top of an RMA event model.

3 MPI-2 Remote Memory Access

The interface for RMA operations defined by MPI differs from vendor-specific APIs in many respects. This is to ensure that it can be efficiently implemented on a wide variety of computing platforms, even if a platform does not provide any direct hardware support for RMA. The design behind the MPI RMA API is similar to that of weakly coherent memory systems: correct ordering of memory accesses has to be specified by the user with explicit synchronization calls; for efficiency, the implementation can delay communication operations until the synchronization calls occur.

MPI does not allow RMA operations to access arbitrary memory locations. Instead, they can access only designated parts of the memory, which are called *windows*. Such windows must be explicitly initialized with a call to `MPI_Win_create` and released with a call to `MPI_Win_free` by all processes that either want to expose or to access this memory. These calls are collective between all participating partners and may include an internal barrier operation. By *origin* MPI denotes the process that performs an RMA read or write operation, and by *target* the process the memory of which is accessed.

There are three RMA communication calls in MPI: `MPI_Get` to read from and `MPI_Put` and `MPI_Accumulate`¹ to write to the target window. MPI-2 RMA synchronization falls in two categories: *active target* and *passive target* synchronization. In active mode both processes, origin and target, have to participate in the synchronization, whereas in passive mode explicit synchronization occurs only on the origin process. MPI provides three RMA synchronization mechanisms:

Fences: The function `MPI_Win_fence` is used for active target synchronization and is collective over the communicator used when creating the window. RMA operations need to occur between two fence calls.

General Active Target Synchronization (GATS): In this scheme, synchronization occurs between a group of processes that is explicitly supplied as a parameter to the synchronization calls. A so-called *access epoch* is started at an origin process by `MPI_Win_start` and terminated by a call to `MPI_Win_complete`. The start call specifies the group of targets for that epoch. Similarly, an *exposure epoch* is started at a target process by `MPI_Win_post` and completed by `MPI_Win_wait` or `MPI_Win_test`. Again, the post call specifies the group of origin processes for that epoch.

Locks: Finally, shared and exclusive locks are provided for the so-called passive target synchronization through the `MPI_Win_lock` and `MPI_Win_unlock` calls, defining the access epoch for this window at the origin.

It is implementation-defined whether some of the above-mentioned calls are blocking or non-blocking, for example, in contrast to other shared memory programming paradigms, the lock call may not be blocking. In the remainder of this paper, we exclusively focus on active target communication. However, as part of our future work, we plan to address also passive target communication.

¹ A generalized version of `MPI_Put` with the possibility of using a reduction operator.

4 Automatic Detection of RMA Inefficiency Patterns

In this section, we describe how the MPI RMA-related inefficiency patterns defined in [3] can be automatically detected in a scalable way within the framework of the Scalasca performance-analysis toolset. Scalasca is an open-source toolset that can be used to analyze the performance behavior of parallel applications and to identify opportunities for optimization. As a distinctive feature, Scalasca provides the ability to identify wait states in a program that occur, for example, as a result of unevenly distributed workloads, by searching event traces for characteristic patterns. To make the trace analysis scalable, process-local traces are analyzed in parallel without prior merging. The central idea behind Scalasca's parallel trace analyzer is to reenact the application's communication behavior recorded in the trace, analyzing communication operations using operations of the same type. For example, to detect wait-states related to point-to-point message transfers, the events necessary to analyze such a communication are exchanged between the participating processes in point-to-point mode as well. This techniques relies on reasonably synchronized timestamps between the different processes. On platforms without synchronized clocks, a software correction mechanism is applied post mortem [12]. The scalability of the parallel replay mechanism has already been demonstrated for up to 65,536 cores [13].

Here, we apply the same methodology to MPI RMA operations, that is, RMA transfers are used to exchange the data required for the analysis. For this purpose, our analysis creates a small buffer window for every window created by the application itself. The buffers are used by origin and target processes to exchange the timestamps needed for the calculation of waiting times. Earlier, during trace acquisition (i.e., at application runtime), Scalasca's measurement layer keeps track of all windows being created and records the window definitions plus all synchronization and communication operations acting on these windows. When the replay is performed in the analysis step, all those windows can be recreated using the same set of processes based on the recorded window definitions. The timestamps are subsequently transferred using `MPI_Get` and `MPI_Accumulate` operations.

To ensure that the access and exposure epochs are available at the time when the analyzer processes the corresponding part of the event trace, the synchronization pattern used by the original application is reconstructed during the replay. That is, synchronization on the exchange window is triggered by the exit events recorded for the RMA synchronization calls involved. The exit event for `MPI_Win_fence` collectively synchronizes the exchange window, whereas the exit of `MPI_Win_start` opens an access epoch for the recorded group of processes, which is closed whenever the exit event of the corresponding `MPI_Win_complete` call is found. Similarly, an exposure epoch is opened and closed at the exit events of `MPI_Win_post`, `MPI_Win_wait`, and `MPI_Win_test` regions completing an exposure epoch. Please note that the analysis relies on correctly applied synchronization, which is why it may deadlock in cases of erroneous synchronization by the application.

During the replay, specific call backs are triggered for RMA-related events to detect the different inefficiency patterns, as described below. For the sake of simplicity, the individual actions taken are described in the context of the respective pattern. However, our implementation actually combines all these actions using a sophisticated

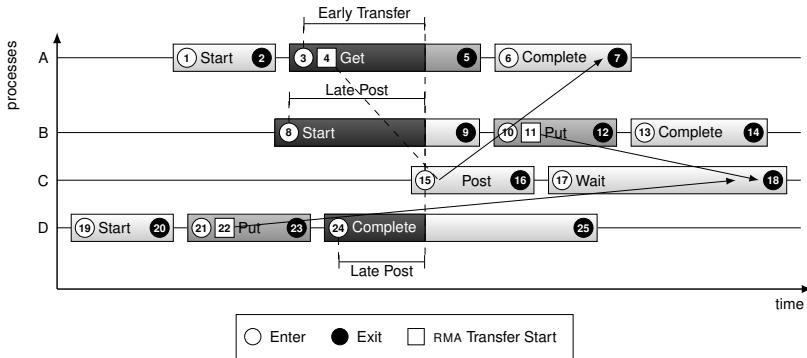


Fig. 1. The Early Transfer and Late Post (in two variants) inefficiency patterns. The waiting time attributed to each pattern is marked in dark gray. Origin and target roles are isolated in different processes—process C is the target for processes A, B, and D.

notification and call-back mechanism not to transfer the same data twice, thereby minimizing the communication costs of the analysis.

Late Post. The Late Post inefficiency pattern refers to waiting time occurring during general active target synchronization (GATS) operations of an access epoch that block until access is granted by the corresponding exposing process as depicted in Figure 1. Depending on the MPI implementation, this may happen either during `MPI_Win_start` (proc. B and C) or `MPI_Win_complete` (proc. D and C). However, the exact blocking semantics are usually not known. Therefore, we use a heuristic to determine which calls are blocking. If and only if the enter event of the call to the latest `MPI_Win_post` on the exposing processes (15) occurs within the time interval of the `MPI_Win_start` call on the accessing process (8,9), we assume that the call to `MPI_Win_start` is blocking, and the waiting time is determined by the time difference between entering the `MPI_Win_post` operation (15) and entering `MPI_Win_start` (8), to which the waiting time is finally ascribed. Likewise, waiting time during the call to `MPI_Win_complete` is determined on the accessing process, where the enter event of the complete call (24) is used to calculate the waiting time. In the case one of these calls is falsely assumed to be blocking, the overall time spent in the call will be very small, resulting in a negligible inaccuracy with respect to the overall severity of this pattern.

To detect the Late Post pattern, the following MPI RMA operations occur during the replay: The exit event of the `MPI_Win_post` call (16) triggers the start of the exposure epoch on the target process after initializing the exchange buffer with the timestamp of the post enter event (15) and default values for all other fields. At the origin processes, the exit events of the call to `MPI_Win_start` (2,9,20) trigger the start of the access epochs for the exchange window and the post enter timestamp of each target process is retrieved using `MPI_Get`. Accordingly, the exit events of the calls to `MPI_Win_complete` (7,14,25) close the access epoch and the post enter timestamps can be accessed to locally determine the latest post. This timestamp can then be compared to the timestamps of the locally available events to determine the Late Post variant and finally calculate the waiting time if applicable. On the target processes, the end of the

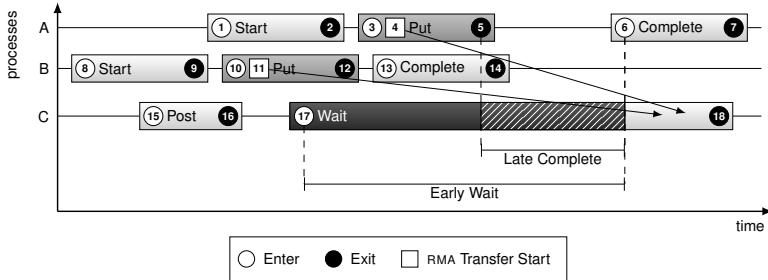


Fig. 2. The Early Wait (dark gray+hatched) and Late Complete (hatched) inefficiency patterns

exposure epoch is ensured by calling `MPI_Win_wait` when reaching the corresponding exit event (18).

Early Transfer. The Early Transfer pattern occurs when an RMA operation is blocking because the relevant exposure epoch has not yet been started (Fig. 1, proc. A and C). It is therefore similar to Late Post, and in fact requires exactly the same data to be transferred (i.e., the post enter timestamps), but the waiting time is attributed to the remote access operation. As before, it can not easily be determined whether the RMA transfer call was actually blocking. However, we assume this to be the case if the corresponding `MPI_Win_post` (15) call was issued on the target side within the time interval of the remote access in question (3,5). Since the post enter timestamps are only accessible after closing the access epoch, a backward traversal of the event data is required, comparing the timestamps recorded for each RMA operation with the post enter timestamp of the corresponding target process. If the RMA operation was non-blocking in reality, the time falsely classified as waiting time would again be very small.

Early Wait. This pattern refers to the situation where the exposing process is waiting for other processes to complete the remote accesses of their access epoch (Fig. 2). As the call to `MPI_Win_wait` cannot return until all access epochs have been finished, the time span between the enter event of the call to `MPI_Win_wait` and the latest enter event of the corresponding calls to `MPI_Win_complete` on the accessing processes is counted as waiting time.

To detect the Early Wait pattern, the timestamps of the enter events of calls to `MPI_Win_complete` (6,13) are transferred to the target processes via `MPI_Accumulate` using the `MPI_MAX` operator just before closing the access epoch, thereby storing the latest complete enter timestamp in the target's exchange buffer. The waiting time can then be determined by subtracting the timestamp of the wait enter event (17) from the latest complete enter timestamp (6) stored in the exchange buffer. As can be seen, the one-sided model naturally lends itself to perform this type of analysis.

Late Complete. To allow for efficient synchronization, access epochs should be as compact as possible. As the target process can close the exposure epoch only after all access epochs have been completed, waiting time in the Early Wait pattern that occurs between the last RMA operation and the completion of the respective access epoch is attributed to the Late Complete pattern (Fig. 2, hatched area), a sub-pattern of Early Wait.

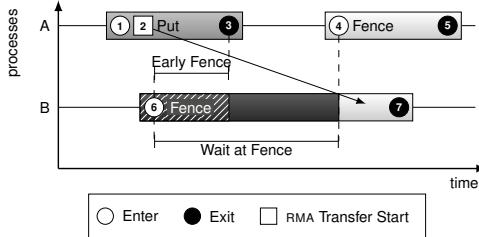


Fig. 3. The Wait-at-Fence (dark gray+hatched) and Early Fence (hatched) inefficiency patterns

During the detection, each origin caches the exit event of the latest RMA operation (5,12) separately for each target. If no RMA operation is present in the access epoch, the exit timestamp of the `MPI_Win_start` call is taken. Then, all the origins of a given target transfer their cached timestamp to the target via `MPI_Accumulate` using the `MPI_MAX` operator just before closing the access epoch while processing the exit events of the calls to `MPI_Win_complete` (7,14). There the maximum value obtained can then be subtracted from the timestamp of the latest complete enter event, which is already available from the Early Wait detection algorithm.

Wait at Fence. This pattern refers to a wait state during the completion of a fence operation as shown in Figure 3. Although `MPI_Win_fence` is a collective call, it may not be synchronizing, depending on given assertions or MPI-internal window status information. However, as potentially all processes of the communicator may access the local window, a confirmation is needed from the remaining processes that their access epoch on this window has ended. Thus, at least a partial synchronization is required. We assume a collective call of `MPI_Win_fence` to be globally synchronizing if the timestamps of all associated enter events occur before any exit event of the same fence call. Unfortunately, this heuristic does not detect and attribute time to the Wait at Fence pattern if only some of the processes synchronize. However, waiting time due to partial synchronization is detected, if the sub-pattern Early Fence is present (see below).

To detect the Wait at Fence pattern, the latest enter and earliest exit timestamps of the fence (4,7) are determined with a single `MPI_Allreduce` call using a user-defined operator. If the above-mentioned overlap criterion is met, the difference between the latest enter event (4) and the local enter event (6) is counted as waiting time.

Early Fence. Waiting time for entering a fence before all remote accesses have finished is attributed to the Early Fence pattern, a sub-pattern of Wait at Fence (Fig. 3, hatched area). It will always be accounted as waiting time, even in situations where only a subset of the processes are synchronizing.

Here, all processes locally determine the latest exit timestamp of their remote accesses (3) for each target and transfer them to the matching target processes via accumulate, again using the `MPI_MAX` operator. These transfers are surrounded by two calls to fence to ensure correct synchronization. In this way the earliest possible completion of the latest RMA operation of all accessing origin processes is determined and used to calculate the waiting time of this pattern as the time difference between leaving the

Table 1. Event statistics and analysis times for the red-black SOR Poisson solver. The last column shows the analysis time in percent of the application runtime.

# cores	# events	trace size [MB]	analysis time [s]	analysis time [%]
128	12,681,376	106.78	2.19	0.75
256	26,130,816	217.13	2.22	0.79
512	53,029,696	437.63	2.78	0.77
1,024	107,595,520	883.25	2.37	0.84
2,048	216,727,168	1,774.63	2.54	0.86
4,096	436,536,592	3,565.63	2.91	1.01
8,192	876,125,440	7,151.25	3.60	1.19

latest RMA operation (3) and the local enter event for the fence (6). Time attributed to the Early Fence pattern is also attributed as time in Wait at Fence, even if due to our heuristic the analyzer was unable to detect the Wait at Fence pattern directly.

5 Results

In this section, we present early results for two different MPI-2 RMA codes. We took our measurements at the Jülich Supercomputing Centre on the IBM Power6 575 cluster ‘‘Jump’’ and an IBM Blue Gene/P system at IBM Rochester. Based on the results collected with up to 8,192 processes on the two-rack Blue Gene/P so far, and the experiences with our replay-based analysis method in general, we believe that the RMA analysis scales well beyond this point.

5.1 SOR Solver

With the first code, SOR, we verify the scalability of our analysis. SOR solves the Poisson equation using a red-black successive over-relaxation method. The two main communication steps are halo-exchange and scalar reduction operations. The former was adapted to use MPI RMA instead of the original non-blocking point-to-point communication. The latter still uses MPI collective communication as before. The global domain is a three-dimensional grid of size $N_{\text{horiz}} \times N_{\text{horiz}} \times N_{\text{vert}}$, which is partitioned along the two horizontal dimensions using a 2D process mesh. The communication pattern of this application is typical for grid-point codes used in earth and environmental science.

A series of experiments was collected and analyzed at different scales on a two-rack IBM Blue Gene/P system at IBM Rochester. The solver was configured to run for approx. 5 minutes using weak scaling and a problem size where no convergence was reached within the maximum number of 1000 iterations. The key numbers are given in Table 1. As can be seen, the total number of events increases linearly with the number of cores, as is the case for the total size of the compressed trace data. The time exclusively needed for the replay analysis (i.e., without loading the traces and writing the results, which together took less than 16 seconds for the 8,192-core run) is only mildly ascending, as expected for weak scaling.

5.2 BT-RMA

To evaluate the usefulness of our analysis for application optimization and to verify that the inefficiency patterns described earlier appear in practice, we incrementally developed a version of the BT benchmark from the NAS Parallel Benchmark Suite 2.4 [14] that uses one-sided instead of non-blocking point-to-point communication named BT-RMA. The BT benchmark solves three sets of uncoupled systems of equations in the three dimensions x , y , and z . The systems are block tridiagonal with 5×5 blocks. The domains are decomposed in each direction, with data exchange in each dimension during the solver part, as well as a so-called face exchange after each iteration. Those exchanges are implemented using non-blocking point-to-point communication in BT. Due to a known problem with fence synchronization on Blue Gene/P systems with V1R3M0 runtime environments, we measured, analyzed and subsequently optimized the BT-RMA code on our IBM Power6 575 cluster “Jump” using the “class D” problem size on 256 cores in ST mode. For measurement, five purely computational subroutines were excluded from instrumentation, lowering the runtime intrusion to about 1% and keeping the trace size manageable.

For simplicity, our initial version of BT-RMA used fence synchronization for both data exchanges. The analysis results (Tab. 2) showed that more than 44% of the overall runtime was spent in active target synchronization calls, that is, `MPI_Win_fence`. Approximately 6% of the total time was found to be waiting time attributable to the Wait at Fence pattern. Further investigation revealed that most of this time was spent in synchronizing the solver exchanges.

We subsequently modified the code to use GATS synchronization in the solver, while still using fences in the face exchange. This version shows a dramatic reduction of the overall execution time to only 57% of the runtime of the fence-only variant. Although significantly faster, active target synchronization still accounts for about 4.2% of the application runtime, with Wait at Fence requiring 1.3% and Early Wait about 0.9%. In addition, this variant uses $2.5 \times$ more time for remote access operations compared to the fence-only version, now spending 1.6% of the total time in the Early Transfer wait state.

Table 2. Performance metrics for different variants of BT-RMA in CPU seconds (first number) and percent of total CPU seconds (second number). All values are inclusive, that is, they include the time for sub-patterns (indicated through indentation).

Metric	fence only		GATS/fence		GATS only		GATS only (opt)	
Total time	109,361.7	100.0	61,888.9	100.0	61,248.7	100.0	60,504.0	100.0
MPI time	51,252.5	46.9	7,156.5	11.6	6,882.5	11.3	6,284.3	10.4
RMA sync.	48,703.8	44.5	2,585.9	4.2	2,177.9	3.6	3,476.0	5.8
Wait at Fence	6,080.0	5.7	805.9	1.3	0.0	0.0	0.0	0.0
Early Wait	0.0	0.0	568.5	0.9	950.1	1.6	1,923.8	3.2
Late Complete	0.0	0.0	1.2	0.0	289.6	0.5	2.0	0.0
Late Post	0.0	0.0	2.9	0.0	4.4	0.0	0.9	0.0
RMA comm.	1,324.9	1.2	3,246.6	5.3	3,507.4	5.7	1,603.2	2.7
Early Transfer	0.0	0.0	980.5	1.6	2,299.6	3.8	848.6	1.4

As a next step, we completely eliminated the calls to `MPI_Win_fence` by adapting the face exchange to also use GATS synchronization with individual windows for each of the six neighbors. Although the Wait at Fence wait state disappeared, the waiting time almost entirely migrated to the Late Complete (mostly in the face exchange) and Early Transfer patterns (predominantly in the solver), thus only providing an additional speedup of approximately one percent.

Based on these analysis results, we finally rearranged the GATS synchronization calls slightly, starting the exposure epochs as early as possible and shortening the access epochs by moving the start/complete calls close to the RMA transfers, decreasing the overall runtime again. BT-RMA is now almost 45% faster than the first fence-based version and marginally faster than the original BT code.

6 Conclusion

MPI-2 remote memory access is a portable interface for one-sided communication on current large-scale HPC systems. To better support developers in using this interface, we have presented a scalable method for identifying wait states in event traces of RMA applications. A particular challenge to overcome was the availability of the communication parameters on only one side of an interaction between two processes, requiring one-sided transfers of analysis data during the parallel replay. We have shown the scalability of our method using one application kernel with up to 8,192 cores and incrementally optimized a second and more complex code guided by results of our analysis.

Future research will evaluate the scalability on even larger process configurations. In addition, we plan to investigate further inefficiency patterns for MPI-2 RMA such as passive target lock competition. We also consider leveraging our method for the scalable automatic analysis of applications written in PGAS languages such as UPC.

Acknowledgments

This work has been supported by the German Ministry for Education and Research (BMBF) under Grant No. 01IS07005C (“ParMA”) and the Helmholtz Association of German Research Centers under Grant No. VH-NG-118. The authors also would like to thank the Rechenzentrum Garching (RZG) of the Max Planck Society and the IPP for the opportunity to run the BT-RMA benchmark on their IBM Power6 system “VIP”, as well as the IBM Rochester Blue Gene Benchmark Center for providing access to their two-rack Blue Gene/P system.

References

1. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard, Version 2.1 (June 2008), <http://www.mpi-forum.org/>
2. Mirin, A.A., Sawyer, W.B.: A scalable implementation of a finite-volume dynamical core in the community atmosphere model. International Journal on High Performance Computing Applications 19(3), 203–212 (2005)

3. Kühnal, A., Hermanns, M.-A., Mohr, B., Wolf, F.: Specification of inefficiency patterns for MPI-2 one-sided communication. In: Nagel, W.E., Walter, W.V., Lehner, W. (eds.) Euro-Par 2006. LNCS, vol. 4128, pp. 47–62. Springer, Heidelberg (2006)
4. Geimer, M., Wolf, F., Wylie, B.J.N., Mohr, B.: Scalable parallel trace-based performance analysis. In: Mohr, B., Träff, J.L., Worringen, J., Dongarra, J. (eds.) PVM/MPI 2006. LNCS, vol. 4192, pp. 303–312. Springer, Heidelberg (2006)
5. Scalasca, <http://www.scalasca.org/>
6. Mohror, K., Karavanic, K.L.: Performance tool support for MPI-2 on Linux. In: Proceedings of the Supercomputing Conference (SC), Pittsburgh, PA (2004)
7. Shende, S.S., Malony, A.D.: The TAU parallel performance system. International Journal of High Performance Computing Applications 20(2), 287–331 (2006)
8. Knüpfer, A., Brunst, H., Doleschal, J., Jurenz, M., Lieber, M., Mickler, H., Müller, M.S., Nagel, W.E.: The Vampir performance analysis tool set. In: Resch, M., Keller, R., Himmeler, V., Krammer, B., Schulz, A. (eds.) Tools for High Performance Computing, pp. 139–155. Springer, Heidelberg (2008)
9. Knüpfer, A.: Personal communication (2009)
10. Wolf, F., Mohr, B.: Automatic performance analysis of hybrid MPI/OpenMP applications. Journal of Systems Architecture 49(10-11), 421–439 (2003)
11. Leko, A., Su, H.H., Bonachea, D., Golden, B., Billingsley, M., George, A.: Parallel Performance Wizard: A performance analysis tool for partitioned global-address-space programming models. In: Proc. of the Supercomputing Conference (SC), vol. 186. ACM, New York (2006)
12. Becker, D., Rabenseifner, R., Wolf, F., Linford, J.: Replay-based synchronization of timestamps in event traces of massively parallel applications. Scalable Computing: Practice and Experience 10(1), 49–60 (2009); Special Issue International Workshop on Simulation and Modelling in Emergent Computational Systems (SMECS)
13. Geimer, M., Wolf, F., Wylie, B.J., Mohr, B.: A scalable tool architecture for diagnosing wait states in massively parallel applications. Parallel Computing (in press) (2009)
14. Bailey, D.H., Barczcz, E., Dagum, L., Simon, H.D.: NAS parallel benchmark results. IEEE Parallel Distrib. Technol. 1(1), 43–51 (1993)

Processing MPI Datatypes Outside MPI

Robert Ross¹, Robert Latham¹, William Gropp²,
Ewing Lusk¹, and Rajeev Thakur¹

¹ Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439, USA
`{rross,robl,lusk,thakur}@mcs.anl.gov`

² Computer Science Department
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
`wgropp@illinois.edu`

Abstract. The MPI datatype functionality provides a powerful tool for describing structured memory and file regions in parallel applications, enabling noncontiguous data to be operated on by MPI communication and I/O routines. However, no facilities are provided by the MPI standard to allow users to efficiently manipulate MPI datatypes in their own codes.

We present *MPITypes*, an open source, portable library that enables the construction of efficient MPI datatype processing routines outside the MPI implementation. MPITypes enables programmers who are not MPI implementors to create efficient datatype processing routines. We show the use of MPITypes in three examples: copying data between user buffers and a “pack” buffer, encoding of data in a portable format, and transpacking. Our experimental evaluation shows that the implementation achieves rates comparable to existing MPI implementations.

1 Introduction

An overwhelming majority of high-performance computing (HPC) codes rely on the Message Passing Interface (MPI) standard. Because MPI is understood so well by application teams and is available on virtually all platforms, additional software has been developed that builds on MPI capabilities and concepts, such as ROMIO [1] and Parallel netCDF [2]. Rich datatype description capabilities are an integral part of the MPI standard and are used in all aspects of MPI, from point-to-point and collective communication to I/O and remote memory access. The MPI datatype facilities make it possible for users to conveniently describe complex, noncontiguous, structured data and operate on this data using a variety of MPI calls in an efficient manner.

Unfortunately, there is a distinct lack of functionality in the MPI standard to allow external libraries to efficiently process MPI datatypes. As a result it is difficult to build libraries that use the MPI descriptive capabilities beyond simply passing these descriptions directly to MPI calls. In this paper we describe *MPITypes*, a portable, open source library for manipulating MPI datatypes in

libraries and applications. Adapted from the datatype processing component of the MPICH2 implementation [3], the MPITypes library includes a set of commonly desired operations, such as copying structured data between a user buffer and a contiguous buffer or generating lists of offsets and lengths described by a datatype (flattening). Additionally, MPITypes provides a framework for developing more complex, case-specific datatype operations for use in libraries and applications. We describe the capabilities of MPITypes through a set of use cases: copying data described with an MPI datatype, data format conversion as performed in Parallel netCDF [2], and an implementation of transpacking [4].

2 Background

HPC libraries take advantage of MPI in various ways. The ROMIO MPI-IO library [1] relies heavily on MPI point-to-point and collective communication to maintain portability across MPI implementations. Parallel netCDF (PnetCDF) [2], a high-level I/O library that provides parallel access semantics to data stored in the netCDF data format [5], likewise relies on MPI file operations to maintain portability across a variety of serial and parallel file systems. In both cases, these libraries rely on MPI datatypes. In the case of PnetCDF, these are used to describe user data structures in memory, while in the case of ROMIO, these types describe regions in memory and in files.

When libraries such as these incorporate the use of MPI datatypes in their interfaces, it is often necessary to process these datatypes in various ways. For example, the ROMIO library must break datatypes apart in order to determine what regions of a file should be accessed and where data resides in memory. In the PnetCDF case, data in memory described by a datatype needs to be encoded in a portable format prior to data movement to file, or decoded prior to placement in the user's buffers. Unfortunately, the MPI standard provides virtually no support for processing of datatypes outside the MPI library. Even the `MPI_Pack` routine, which naïvely appears to provide the functionality needed to move data into a contiguous buffer, cannot be used by external libraries:

The restriction on “atomic” packing and unpacking of packing units allows the implementation to add at the head of packing units additional information, such as a description of the sender architecture (to be used for type conversion, in a heterogeneous environment). [6]

In other words, the data placed in a buffer by `MPI_Pack` cannot reliably be interpreted by an external application.

Libraries building on MPI have addressed this deficiency in different ways. In ROMIO, facilities were implemented to “flatten” types into a list of offset-length pairs, a limiting factor for some access patterns [7]. PnetCDF processes noncontiguous datatypes by first calling `MPI_Pack` to put the data into a contiguous buffer in this undefined format and then using `MPI_Unpack` with a contiguous datatype to put the data in a separate contiguous buffer. This data is then encoded in the appropriate format and written to file. Libraries such as these would

benefit greatly from a library of routines that allow efficient processing of MPI datatypes. Several researchers have addressed the problem of efficiently processing MPI datatypes [8,3]. To date, however, these capabilities have not been made available in a useful form to developers building applications or libraries outside the context of MPI implementations. MPITypes fills this gap.

3 The MPITypes Library

MPITypes is a portable library for efficiently processing MPI datatypes in HPC libraries and applications that rely on MPI. MPITypes is based on the MPICH2 datatype processing functionality [3]. It relies on the *dataloop* representation described in this previous work for efficient processing and operates in the same nonrecursive manner in order to maximize performance. MPITypes provides two capabilities: a set of datatype operators that provide functionality commonly needed by libraries processing MPI datatypes, and a set of functions that form a toolkit for building more specialized type processing.

3.1 Basic MPITypes Functionality

Figure 1 summarizes the MPITypes interface. The first five of these functions provide a basic interface for two of the most common operations on datatypes, packing/unpacking and flattening. The `init` function is responsible for generating the *dataloop* representation of the type used internally for processing, building this from the data available via the MPI datatype envelope and contents calls. These calls provide access to the parameters used to build the MPI datatype. By using these calls to gather this data, MPITypes is portable across all MPI-2 compliant MPI implementations. `init` also allocates a keyval with `MPI_Type_create_keyval` on initial execution and stores data as an attribute on the datatype to be processed. The delete callback associated with the keyval implicitly frees these internal data structures when the last reference to the datatype is freed, eliminating the need for an explicit free operation.

The next three functions perform useful operations on MPI datatypes. `memcpy` is the datatype equivalent of the UNIX function: it copies between a memory region described by an MPI {buffer, count, type} tuple and a contiguous memory region. `flatten` generates lists of displacements and block lengths that specify the regions of memory described by the datatype tuple. `blockct` provides a count of the distinct contiguous regions described by the datatype tuple. All of these functions take start and end offsets, in bytes, that provide the ability to perform partial processing, for example if limited memory is available.

These functions alone would greatly simplify the ROMIO implementation by eliminating the need for an internal flattening functionality and for keeping track of flattened representations. In the case of PnetCDF, the `memcpy` function provides a convenient way to move data between the user's buffer and a contiguous encode/decode buffer, eliminating the need for the current `MPI_Pack/MPI_Unpack` approach. In both cases, however, more task-specific datatype processing would provide additional benefits.

```

/* Basic Functions (MPI_Type) */
int MPIT_Type_init(MPI_Datatype type, int flag);

int MPIT_Type_memcpy(void *typebuf, int count, MPI_Datatype type,
                     void *streambuf, int direction, MPI_Aint start, MPI_Aint *end);

int MPIT_Type_flatten(void *typebuf, int count, MPI_Datatype type,
                      MPI_Aint start, MPI_Aint *end, MPI_Aint *disps, int *blocklens,
                      int *count);

int MPIT_Type_blockct(int count, MPI_Datatype type, MPI_Aint start,
                      MPI_Aint *end, MPI_Aint *blockct);

/* Toolkit Functions (MPI_Type_Segment) */
MPIT_Segment *MPIT_Segment_alloc();

int MPIT_Segment_init(void *buf, int count, MPI_Datatype type,
                      MPIT_Segment *seg, int flag);

int MPIT_Segment_free(MPIT_Segment *seg);

int MPIT_Segment_manipulate(MPIT_Segment *seg, MPI_Aint start, MPI_Aint *end,
                            int (*contigfn) (...), int (*vectorfn) (...), int (*blkidxfn) (...),
                            int (*indexfn) (...), MPI_Aint (*sizefn) (MPI_Datatype el_type),
                            void *pieceparams);

```

Fig. 1. The functions in MPITypes. The MPI_Type functions provide a basic set of operators on MPI datatypes, while the MPI_Segment functions serve as a toolkit for implementing more advanced functionality.

3.2 MPITypes as a Toolkit

The real power of MPITypes is in its use as a toolkit for building more complex, task-specific datatype processing operations. The `memcpy`, `flatten`, and `blockct` functions are all written in terms of the second set of functions in Figure 1. To explain the use of these functions, we first examine the implementation of `MPIT_Type_memcpy` (Figure 2). The `memcpy` implementation consists of three main components: the `MPIT_memcpy_params` structure, the `memcpy` function, and a set of leaf functions.

The `MPIT_memcpy_params` structure holds data specific to the processing we wish to perform (i.e. buffer locations and a direction for copying). For other types of processing different data might be needed, such as arrays to hold offsets and lengths in the case of flattening. The `memcpy` function initializes the task-specific data structure and calls *segment* functions to accomplish the datatype processing. A *segment* is a structure that holds state about the processing of an MPI datatype. This structure is also used to optimize partial processing of datatypes by maintaining the current position in the datatype. The segment functions are provided as part of MPITypes.

The third main component, the leaf functions, are used by `Segment_manipulate`. `Segment_manipulate` drives datatype processing by walking the dataloop representation of the datatype, tracking the current position in the datatype and storing this information in the segment. When it encounters a “leaf” node in the dataloop tree, it executes the leaf function corresponding to the type of leaf node encountered. One of these, `MPIT_Leaf_contig_memcpy`, is shown.

```

/* MPIT_Type_memcpy - Copies data between a region described by an MPI
   (buf, count, type) tuple and a contiguous data buffer.

   Start and end refer to starting and ending byte locations in the type
   map defined by the user datatype. Specifically, end refers to the
   byte offset just past the last to be processed (e.g. to process
   bytes [0..5], start = 0 and end = 6).
*/
typedef struct MPIT_memcpy_params_s {
    int direction;
    char *packbuf;
    char *userbuf;
} MPIT_memcpy_params;

int MPIT_Type_memcpy(void *typebuf, int count, MPI_Datatype type,
                     void *streambuf, int direction, MPI_Aint start, MPI_Aint *end)
{
    int mpi_errno;
    MPIT_Segment *segp;
    MPIT_memcpy_params params;

    segp = MPIT_Segment_alloc();
    MPIT_Segment_init(NULL, count, type, segp, 0);

    params.userbuf      = typebuf;
    params.packbuf      = packbuf;
    params.direction    = direction;

    MPIT_Segment_manipulate(segp, start, end, MPIT_Leaf_contig_memcpy,
                            MPIT_Leaf_vector_memcpy, MPIT_Leaf_blkidx_memcpy,
                            MPIT_Leaf_index_memcpy, NULL, &params);

    MPIT_Segment_free(segp);
    return MPI_SUCCESS;
}

int MPIT_Leaf_contig_memcpy(MPI_Aint *blocks_p, MPI_Type el_type,
                           MPI_Aint dtype_pos, void *unused, void *v_paramp)
{
    MPI_Aint el_size;
    MPI_Aint size;
    MPIT_memcpy_params *paramp = v_paramp;

    MPI_Type_size(el_type, &el_size);
    size = *blocks_p * el_size;

    if (paramp->direction == MPIT_MEMCPY_TO_USERBUF)
        memcpy(paramp->userbuf + dtype_pos, paramp->packbuf, size);
    else
        memcpy(paramp->packbuf, paramp->userbuf + dtype_pos, size);

    paramp->packbuf += size;
    return 0;
}

```

Fig. 2. Excerpts from the implementation of MPIT_Type_memcpy. This implementation allows copying of subregions for pipelining or memory management purposes. Only contiguous leaf function is shown.

The four leaf functions correspond to the four possible dataloop leaf types: contiguous, vector, block indexed, and indexed. Along with the relative position tracked by the manipulate function, the data in the leaf node specifies the mapping of a specific set of user buffer locations to types in the MPI typemap. For example, in the case of a vector leaf node, the leaf would describe a set of strided data regions using a count, block size, and a stride. The task of the leaf function is to perform whatever operation is desired for these regions, in this case copying data between a contiguous user buffer and the region(s) described in the leaf node.

In `MPIT_Leaf_contig_memcpy`, the function copies data for a single contiguous region in both the user buffer and the contiguous buffer. The `MPIT_memcpy_params` tracks the initial user buffer pointer (`userbuf`) and the next contiguous buffer offset (`packbuf`), while the current datatype location, relative to the initial user buffer pointer, is provided by the manipulate function (`dtype_pos`).

`Segment_manipulate` understands how to use a contiguous leaf function to process any other type of leaf function, so a simple implementation of `memcpy` would only include our contiguous leaf function. However, in cases where the overhead of processing the type is expected to dominate, such as when simply copying data, implementing support for the other three types of leaf nodes provides a substantial boost in performance. The MPITypes implementation of `memcpy` includes all four leaf-processing functions.

4 Case Studies in Datatype Processing

MPITypes provides a great deal of convenience for users who wish to work with MPI datatypes, but if it does not perform efficiently, then it has little practical value. In this section we evaluate the MPITypes framework. First, we compare the performance of MPITypes when copying data to that of `MPI_Pack/MPI_Unpack` and manual copying of data. Next, we examine the cost of an MPITypes-based PnetCDF data coding implementation as compared to simply copying data and to the existing PnetCDF approach, for an example data type. We also examine the performance of a MPITypes-based transpacking implementation as compared to the naïve pack/unpack approach.

All tests were performed on the “breadboard” system at Argonne National Laboratory. The node used for testing is an 8-core, 2.66 GHz Intel Xeon system with 16 Gbytes of main memory, Linux 2.6.27, and gcc 4.2.4. Tests with MPICH2 use version 1.0.8p1, compiled with “`--enable-fast=03`”. Tests with Open MPI use version 1.3.1, compiled with “`CFLAGS=-O3 --disable-heterogeneous --enable-shared=no --enable-static --with-mpi-param-check=no`”.

4.1 Moving Data between User Buffers and Contiguous Buffers

We have modified the synthetic tests used in [3] to compare the MPITypes implementation of `MPIT_Type_memcpy` with the `MPI_Pack` and `MPI_Unpack` routines and hand-coded routines that manually pack and unpack data using tight loops

Table 1. Comparing MPI_Pack/MPI_Unpack, MPIT_Type_memcpy, and manual copy rates

Test	Element Type	MPICH2 (MB/sec)	Open MPI (MB/sec)	MPITypes (MB/sec)	Manual (MB/sec)	Size (MB)	Extent (MB)
Contig	float	4578.49	4561.09	4579.84	2977.98	4.00	4.00
Contig	double	4152.07	4157.69	4149.13	2650.81	8.00	8.00
Vector	float	1788.85	1088.01	1789.37	1791.59	4.00	8.00
Vector	double	1776.81	1680.23	1777.04	1777.60	8.00	16.00
Indexed	float	803.49	632.32	829.75	1514.94	2.00	4.00
Indexed	double	1120.59	967.69	1123.97	1575.41	4.00	8.00
XY Face	float	18014.16	15700.85	17962.98	9630.47	0.25	0.25
XY Face	double	17564.43	18143.63	17520.11	16423.59	0.50	0.50
XZ Face	float	3205.28	3271.29	3190.69	3161.28	0.25	63.75
XZ Face	double	4004.26	4346.81	3975.23	3942.41	0.50	127.50
YZ Face	float	145.32	93.08	145.32	143.68	0.25	63.99
YZ Face	double	153.89	154.19	153.88	153.96	0.50	127.99

that exploit knowledge of data layout. These tests cover a wide variety of possible user types, from simple strided patterns to complex, nested strides and types with no apparent regularity.

Each test begins by allocating memory, initializing the data region, and creating a MPI type describing the region. Next, `MPIT_Type_init` is called to generate the dataloop representation used by MPITypes and store it as an attribute on the type. A set of iterations is performed using `MPI_Pack` and `MPI_Unpack` in order to get a rough estimate of the time of runs. Using this data, we then calculate a number of iterations to time and execute those iterations. The process is repeated for the MPITypes approach, replacing `MPI_Pack` and `MPI_Unpack` with two calls to `MPIT_Type_memcpy`. Finally, manual packing and unpacking routines (hand-coded loops) are timed. Table 1 summarizes the results of this testing.

The *Contig* test operates on contiguous data using `MPI_FLOAT` and `MPI_DOUBLE` datatypes. A contiguous datatype of 1,048,576 elements is created, and a count of 1 is used. The *Vector* tests operate on a vector of 1,048,576 basic types with a stride of 2 types (i.e. accessing every other type). A count of 1 is used when calling pack/unpack routines. MPITypes performance tracks MPICH2 in these tests. Open MPI performs slightly more slowly for the vector type, indicating room for improvement in this case.

The *Indexed* set of tests use an indexed type with a fixed, regular pattern with multiple strides. Every block in the indexed type consists of a single element (of type `MPI_FLOAT` or `MPI_DOUBLE`, depending on the particular test run). There are 1,048,576 such blocks in the type. In these tests we see higher performance for the manual approach, because our manual data movement implementation takes advantage of knowledge of these two strides, whereas this information is not recognized by the MPI implementations or MPITypes. Performance is similar for the two MPI implementations and MPITypes in these tests.

The *3D Face* tests pull entire faces off a 3D cube of elements, described in Appendix E of [9]). Element types are varied between `MPI_FLOAT` and `MPI_DOUBLE`

types. The 3D cube is 256 elements on a side. Performance is virtually identical for the MPI implementations, MPITypes, and the manual copying in these tests, with the exception of the XY Face results for hand-coded loop. This is due to an optimized UNIX `memcpy()` being used in the MPI and MPITypes cases.

4.2 Parallel netCDF Data Encoding and Decoding

As described earlier, PnetCDF uses an inefficient approach when encoding or decoding data for a noncontiguous user buffer, in order to avoid the need to process MPI datatypes. A more interesting use of MPITypes would remove the need for this intermediate buffer in the PnetCDF encode/decode process. There are two components to this process: data translation and byte reordering. Data translation occurs when the data in the user's buffer is of a different type from the variable in which it is to be stored. This can happen, for example, when writing data for visualization purposes in simulations codes: data in memory in double-precision floating points is stored in single-precision format to reduce I/O demands. If the two types are the same, format translation is not necessary. The netCDF file format calls for big-endian data. If the system is a little-endian machine (e.g., Intel), then byte translation must be performed. Since our test system is an Intel system, we will perform this byte swapping.

The MPITypes `memcpy` implementation provides a convenient starting point for an implementation of this PnetCDF functionality. Through experimentation, we found that for cases where data translation is not necessary, the most efficient approach was to use the MPITypes `memcpy` function unchanged to copy data into a contiguous buffer, then perform a single pass over the buffer to swap bytes.

In the case where data translation is necessary, we constructed a new function, `MPIT_Type.netcdf.translate`, to perform the data translation as part of the copy process (and to perform byte swapping when required). The type of data being stored in netCDF is passed along in the structure of parameters to a new set of leaf functions. In the case of encoding, the leaf functions first perform data translation into the new buffer, then byte swap. This process is reversed for decoding.

The availability of the “element type” in the leaf function provides critical information for the translation process. The combination of the type of data in the user's buffer, available via the element type parameter, and the desired format of the data in the file, stored in the parameters passed to the leaf functions, defines the translation to be applied. Notice that because the position in the contiguous buffer is tracked by the leaf function, data encoding that requires a change in the size of the data is possible; we simply increment the location by the size of the data in the encoded format, rather than by the size of the data in the native format.

We use the basic data structure from the Flash astrophysics application as our test case. The Flash code is an adaptive mesh refinement application that solves fully compressible, reactive hydrodynamic equations, developed mainly for the study of nuclear flashes on neutron stars and white dwarfs [10]. The data consists of some number of 3D blocks of data. Each block consists of a $8 \times 8 \times 8$

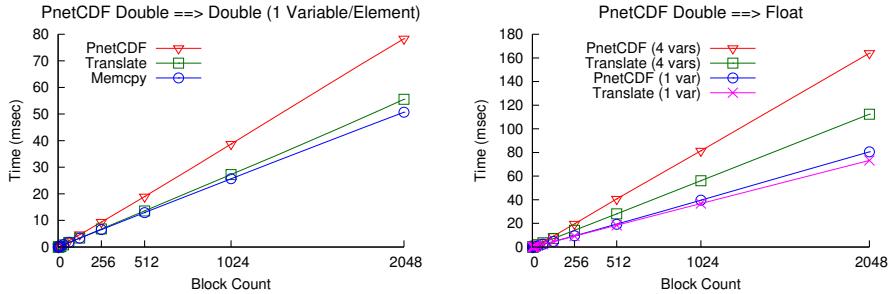


Fig. 3. Performance of PnetCDF data encode implementation for quadruply nested vector type, for the case where data only needs to be byte swapped (left) and for a double-to-float translation and byte swap (right). A block is an $8 \times 8 \times 8$ array of elements.

array of elements surrounded by a guard cell region four elements deep on each side. Each element consists of 24 variables, each an MPI_DOUBLE. This type is representative of the most complex types we would expect to see from most applications.

In our tests we construct a quadruply nested vector type that references all local elements of one or more variables, skipping the guard cells. Figure 3 shows the results of our experiments. On the left, performance for encoding a single variable out of n blocks of the Flash type is examined. Our new approach, using `memcpy` followed by byte swapping, results in a 29% reduction in time as compared to the original PnetCDF approach. A standard MPIType `memcpy` is shown as a reference. On the right, we examine performance for the case where data in the user buffer in double-precision floating point is converted to single precision for writing into the file. We show results for extraction of both one variable and four contiguous variables. For the case of a single variable, we see only approximately a 9% improvement over the existing approach despite a significant tuning effort. Performance for four adjacent variables is 31% faster using the new approach. This indicates that for types with very small contiguous regions, we should not expect substantial gains in directly manipulating the data as compared to moving it into a contiguous buffer. On the other hand, in a situation with memory constraints, the ability to operate in this manner with some performance improvement, while simultaneously reducing the total memory requirement, is a significant advantage.

4.3 Transpacking

Transforming data between datatype representations is a complex task that can be an important part of certain algorithms. For example, the data sieving implementation in ROMIO requires mapping data between the user's datatype and a portion of the file view defined on a file [11] in order to combine noncontiguous I/O operations into fewer, larger I/O accesses.

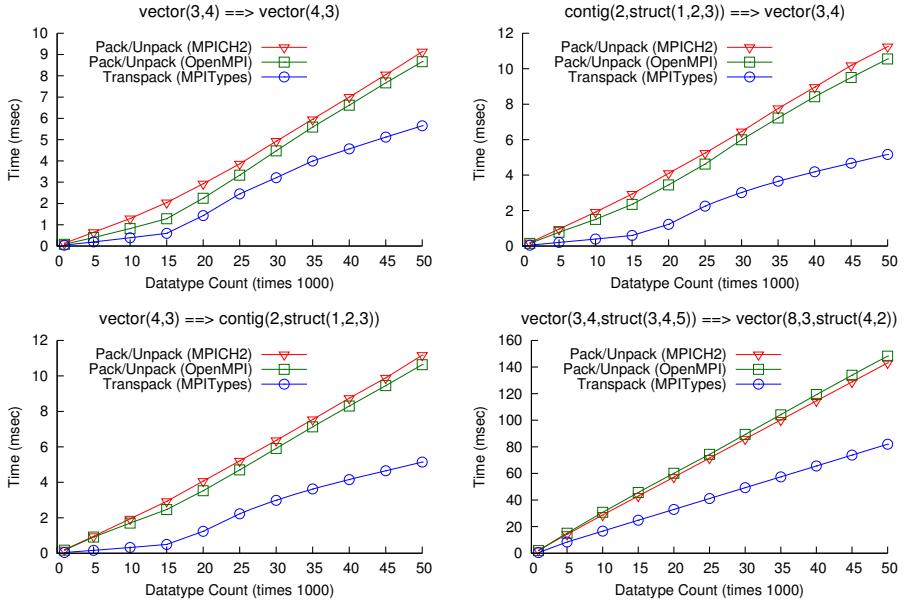


Fig. 4. Performance of template-based transpack implementation, as compared to MPICH2 and Open MPI `MPI_Pack/MPI_Unpack`. Time to create the `MPITypes` representation and to generate the template is included in the Transpack times.

Data sieving can be thought of as an example of the *typed copy problem* [4].¹ The typed copy problem consists of moving data from one datatype representation to another, and a naïve solution to this problem is to simply pack and unpack using the two datatypes. The approach of moving the data directly from one representation to the other, without using an intermediate buffer, is known as *transpacking*. An elegant solution to transpacking has been described that relies on the generation of a new datatype representation that includes two offsets for a given element rather than one [4].

Our approach recognizes that based on the prior work in this area [4], transpacking is most often effective when applied on large counts of relatively small types and that in many cases the combination of types results in a pattern without any expressible regularity (i.e., the resulting combination type is an indexed type).

Working under these assumptions, and assuming identical type sizes as in the original work, we constructed an implementation that generates a template necessary to copy a single instance of each type (i.e. a flattened *input-output type*). This construction also uses nested calls to `Segment_manipulate`, but only for one instance of each type. For example, the template generated from a vector with a count of four would list the four regions described by the type. The

¹ Actually data sieving requires the ability to perform *partial* typed copy operations in order to limit buffer requirements.

template generated from this process is then used to copy between multiple instances of the types. The overhead of processing in this case is quite low.

We implemented a subset of the tests used in [4] to evaluate our prototype transpack implementation. We did not test cases where one or both buffers were contiguous, as these cases do not require the use of our new functionality. Figure 4 shows the results of these tests. Our template generation approach is able to exceed a factor of two performance gain for most cases. The cost to generate the template for copying is small enough that the approach is viable for even small numbers of types: the break-even point occurs before 100 type instances for all tests.

5 Conclusions and Future Work

In this paper we have presented MPITypes, a library for processing MPI datatypes and building case-specific processing functions in software using the MPI programming model. We have shown an example of the use of MPITypes for data copying, and we have described its application in PnetCDF as a tool for more efficiently performing data encoding and decoding in this library. We also discussed an advanced application, transpacking, and the nested use of MPITypes to implement this capability.

MPITypes is a portable package that relies only on standard interfaces available as part of the MPI-2 extensions, namely, datatype attributes and the envelope and contents calls. So far it has been tested and shown to work correctly on top of both MPICH2 and Open MPI, two popular implementations. We are releasing MPITypes under an open source license in the hope that it will encourage greater use of MPI datatypes as a language for describing noncontiguous data regions in parallel applications.² Perhaps the capabilities provided in MPITypes could be considered for incorporation into the MPI 3.0 standard, ensuring these facilities are available for each implementation.

We intend to further explore the use of MPITypes in I/O libraries, including the use of MPITypes functionality as the basis for new implementations of data sieving and two-phase optimizations in the ROMIO MPI-IO library. Augmenting MPITypes to include functionality for serializing and deserializing types, so that they may be efficiently passed between processes, seems a useful enhancement along these lines as well.

Acknowledgments

We would like to thank our reviewers for their helpful suggestions. This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy, under Contract DE-AC02-06CH11357, and in part by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy award DE-FG02-08ER25835.

² See <http://www.mcs.anl.gov/research/projects/mpitypes/>

References

1. Thakur, R., Gropp, W., Lusk, E.: On implementing MPI-IO portably and with high performance. In: Proceedings of the 6th Workshop on I/O in Parallel and Distributed Systems, pp. 23–32. ACM Press, New York (1999)
2. Li, J., Liao, W.-k., Choudhary, A., Ross, R., Thakur, R., Gropp, W., Latham, R., Siegel, A., Gallagher, B., Zingale, M.: Parallel netCDF: A high-performance scientific I/O interface. In: Proceedings of SC2003: High Performance Networking and Computing, Phoenix, AZ. IEEE Computer Society Press, Los Alamitos (2003)
3. Ross, R.J., Miller, N., Gropp, W.D.: Implementing fast and reusable datatype processing. In: Dongarra, J., Laforenza, D., Orlando, S. (eds.) EuroPVM/MPI 2003. LNCS, vol. 2840, pp. 404–413. Springer, Heidelberg (2003)
4. Mir, F., Träff, J.: Constructing MPI input-output datatypes for efficient transpacking. In: Lastovetsky, A., Kechadi, T., Dongarra, J. (eds.) EuroPVM/MPI 2008. LNCS, vol. 5205, pp. 141–150. Springer, Heidelberg (2008)
5. Rew, R., Davis, G.: The unidata netCDF: Software for scientific data access. In: Sixth International Conference on Interactive Information and Processing Systems for Meteorology, Oceanography and Hydrology, February 1990, pp. 33–40 (1990)
6. Message Passing Interface Forum: MPI-2: Extensions to the message-passing interface (July 1997), <http://www.mpi-forum.org/docs/docs.html>
7. Worringen, J., Träff, J.L., Ritzdorf, H.: Improving generic non-contiguous file access for MPI-IO. In: Dongarra, J., Laforenza, D., Orlando, S. (eds.) EuroPVM/MPI 2003. LNCS, vol. 2840, pp. 309–318. Springer, Heidelberg (2003)
8. Träff, J., Hempel, R., Ritzdoff, H., Zimmermann, F.: Flattening on the fly: Efficient handling of MPI derived datatypes. In: Margalef, T., Dongarra, J., Luque, E. (eds.) PVM/MPI 1999. LNCS, vol. 1697, pp. 109–116. Springer, Heidelberg (1999)
9. Gropp, W., Lusk, E., Skjellum, A.: Using MPI: Portable Parallel Programming with the Message-Passing Interface. MIT Press, Cambridge (1994)
10. Fryxell, B., Olson, K., Ricker, P., Timmes, F.X., Zingale, M., Lamb, D.Q., MacNeice, P., Rosner, R., Tufo, H.: FLASH: An adaptive mesh hydrodynamics code for modelling astrophysical thermonuclear flashes. Astrophysical Journal Supplement 131, 273–334 (2000)
11. Thakur, R., Gropp, W., Lusk, E.: Data sieving and collective I/O in ROMIO. In: Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation, pp. 182–189. IEEE Computer Society Press, Los Alamitos (1999)

Fine-Grained Data Distribution Operations for Particle Codes

Michael Hofmann* and Gudula Rünger

Department of Computer Science
Chemnitz University of Technology, Germany
`{mhofma,ruenger}@cs.tu-chemnitz.de`

Abstract. This paper proposes a new fine-grained data distribution operation `MPI_Alltoall_specific` that allows an element-wise distribution of data elements to specific target processes. This operation can be used to implement irregular data distribution operations that are required, for example, in particle codes. We present different implementation variants for `MPI_Alltoall_specific` which are based on collective MPI operations, on point-to-point communication operations, or on parallel sorting. The properties of the implementation variants are discussed and performance results with different data sets are presented. For the performance results two high scaling hardware platforms, including a Blue Gene/P system, are used.

Keywords: MPI, all-to-all communication, irregular communication, personalized communication, particle codes.

1 Introduction

Implementing efficient data distribution operations is a crucial task for achieving performance in parallel scientific applications, especially on high scaling supercomputer systems. Often MPI communication operations are used to implement data distribution operations. MPI provides point-to-point operations and collective operations that implement classical communication patterns like *one-to-all*, *all-to-one* and *all-to-all*. Vector variants of collective operations like `MPI_Scatterv`, `MPI_Gatherv`, and `MPI_Alltoallv` provide a way to use the operations in applications with irregular communication patterns, too. All these communication operations require that the data elements to be distributed are organized in contiguous blocks. This can require additional efforts for preparing the data for the communication operations, especially when the data is organized in application specific data structures.

In this paper we introduce a new fine-grained data distribution operation `MPI_Alltoall_specific` that allows an element-wise distribution of data elements to specific target processes instead of the block-wise distribution currently provided by MPI. The `MPI_Alltoall_specific` operation requires that the information about the target process of an element is located inside the element itself.

* Supported by Deutsche Forschungsgemeinschaft (DFG).

This kind of data structures can be found, for example, in particles codes where the data is distributed according to domain decomposition schemes and the information about the data partitioning is associated with the single particles. We introduce several alternative implementations for the `MPI_Alltoall_specific` operation and discuss their properties. To investigate the performance of the different implementations, we present performance results on an SMP cluster using up to 256 processes and on a Blue Gene/P system using up to 4096 processes.

The rest of this paper is organized as follows. Section 2 presents related work. Section 3 proposes the new operation `MPI_Alltoall_specific` for the fine-grained data distribution. Section 4 presents different implementation variants of this operation. Section 5 shows performance results and Section 6 concludes the paper.

2 Related Work

Spatial decomposition techniques and load balancing techniques used in particles codes for molecular dynamics or astrophysical simulations often require many-to-many personalized communication (e.g., [1]). The resulting irregular communication pattern can require specialized algorithms for achieving good performance on different hardware platforms [2,3]. Also, there have been efforts to perform these data distribution operations with limited additional memory [4,5]. Sparse collective operations provide an approach to communicate only with a subset of neighboring processes, thus making the interfaces of the collective operations independent from the total number of nodes [6].

All these communication operations as well as their underlying algorithms require that the data to be distributed consists of contiguous blocks of elements. Non-contiguous blocks of elements can be distributed in MPI by creating specific MPI data types (see [7], examples 4.17 and 4.18). Because of the additional costs and resources required for the creation of these data types, the feasibility of this approach strongly depends on the number of blocks and the reuse of the data types. In [8], Bader et al. present parallel algorithms for h -relation personal communication. This communication operation sends elements to specific target locations only assuming that no process receives more than h elements. The proposed architecture-independent algorithm can be adapted for an implementation of the `MPI_Alltoall_specific` operation.

3 Fine-Grained Data Distribution Operations

Message passing operations in MPI can be used to send blocks of data to target processes. The blocks consist of consecutive data elements, specified by the number of elements, the MPI data type of a single element and the starting address. Collective operations can be used to send multiple blocks to distinct processes in one pass. This coarse-grained nature (*one block for one process*) is inherent to all operations in MPI. To overcome this limitation, we propose a new fine-grained data distribution operation by specifying target processes on a per element basis.

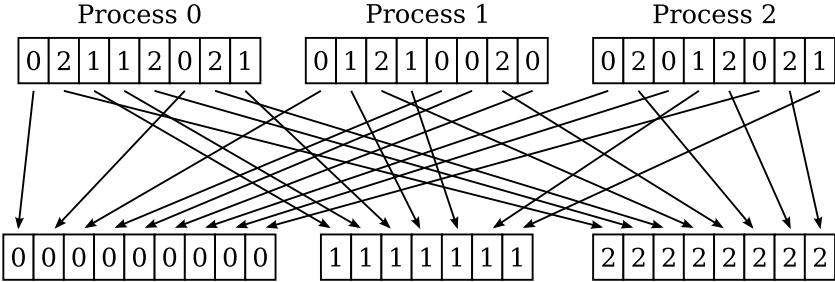


Fig. 1. Fine-grained data re-distribution example with three processes

Every process j with $0 \leq j < p$ participating in the fine-grained distribution operation contributes a list of n_j elements $e_{j0}, \dots, e_{jn_j-1}$ and receives \hat{n}_j elements $\hat{e}_{j0}, \dots, \hat{e}_{j\hat{n}_j-1}$. Let $tproc(e)$ denote the target process of the distribution for an element e . After performing the distribution operation, every process j has received exactly those elements \hat{e}_{ji} with $0 \leq i < \hat{n}_j$ and $tproc(\hat{e}_{ji}) = j$. The function $tproc$ specifies only the target process, but not the resulting order of the elements on a target process. We adopt the definition of stability from sorting algorithms (see [9]) to define a special ordering of the elements after the distribution. Let $sproc(e)$ and $spos(e)$ denote the source process of an element e and its original position on the source process, respectively. We call the fine-grained distribution *stable* if for any two elements \hat{e}_{ji} and $\hat{e}_{j'i'}$ with $j = j'$ and $i < i'$ either $sproc(\hat{e}_{ji}) < sproc(\hat{e}_{j'i'})$ or $sproc(\hat{e}_{ji}) = sproc(\hat{e}_{j'i'}) \wedge spos(\hat{e}_{ji}) < spos(\hat{e}_{j'i'})$. We call an implementation of the fine-grained data distribution operation stable if it always performs a stable distribution. Otherwise, the implementation is called *unstable*.

Figure 1 shows an example with three processes for the situation before and after such a fine-grained re-distribution. Initially, the elements are arbitrarily placed on the processes. Every element holds a process rank that specifies the target process for the distribution. After the distribution all elements with target process rank 0 are located on process 0, etc. The example in Figure 1 shows a stable distribution since the original order of the elements on a single target process is maintained.

The fine-grained data distribution operation is customized for the needs of irregular applications like particle codes. In these codes, domain decomposition or partition techniques (e.g., based on space-filling curves) are used to assign single data elements to individual target processes. At least two different kinds of data distribution can be distinguished in common scientific applications. (1) Distribution of initial data (before any locality preserving scheme is applied) and (2) distribution of (slightly) altered data, e.g. after certain time steps in molecular dynamics or astrophysical simulations. In the first case, the elements are uniformly distributed to all processes and the amount of communication between all processes is almost equal. In the second case, each process exchanges elements only with a small number of neighboring processes, which leads to an unequal amount of communication load between the processes.

We propose a new collective operation `MPI_Alltoall_specific` to implement the fine-grained data distribution operation. It has the following prototype:

```
int MPI_Alltoall_specific(
    void *sendbuf, int sendcount, MPI_Datatype sendtype,
    void *recvbuf, int recvcount, MPI_Datatype recvtype,
    int tproc_pos, int *received, MPI_Comm comm);
```

The operation uses a send buffer (`sendbuf`, `sendcount`, `sendtype`) to specify the local contribution of elements and a receive buffer (`recvbuf`, `recvcount`, `recvtype`) to store incoming elements. Because `recvcount` specifies only the maximum size of the receive buffer, the actual number of elements received is returned in the output value `received`. The individual target processes of the elements (previously given by $tproc(e)$ for an element e) have to be part of the elements. `tproc_pos` specifies the displacement in bytes (relative to the base address of an element) where the information about the target process is located inside an element. This current approach requires to use a single integer value of fixed size at a fixed position and with a fixed range of values. A more flexible approach is to use a user-defined callback function that maps a single element to a target process. However, using a callback function can result in a loss of performance in comparison to having direct access to the target process values. Using callback functions also imposes a read-only access to the target process values that makes it impossible to apply modifications to these values (e.g. to gain stability, see Section 4). To support both, performance and flexibility, an alternative would be to support a set of pre-defined (parametrized) mapping functions as well as user-defined mapping functions (similar to pre-defined and user-defined reduction operations in MPI).

In comparison to the standard all-to-all communication operations, the interface of the `MPI_Alltoall_specific` operation is independent from the number of processes p . It depends only on the local number of elements n_j to be distributed by process j . Specifying individual target processes per element usually requires more memory than specifying only counts and displacements for each process. But, in many situations the additional storage location for the target process is already part of the application data, whereas the memory for send and receive counts and displacements (e.g., required by `MPI_Alltoallv`) needs to be allocated additionally.

Several MPI operations can work in-place by providing the `MPI_IN_PLACE` argument instead of the send or receive buffer. For in-place operations, the output buffer is identical to the input buffer and the input data is replaced by the output data of the communication operation. We call operations with separate input and output buffer *out-of-place* operations.

We give three examples for using the `MPI_Alltoall_specific` operation with different sample data sets. The examples are later used for the performance measurements in Section 5.

Example A. As a first example, we consider p processes and an array of integers with values in the range of $0 \dots p - 1$ as send buffer for each process (Figure 1

can be seen as an example with $p = 3$). The integers are the elements to be distributed and their values are used as target process values by specifying `tproc_pos = 0`. After a call to `MPI_Alltoall_specific` all integers with value j are located on process j . We use random values (in the range of $0 \dots p - 1$) for the integer values.

Example B. Another example can be found in particle codes, where the elements contain more complex information describing the properties of the particles. We use arrays of the following example data structure for the particles:

```
typedef struct {
    double pos[3]; /* position */
    double q;       /* charge or mass */
    double vel[3]; /* velocity */
    int tproc;     /* target process */
} particle_data_t;
```

The `tproc` component specifies the target process of the particle. The offset of the `tproc` component inside the data structure has to be used for the `tproc_pos` value and can be determined, for example, with `MPI_Get_address`.

We determine the target processes of the particles by adopting a data partitioning scheme based on space filling curves [10]. To get a distribution similar to initial data before any locality preserving scheme is applied, the particles are randomly placed inside a unit cube. This cube is equally divided into 8^6 smaller sub-cubes. The sub-cubes are numbered according to a Z-order space filling curve and then equally distributed to all processes. Each particle is located in one of the sub-cubes and gets its target process from the distribution of the sub-cubes to the processes. Because of the initial random placement of the particles, almost all particles need to be distributed to different processes.

Example C. The last example uses the output data from Example B and additionally moves the particles randomly. Each particle is allowed to do a step in each of the three dimensions. The step size is limited to the half width of a sub-cube. After the movement, the particles get new target processes corresponding to the sub-cubes they have moved to. Due to the limited movement, only a small number of particles move to neighboring sub-cubes that are assigned to other processes. This leads to an unequal amount of communication load between the different processes.

4 Implementation Variants for MPI_Alltoall_specific

In the following, we present four implementation variants called `alloallw`, `alltoally`, `sendrecv` and `sort` for the `MPI_Alltoall_specific` operation and discuss their properties.

alltoallw. The alltoallw implementation uses the MPI operation `MPI_Alltoallw`. `MPI_Alltoallw` is the most flexible all-to-all communication operation in MPI and allows the specification of lists with separate send and receive data types for every process. To implement `MPI_Alltoall_specific` using `MPI_Alltoallw`, each process creates p different send types with `MPI_Type_create_indexed_block` that cover exactly the fine-grained structure of the data. On process j , the k -th send type with $0 \leq k < p$ uses all indices i with $0 \leq i < n_j$ and $tproc(e_{ji}) = k$. The receive types are contiguous types, whose counts and displacements are determined in advance (this requires an additional call to `MPI_Alltoall`). Using these MPI data types, the `MPI_Alltoall_specific` operation can be implemented with a call to `MPI_Alltoallw`. An advantage of this implementation is that the original data of the input buffer remains unchanged. Potential disadvantages are high costs for creating the appropriate MPI data types. These data types can not be reused and need to be created for each call to `MPI_Alltoall_specific`. The alltoallw implementation is stable if the indexed data types and the receive displacements are created in an appropriate (straight forward) way. The alltoallw implementation can not work in-place, because the `MPI_IN_PLACE` option is currently not supported by `MPI_Alltoallw`.

alltoally. This implementation first sorts the local elements of each process according to the target process ranks. The standard operation `MPI_Alltoally` can then be used to distribute the sorted elements as contiguous blocks. We implement the local sort, by copying the elements from the send buffer to the receive buffer and then sorting the elements from the receive buffer to the send buffer using a (stable) radix sort algorithm. Unless an additional buffer (equal to the size of the input data) is used to store the sorted elements, the local sort changes the original data of the input buffer and requires that the receive buffer is as large as the send buffer (this can be avoided using an in-place local sort). The alltoally implementation is stable if the local sort is stable. The alltoally implementation can not work in-place, because the `MPI_IN_PLACE` option is currently not supported by `MPI_Alltoally`.

sendrecv. The sendrecv implementation iterates over all elements and sends them to the target processes using point-to-point communication operations. Because transferring single elements is very inefficient, we use auxiliary buffers (one per target process) to aggregate larger messages. We use buffers equal to the size of 128 elements for each target process. `MPI_Isend` operations are used to send the messages, while at the same time every process receives messages with `MPI_Irecv`. The data of the input buffer remains unchanged. Additionally, this implementation variant may be advantageous if only a small number of elements (that fit into the auxiliary buffers) need to be exchanged. The sendrecv implementation performs a stable distribution if the iteration over the elements starts at the first element and if the exact receive displacements are determined in advance, so that they can be used to place received elements. An unstable variant of the sendrecv implementation can neglect the distinct receive displacements for every source process.

sort. This implementation uses a parallel sorting algorithm to sort all elements according to the target process ranks. A final balancing step after the parallel sorting is used to make sure that every process receives its part of the sorted list of elements. We use the merge-based parallel sorting algorithm introduced in [11] for this implementation variant of the `MPI_Alltoall_specific` operation. The sort implementation performs an unstable distribution, because our underlying merge-based parallel sorting algorithm is unstable, too. The sort implementation can be made stable, by replacing the (non-distinct) target process values with new distinct keys computed from the source process rank and the original position of the elements. The original target process values can be restored afterwards. The parallel sorting algorithm works in-place, but is also able to use additional memory to improve its performance. We use the send buffer as additional memory for the out-of-place variant of the sort implementation. Consequently, the original data in the send buffer is changed. For the in-place variant, we use additional memory of fixed size (1024 elements).

5 Performance Results

We use the three examples presented in Section 3 to demonstrate the performance of the different implementations of the `MPI_Alltoall_specific` operation. The performance experiments are performed on the supercomputer systems JUMP and JUGENE. The IBM Power 575 system JUMP consists of 14 SMP nodes (32 Power6 4.7 GHz processors and 128 GB main memory per node) with InfiniBand interconnects for MPI communication. A vendor specific MPI implementation is used (PE MPI, part of *IBM Parallel Environment*). JUGENE is an IBM Blue Gene/P system with a total number of 16384 compute nodes (4-way SMP processor and 2 GB main memory per node) and several specific networks (3D torus network, collective network, barrier network) for MPI communication. An MPICH2 based MPI implementation optimized for the Blue Gene architecture is used [3].

Figure 2 shows the communication times for the different implementation variants of `MPI_Alltoall_specific` depending on the number of processes. Each process contributes 100.000 elements. The input data corresponds to Example A. The results for the unstable sendrecv implementation are not shown, because the differences to the stable variant were too small.

The results on both hardware platforms show that the alloally implementation is always the best while all other implementations are significantly slower. Comparing only the two implementation variants alloallw and sendrecv that do not change the input data, sendrecv achieves better results. The performance of alloallw suffers from the expensive creation of the indexed data types. On JUGENE with more than 1024 processes, the differences between these two variants become rather small. The sort implementations show differences in performance between the stable and unstable variants. This is caused by the additional work required for the key modification of the stable variant. Working in-place reduces the performance of the sort implementation on both hardware platforms, too.

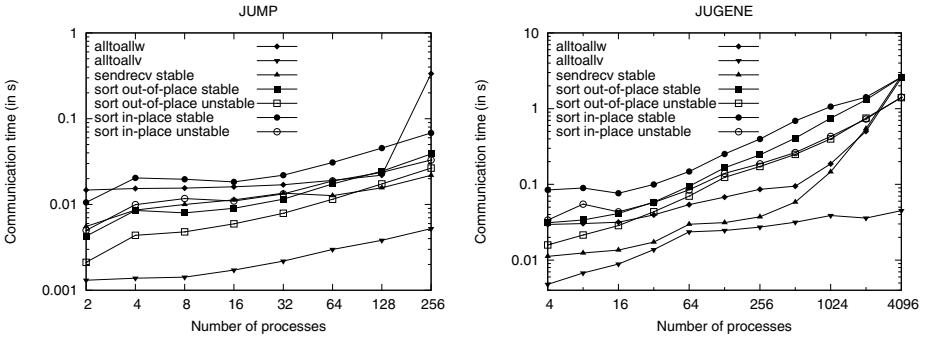


Fig. 2. Communication times for the implementation variants of `MPI_Alltoall_specific` with data corresponding to Example A on the JUMP (left) and the JUGENE (right) system

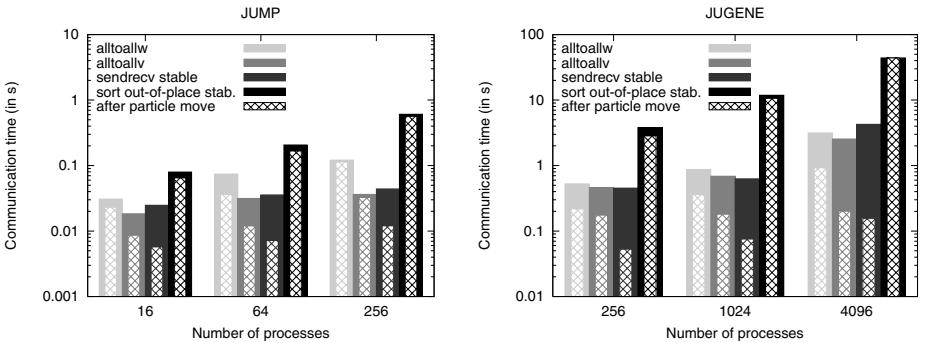


Fig. 3. Communication times for the stable out-of-place implementation variants of `MPI_Alltoall_specific` with particle data from Example B (solid bars) and Example C (patterned bars) on the JUMP (left) and the JUGENE (right) system

On JUGENE, the differences between the in-place and the out-of-place sort implementation variants become smaller for increasing numbers of processes. The communication times of all implementation variants increase for increasing numbers of processes. This is caused by the growing amount of data and the increasing amount of communication required for the distribution of the data. With p processes, the probability for a single element (with a random target process) to stay on the initial process is only about $\frac{1}{p}$.

Figure 3 shows communication times for the stable out-of-place implementation variants of `MPI_Alltoall_specific` depending on the number of processes. Each process contributes 100.000 elements. The results shown with the solid bars use the particle data of Example B. The (smaller) patterned bars show results after the particles have moved corresponding to Example C.

The results of Examples B and C confirm the good performance of the all-to-all implementation variant on both hardware platforms. The performance of the sendrecv implementation with particle data has improved and is close to

alltoally. The alltoallw and the sort implementation are significantly slower on JUMP, especially for larger numbers of processes. On JUGENE, only the sort implementation has significantly higher communication times.

Depending on the number of processes, about 3-17% of the particles need to be distributed to other processes when the particles have moved corresponding to Example C. In comparison to the results with Example B, almost all implementation variants benefit from the reduced number of elements that need to be distributed. The highest reduction in communication time is achieved by the sendrecv implementation, which becomes the best implementation variant of `MPI_Alltoall_specific` for this kind of unbalanced communication.

The overall results show that the alltoally implementation achieves always good performance. This can be attributed to the separation between rearranging the data and sending the data to the target processes with collective communication operations. Therefore, the alltoally implementation benefits from the performance of optimized `MPI_Alltoallv` operations. At least on Blue Gene systems, this operation is known to be very efficient [3]. The sendrecv implementation shows good results too, especially in situations where only a small portion of the data needs to be distributed. The organization of the communication of the sendrecv implementation depends only on the data to be distributed. With large numbers of processes and balanced amounts of communication between the processes, the performance is reduced by the missing adaptation to the hardware platform. The alltoallw implementation suffers from the expensive creation of the indexed data types. Additionally, the underlying `MPI_Alltoallw` operation appears to be less optimized on the chosen hardware platforms. The low performance of the sort implementation is caused by the merge-based parallel sorting algorithm. The algorithm uses multiple steps to send the elements (across intermediate processes) to the target processes. This increases the amount of communication in comparison to the other implementation variants, which send the elements directly to the target processes. The main advantage of the sort algorithm is that it can work in-place.

6 Summary

In this paper, we have proposed a new fine-grained data distribution operation called `MPI_Alltoall_specific` together with several different implementation variants. In comparison to standard MPI all-to-all communication operations, the new operation is more flexible and appropriate for applications like particle codes that require irregular or unbalanced distribution operations. The presented implementation variants possess different capabilities, e.g. in terms of guaranteeing a specific ordering of the output data, preserving the input data, or being able to work in-place by replacing the input data with the output data. We have shown performance results on two supercomputer systems with different sample data sets including particle data. The results demonstrated the advantages and disadvantages of the different implementation variants.

Acknowledgment

The measurements are performed at the John von Neumann Institute for Computing, Jülich, Germany. <http://www.fz-juelich.de/nic>

References

1. Fitch, B.G., Rayshubskiy, A., Eleftheriou, M., Ward, T.J.C., Giampapa, M., Zhhestkov, Y., Pitman, M.C., Suits, F., Grossfield, A., Pitera, J., Swope, W., Zhou, R., Feller, S., Germain, R.S.: Blue Matter: Strong Scaling of Molecular Dynamics on Blue Gene/L. In: Alexandrov, V.N., van Albada, G.D., Sloot, P.M.A., Dongarra, J. (eds.) ICCS 2006. LNCS, vol. 3992, pp. 846–854. Springer, Heidelberg (2006)
2. Liu, W., Wang, C.L., Prasanna, V.: Portable and scalable algorithm for irregular all-to-all communication. *J. Parallel Distrib. Comput.* 62(10), 1493–1526 (2002)
3. Almási, G., Heidelberger, P., Archer, C., Martorell, X., Erway, C., Moreira, J., Steinmacher-Burow, B., Zheng, Y.: Optimization of MPI collective communication on BlueGene/L systems. In: Proc. of the 19th annual Int. Conf. on Supercomputing, pp. 253–262. ACM Press, New York (2005)
4. Pinar, A., Hendrickson, B.: Interprocessor Communication with Limited Memory. *IEEE Trans. Parallel Distrib. Syst.* 15(7), 606–616 (2004)
5. Siegel, S.F., Siegel, A.R.: MADRE: The Memory-Aware Data Redistribution Engine. In: Lastovetsky, A., Kechadi, T., Dongarra, J. (eds.) EuroPVM/MPI 2008. LNCS, vol. 5205, pp. 218–226. Springer, Heidelberg (2008)
6. Höfler, T., Träff, J.L.: Sparse Collective Operations for MPI. In: Proc. of the 23rd Int. Parallel & Distributed Processing Symposium, HIPS Workshop, pp. 1–8 (2009)
7. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard Version 2.1. (2008)
8. Bader, D., Helman, D., JáJá, J.: Practical parallel algorithms for personalized communication and integer sorting. *J. Exp. Algorithmics* 1, Article No. 3 (1996)
9. Knuth, D.: The Art of Computer Programming, Volume III: Sorting and Searching, 2nd edn. Addison-Wesley, Reading (1998)
10. Aluru, S., Sevilgen, F.E.: Parallel Domain Decomposition and Load Balancing Using Space-Filling Curves. In: Proc. of the 4th Int. Conf. on High-Performance Computing, pp. 230–235. IEEE, Los Alamitos (1997)
11. Dachsel, H., Hofmann, M., Rünger, G.: Library Support for Parallel Sorting in Scientific Computations. In: Kermarrec, A.-M., Bougé, L., Priol, T. (eds.) Euro-Par 2007. LNCS, vol. 4641, pp. 695–704. Springer, Heidelberg (2007)

Experiences Running a Parallel Answer Set Solver on Blue Gene

Lars Schneidenbach², Bettina Schnor¹, Martin Gebser¹, Roland Kaminski¹,
Benjamin Kaufmann¹, and Torsten Schaub^{1,*}

¹ Institut für Informatik, Universität Potsdam, D-14482 Potsdam, Germany

² IBM Ireland, Dublin Software Lab, Mulhuddart, Dublin 15, Ireland

Abstract. This paper presents the concept of parallelisation of a solver for Answer Set Programming (ASP). While there already exist some approaches to parallel ASP solving, there was a lack of a parallel version of the powerful *clasp* solver. We implemented a parallel version of *clasp* based on message-passing. Experimental results on Blue Gene P/L indicate the potential of such an approach.

Keywords: Applications based on Message-Passing, Answer Set Programming, Performance evaluation.

1 Introduction

Answer Set Programming (ASP) [1] has become a popular tool for knowledge representation and reasoning. Apart from its rich modelling language, provably more expressive [2] than the ones of neighbouring declarative programming paradigms, as for example Satisfiability checking (SAT), its attractiveness is due to the availability of efficient ASP solvers. In fact, modern ASP solvers rely on advanced Boolean constraint solving technology (cf. [3]), making them competitive with state-of-the-art SAT solvers.

In particular, the sequential ASP solver *clasp* [4,5] is a modern ASP solver incorporating conflict-driven learning, backjumping, restarts, etc. Even though *clasp* is a powerful tool for tackling (NP-hard) search problems, improving the solving time is still a challenge. We thus implemented a parallel *clasp* version, the so-called *claspar* solver, which is portable over a wide range of platforms. Making use of the Message-Passing paradigm, it is particularly designed for distributed memory machines, like clusters are.

2 Related Work

There already are a few approaches to parallel ASP solving. The approaches presented in [6,7] aim at building a genuine parallel solver, running on Beowulf clusters, based on the sequential ASP solver *smodels* [8]. The approach of [9] is to provide a platform for distributing ASP solvers (viz., *smodels* and *nomore++* [10]) in various settings. It supports combinations of Forking, Multi-Threading, and Cluster-Computing using

* Affiliated with Simon Fraser University, Canada, and Griffith University, Australia.

MPI. None of these approaches incorporates modern ASP solving techniques such as conflict-driven learning and backjumping. These are the strengths of the *clasp* solver.

Unlike this, there already are distributed SAT solvers incorporating conflict-driven learning, like *psat* [11], *ysat* [12], *pamira* [13], and *miraxt* [14].

3 Parallelisation Concept

We use the concept of a *guiding path*, which was first implemented within the parallel SAT solver *psato* [15], for workload description. A guiding path defines a path in the search tree from the root node down to the current node with a sub-tree to investigate.

For partitioning the search space captured by an assignment $(v_1, \dots, v_i, \dots, v_n)$, one selects a splittable (Boolean) variable v_i (with $1 \leq i \leq n$) and generates a new guiding path $(v_1, \dots, \overline{v}_i)$ by flipping the truth value assigned to v_i and marking all elements of the path as non-splittable. Also, v_i is marked as non-splittable in the original assignment $(v_1, \dots, v_i, \dots, v_n)$. A systematic application of the guiding path method provides a genuine partition of the search space. This can be exploited for dynamically balancing the workload between distributed solver instances. For handling guiding paths, *clasp* extends the static concept of a *top level* assignment by additionally providing a dynamic variant referred to as *root level*. As with the top level, conflicts within the root level cannot be resolved given that all of its variables are precluded from backtracking.

A heuristic method to estimate the remaining load of a worker is the length of its guiding path. A short guiding path means that only few decisions were made, so that a lot of work may still remain to be done. Hence, we always select the first splittable variable v_i in an assignment $(v_1, \dots, v_i, \dots, v_n)$ to construct a new guiding path $(v_1, \dots, \overline{v}_i)$.

3.1 Master-Worker versus P2P Approach

If we could evenly distribute the workload (i. e. search sub-trees), we could simply start a number of *clasp* solver processes investigating predefined sub-trees. But such an approach is impossible, as we do not know the size of a sub-tree in advance. Starting from any load distribution, sooner or later there will be a load imbalance: some solver instances are still busy, while others are already finished. Thus we need load balancing.

Load balancing techniques can be divided into central, hierarchical, and distributed P2P algorithms. It is important for an efficient distributed load balancing algorithm to know whether a worker is working on a huge branch of the search tree (considered as high load) or whether a worker is almost finished and will ask for additional work soon (low load). This is in advance unknown in our application and varies between inputs.

For the first parallelisation step, we decided to use a central approach where the load distribution is managed by a central component, the *master*. The running *clasp* processes are called *workers*. When a worker has finished processing its search sub-tree, it sends a *work request* to the master. The master then determines another worker that seems to have high load and asks it via a *split request* to split its search space. If the current assignment of the asked worker contains a splittable variable, it returns a new guiding path in terms of a *split response*. The master uses a *work*

cache (see Section 3.3) to reduce the average answer time to work requests. In summary, the tasks of the master are

- receiving a `work request` from a worker w_1 ,
- sending a `split request` to a worker w_2 asking to split its search space,
- receiving a `split response` from worker w_2 ,
- sending a `work response` to worker w_1 ,
- the calculation of overall search statistics (choices, conflicts, restarts, etc.),
- the output of answer sets and statistics, and
- the termination of all workers.

The master has to do a lot of message handling and may become a bottleneck. Therefore we decided to use the master as a dedicated component without an own solver process. Before we describe the concept of work cache management, we discuss the design of worker processes.

3.2 Multi-Threaded Worker versus Single-Threaded Worker

There are two main interactions of a worker with the master: asking for work and replying to split requests.

If a worker runs out of work, it sends a `work request` to the master and waits for a reply, that is, a new guiding path. This can be done using a straightforward blocking communication since there is nothing else to do for an idle worker.

How and when to respond to a `split request` is a more essential decision to make in terms of performance and load balancing. There are two possible approaches to this: the worker can interrupt its processing to handle an incoming request, or an additional dedicated thread handles requests.

In case of a multi-threaded approach, computation and communication could be done concurrently by different threads. The benefit is that the worker will have a good response time, since the communication thread would process a `split request` immediately. This minimises the waiting time for the master and improves the load balancing if another idle worker already waits for the guiding path to be returned. The drawback of such an approach is that threads need exclusive access to shared data structures. For example, the communication thread has to increment the root level, which interferes with the backjumping capability of the solver. As the additional locking would have to be used by both the computation and the communication thread, it would significantly influence the performance of the *clasp* solver. Thus, the multi-threaded approach was out of question for our parallelisation.

A single-threaded worker needs an entry point in the solver code where it is meaningful to test for a `split request`. From the perspective of search, the solver state resulting from conflict analysis is well-suited for making a reasonable decision on whether to split or not. We thus chose the end of the conflict analysis phase, performed to recover from a dead-end encountered in the search. Experiments will have to show the applicability and behaviour of this approach since the frequency of calls to conflict analysis is generally unpredictable (but usually high enough).

3.3 Work Cache Management

The introduction of a work cache (currently holding a single guiding path) solves two problems. It reduces the average answer time to a `work request`, and it avoids the need for a special treatment of the initialisation of all processes.

On start-up, the work cache is initialised with the empty guiding path, representing the entire search space, while all workers request work. One of the workers (first incoming request) will get the entry from the cache and starts processing the search space. The other work requests are appended to a FIFO queue.

The master maintains a list of busy workers. This list contains the number of a worker along with a priority reflecting its estimated workload. To this end, we take the root level of the worker, calculated from the length of its guiding path, as priority. The assumption is that a short guiding path results in a large search sub-tree to process. In the future, other heuristic methods are possible without changing the master, provided that they are based on integer values.

Whenever the work cache is empty, the master sends a `split request` to a busy worker with highest priority (i. e. a worker with smallest priority value). Reconsidering start-up, note that the work cache runs empty immediately when sending the empty guiding path, so that the receiving worker will be the target of a `split request`.

3.4 Implementation

We have implemented our approach in C++ using MPI. The resulting parallel ASP solver is called *claspar*.

Since the size of messages is not known in advance, the workers probe for a `split request` via `MPI_Probe` and `MPI_Iprobe`. The non-blocking `MPI_Iprobe` is used to check for messages after conflict analysis, since immediate return to the solver is required if no messages have arrived. Otherwise, the message size is determined by `MPI_Get_count`, and blocking receives are used to retrieve the message. A non-blocking receive is unnecessary and would rather increase the communication time, since it requires two calls to the MPI library.

Sending messages is always done using blocking `MPI_Send`. Non-blocking sends at a worker's side would require to interrupt the solver and check for completion of the send. In general, overlapping the sending of data would increase the risk of deferred data transmission, causing longer waiting times of the master and workers. Finally, *claspar* uses `MPI_Pack` and `MPI_Unpack` to create and interpret structured messages.

4 Experimental Results

We have conducted experiments to analyse the scalability of the master/worker approach. The examples were run on a Blue Gene/L from 16 up to 1024 cores in Virtual Node mode (two available cores per node are used for computation). Each core runs at 700 MHz and has access to 512 MB RAM. A large number of cores has to be employed in order to traverse a large search space in reasonable time. This requires the application to scale. Additionally, the search space must contain sufficient opportunities to split the

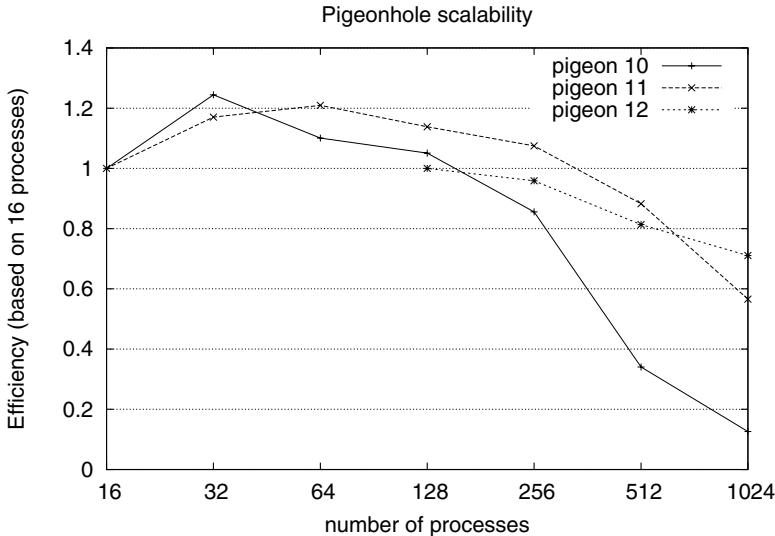


Fig. 1. Efficiency for pigeonhole benchmark

work among the available workers. A few examples were also run on a Blue Gene/P using 2048 and 4096 cores in Virtual Node mode. The Blue Gene/P has four 850 MHz cores per node and 512 MB RAM per core.

It is common practise to stop a run after a certain time. The limit was set to 1200 seconds. Before termination, *claspar* prints out runtime statistics and, in particular, the number of models found. Both time and number of models can be taken to measure scalability.

The choice of examples covers simple and regular problems, such as *pigeonhole*, where the search space has to be traversed completely (since there is no way to put $N+1$ pigeons into N holes allowing only one pigeon per hole). The more sophisticated *blocked queens* benchmark is about putting N queens onto a $N \times N$ checker board such that they do not attack each other, where queens cannot be placed on particular blocked positions. With the *clumpy graphs* benchmark, Hamiltonian cycles are to be found in graphs of a two-layered structure, which gives rise to a non-uniform distribution of solutions in the search space.

4.1 Pigeonhole

Figure 1 shows the efficiency of the pigeonhole examples with 10, 11, and 12 pigeonholes (resp. 11, 12, 13 pigeons). The small example using 10 pigeonholes finishes within about 40–50 seconds with 16 processes. Sustained linear scaling is achieved up to 128 processes. Using 256 processes reduces the time to (no) solution to about 3 seconds. However, 512 processes introduce too much overhead on this small example such that the runtime increases on greater process counts. In contrast, 11 and 12 pigeonholes induce sufficient work to scale to 1024 processes, although efficiency drops

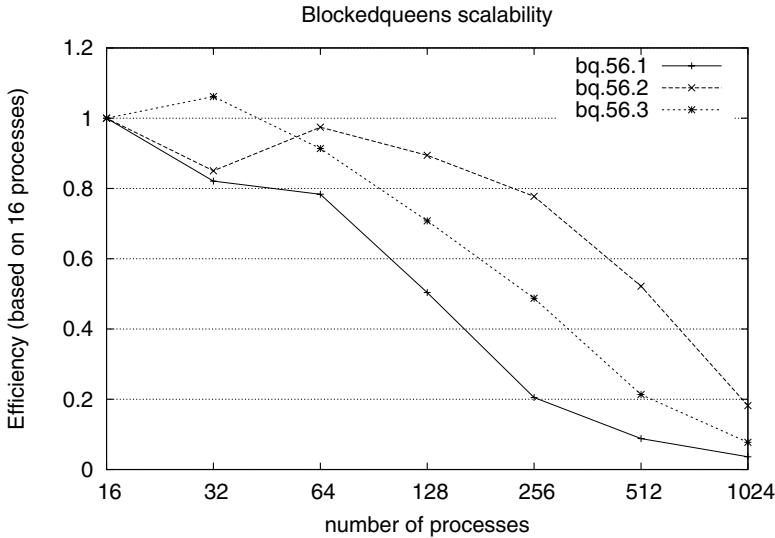


Fig. 2. Efficiency for blocked queens benchmark

with 1024 processes. With 12 pigeonholes, the efficiency is based on the runtime with 128 processes because on smaller number of processes the search space could not be traversed within the time limit of 1200 seconds. The other calculations are based on the time measured with 16 processes.

4.2 Blocked Queens

The scalability of the blocked queens examples, shown in Figure 2, varies between instances. However, the available instances are yet too small for a large number of processes, so that the efficiency rapidly drops below 50 percent. Up to 64 or 128 processes, the examples still scale well. Thus it seems promising to go for larger instances of this class (if they were available).

4.3 Clumpy Graphs

Figure 3 shows the number of detected Hamiltonian cycles in three different graphs of size 9×9 . The experiments were stopped after 1200 seconds. If an experiment has been aborted, the calculation of speedup and efficiency is based on the number of detected cycles. Since *claspar* is able to finish the second example within the time limit when running on more than 256 processes, the efficiency based on the number of detected cycles is omitted for 512 and 1024 processes in the figure. Using 512 processes, finishing the second example takes 1134 seconds, and 625 seconds with 1024 processes. This is a speedup of 1.81 and an efficiency of about 90 percent.

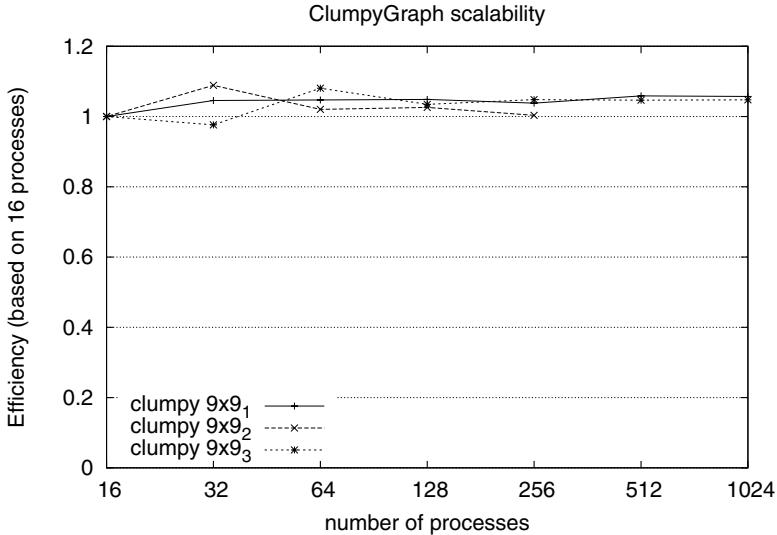


Fig. 3. Efficiency for clumpy graphs benchmark based on number of Hamiltonian cycles found

4.4 Scalability

With the current examples, we could not find a limit for the number of messages the master can handle. The number of messages per second, shown in Table 1, is no indicator of the scalability of the tested environment.

The limiting factor for the above examples is the difficulty in terms of the number of conflicts (encountered dead-ends). The results indicate that the efficiency correlates to the ratio between messages and conflicts. Table 1 shows the ratio and the efficiency with 1024 processes based on the measurement on 16 cores. If the number of messages is more than two orders of magnitude smaller than the number of conflicts, efficiency is good. For one, this correlation is a result of the design of the workers handling incoming requests after conflict analysis. Therefore, the number of conflicts impacts the response time and the load balancing. For another, few messages per conflict indicate that workers perform a significant amount of search before running out of work.

Table 2 shows results for some examples on 2048 and 4096 cores of a Blue Gene/P. The example with 12 pigeonholes does not scale there. A look at the message/conflict ratio (2.86) reveals that it is too small for the larger numbers of processes. In fact, *claspar* completes even the example with 13 pigeonholes on 2048 or more cores within 1200 seconds, while fewer processes could not finish. Therefore, this example was omitted previously. On the clumpy graphs benchmark, the number of detected Hamiltonian cycles for example 09x09_01 and 09x09_03 doubles with the number of processes. This is a promising result for further examples and even larger numbers of processes.

The sometimes super-linear speedup is a phenomenon that was also observed in previous work [9]. A possible explanation is the reduced size of the search sub-trees to

Table 1. Number of messages and conflicts for examples

example	msg/s	msg/conflicts*100	eff. (1024:16)
pigeon10	180387	7.35	0.126
pigeon11	84861	0.91	0.566
pigeon12	9302	0.08	0.71 ¹
blockedqueens.56.1	136966	113.07	0.036
blockedqueens.56.2	129509	18.44	0.182
blockedqueens.56.3	134341	47.31	0.078
clumpy-09x09_01	18.5	0.0014	1.05
clumpy-09x09_02	1113	0.0925	(0.25) ²
clumpy-09x09_03	18.7	0.00066	1.04

Table 2. Results on Blue Gene/P

example	time (2048)	time (4096)	models (2048)	models (4096)
pigeon12	110.593	329.914	0	0
pigeon13	812.751	572.625	0	0
clumpy-09x09_01	1200.061	1200.193	9014355023	18014969357
clumpy-09x09_02	416.835	349.127	2134183512	2134183512
clumpy-09x09_03	1200.061	1200.193	7313232805	14325376947

be processed separately, whose fewer constraints imply less processing time. However, this effect strongly depends on the example and is subject to future investigation.

5 Conclusion and Future Work

We presented a performance analysis of our parallel ASP solver *claspar* on Blue Gene machines with up to 4096 cores. The current results are encouraging, even though they cover only a few problem classes and one cannot assume that *claspar* will behave similarly on every problem. The analyses indicate that *clasp* is well-suited for message-passing parallelisation. This gives us the vision that parallel ASP solving will be useful for even harder problems.

In the current *claspar* version, the communication topology is a star with the master as the centre. This was expected to have a limited scalability. Nevertheless, an efficient design of the master-worker interaction allows us to achieve a scalable behaviour as long as an instance induces a sufficient number of conflicts.

However, since it makes only limited sense to desperately try to achieve linear scalability while the problems and their search spaces usually grow exponentially, this parallel approach was intended to be kept simple. The workers also apply *learning*, i.e. constraints identified on conflicts are locally stored in memory. This advanced technique of the sequential *clasp* solver is planned to be applied to *claspar* (in a distributed manner) in the near future to further speed up the parallel search.

As another future work, we aim at a hybrid version of *claspar* adapted also to multi-core architectures. Between physically distributed workers, learned constraints seeming

¹ Based on 128 processes; see text.

² All solutions found before timeout.

important based on the *clasp* heuristics are subject to exchange via messages, while workers with shared memory use a shared region for the exchange (cf. [14]).

Acknowledgements. This work was supported by DFG under grant SCHA 550/8-1.

References

1. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press, Cambridge (2003)
2. Janhunen, T.: Some (in)translatability results for normal logic programs and propositional theories. *Journal of Applied Non-Classical Logics* 16(1-2), 35–86 (2006)
3. Mitchell, D.: A SAT solver primer. *Bulletin of the European Association for Theoretical Computer Science* 85, 112–133 (2005)
4. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In: Veloso, M. (ed.) *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI 2007)*, pp. 386–392. AAAI Press/MIT Press, Menlo Park (2007)
5. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set enumeration. In: Baral, C., Brewka, G., Schlipf, J. (eds.) *LPNMR 2007. LNCS*, vol. 4483, pp. 136–148. Springer, Heidelberg (2007)
6. Pontelli, E., Balduccini, M., Bermudez, F.: Non-monotonic reasoning on Beowulf platforms. In: Dahl, V., Wadler, P. (eds.) *PADL 2003. LNCS (LNAI)*, vol. 2562, pp. 37–57. Springer, Heidelberg (2003)
7. Balduccini, M., Pontelli, E., El-Khatib, O., Le, H.: Issues in parallel execution of non-monotonic reasoning systems. *Parallel Computing* 31(6), 608–647 (2005)
8. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. *Artificial Intelligence* 138(1-2), 181–234 (2002)
9. Gressmann, J., Janhunen, T., Mercer, R., Schaub, T., Thiele, S., Tichy, R.: On probing and multi-threading in platypus. In: Brewka, G., Coradeschi, S., Perini, A., Traverso, P. (eds.) *Proceedings of the Seventeenth European Conference on Artificial Intelligence (ECAI 2006)*, pp. 392–396. IOS Press, Amsterdam (2006)
10. Anger, C., Gebser, M., Linke, T., Neumann, A., Schaub, T.: The nomore++ approach to answer set solving. In: Sutcliffe, G., Voronkov, A. (eds.) *LPAR 2005. LNCS (LNAI)*, vol. 3835, pp. 95–109. Springer, Heidelberg (2005)
11. Blochinger, W., Sinz, C., Küchlin, W.: Parallel propositional satisfiability checking with distributed dynamic learning. *Parallel Computing* 29(7), 969–994 (2003)
12. Feldman, Y., Dershowitz, N., Hanna, Z.: Parallel multithreaded satisfiability solver: Design and implementation. *Electronic Notes in Theoretical Computer Science* 128(3), 75–90 (2005)
13. Schubert, T., Lewis, M., Becker, B.: Pamira - a parallel SAT solver with knowledge sharing. In: Abadir, M., Wang, L. (eds.) *Proceedings of the Sixth International Workshop on Microprocessor Test and Verification (MTV 2005)*, pp. 29–36. IEEE Computer Society, Los Alamitos (2005)
14. Lewis, M., Schubert, T., Becker, B.: Multithreaded SAT solving. In: *Proceedings of the Twelfth Asia and South Pacific Design Automation Conference (ASP-DAC 2007)*, pp. 926–931 (2007)
15. Zhang, H., Bonacina, M., Hsiang, J.: PSATO: a distributed propositional prover and its application to quasigroup problems. *Journal of Symbolic Computation* 21(4), 543–560 (1996)

Challenges and Issues of the Integration of RADIC into Open MPI

Leonardo Fialho^{1,*}, Guna Santos¹, Angelo Duarte², Dolores Rexachs¹, and Emilio Luque¹

¹ Computer Architecture and Operating Systems Department
University Autonoma of Barcelona. Bellaterra, Barcelona 08193, Spain
`{lfialho,guna}@caos.uab.es, {dolores.rexachs,emilio.luque}@uab.es`
² Departamento de Tecnologia
Universiade Estadual de Feira de Santana. Feira de Santana, Bahia, Brazil
`angeloduarte@ecomp.uefs.br`

Abstract. Parallel machines are growing in complexity and number of components which increases fault probability. Thus, MPI applications running on these machines may not reach completion. This paper presents RADIC/OMPI, which is the integration of RADIC fault tolerance architecture into Open MPI. RADIC/OMPI relies on uncoordinated checkpoints combined with pessimistic receiver-based message logs in a distributed way without the need to use any central or stable elements. Due to this, it assures the application completion automatically and transparently for users and administrators. We concluded that within certain applications RADIC/OMPI provides fault tolerance with an acceptable overhead even in the presence of consecutive faults.

Keywords: Fault Tolerance, High Availability, RADIC, Open MPI.

1 Introduction

Parallel computers are growing in complexity and number of components which increases fault probability. In these machines, Message Passing Interface becomes a *de facto* library used to implement message passing parallel applications. Due to the default fail-stop approach used by the MPI standard, applications which rely on fault-probable computers to run may not reach completion.

As will be presented in section 5 a lot of research has been done to provide automatic fault tolerance for MPI applications. A common approach is to implement fault tolerance on the MPI library. It grants transparency for application developers and parallel machine administrators. RADIC[1] (*Redundant Array of Distributed Independent Fault Tolerance Controllers*) is a rollback/recovery based fault tolerance architecture which has been proposed to be integrated on message passing libraries.

* This research has been supported by the MEC-MICINN Spain under contract TIN2007-64974.

The main challenge during development of this work is to accommodate RADIC characteristics on an established MPI library. Even though RADIC is flexible with respect to state saving strategies, implementations of this architecture must ensure automatic recovery and transparency for the parallel computer administrator and application developers. Furthermore, for the sake of scalability, fault tolerance tasks should be performed in a distributed way. Thus, protection, fault detection, recovery, and system reconfiguration must rely solely on the resources involved in the fault.

This paper presents challenges and issues surrounding the integration of RADIC into a well-known and widely-used message passing library. Open MPI has been selected due to the simplicity of its Modular Component Architecture[2] (MCA) and its checkpoint/recovery support[3].

The content of this paper is organised as follows. Section 2 introduces a brief description of RADIC and Open MPI architectures. Section 3 concerns challenges and issues of architectures integration (RADIC into Open MPI), which is the major contribution of this work. Some experimental evaluations are presented in section 4. Section 5 presents a comparison of related proposals which implement fault tolerance at library level. Finally, conclusions are stated in section 6 along with future work.

2 RADIC Architecture and Open MPI

For better understanding this section presents a short description of RADIC and Open MPI architectures.

2.1 RADIC Architecture

RADIC architecture[1] is based on uncoordinated checkpoints combined with receiver-based pessimistic message logs. *Critical data* like checkpoints and message logs of one application process are stored on other node different from the one in which the application is running. This selection assures application completion if a minimum of three nodes is left operational after n non-simultaneous faults. Even more, concurrent faults are supported if the faulty resource is not involved in recovery (*i.e.* a fault in a node which runs an application simultaneously with a fault in the node which stores its critical data is not supported).

In short, RADIC defines two entities:

- **Observer:** this entity is responsible for monitoring the application’s communications and masks possible errors generated by communication failures. Therefore, the observer performs message logs in a pessimistic way as well as periodically taking an application process checkpoint. Checkpoints and message logs are sent to protectors. There is an observer attached to each application process.
- **Protector:** according to RADIC specifications, each node runs only one protector, which can protect more than one application process. In order to

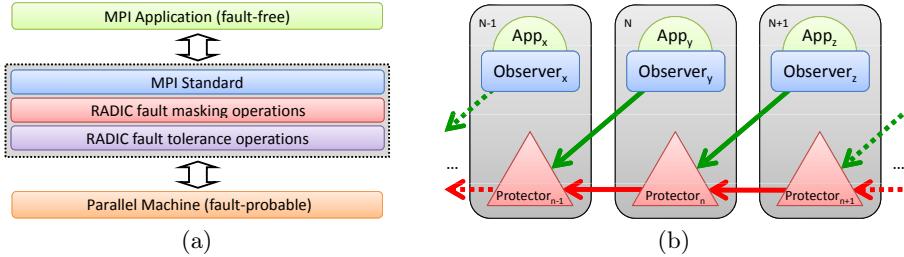


Fig. 1. (a) RADIC layers which provide transparency for applications. (b) Relationship between nodes running an application with RADIC fault tolerance architecture. Diagonal arrows represent critical data flow while horizontal ones represent heartbeats.

protect the application’s critical data, protectors store that on a non-volatile media. In case of failure, the protector recovers the failed application process with its attached observer.

These entities would not run on the same node because, in RADIC, fault tolerance is obtained by joining fault-probable resources. Thus, RADIC does not need any central or stable resource: nodes work together aiming for resilience. As shown in figure 1a, in order to be transparent for the application, observers manage MPI communication masking errors. In addition to masking errors, observers should request a recovery for the faulty application’s protector before retrying communication.

Errors generated due MPI communication attempts are one of RADIC fault detection mechanisms. Moreover, protectors implement a heartbeat/watchdog mechanism between nodes which permits the configuration of the fault detection latency. Figure 1b depicts communication performed by RADIC in order to achieve fault tolerance. Horizontal arrows represent heartbeats. Diagonal arrows represent critical data flow (*i.e.* message logging and checkpoint file transmission). Additionally, figure 1b shows that observers go attached to application processes while protectors are standalone processes.

The major advantage of RADIC is its intrinsic distributed characteristic. No collective operation is needed. Tasks performed by the fault tolerance mechanism involve solely two nodes and do not depend on the number of nodes neither any central element nor unique resource. Thus, RADIC scales according to the application which it protects. Summarising, the RADIC architecture is transparent, scalable, distributed and flexible.

2.2 Open MPI Architecture

Open MPI is a message passing library which follows MPI2 standard. It uses a Modular Component Architecture (MCA)[2] which comprises three functional layers: *i)* *root* which provides basic MCA functions, *ii)* *framework* which specifies functional interfaces to specific services (*e.g.* message transport layer),

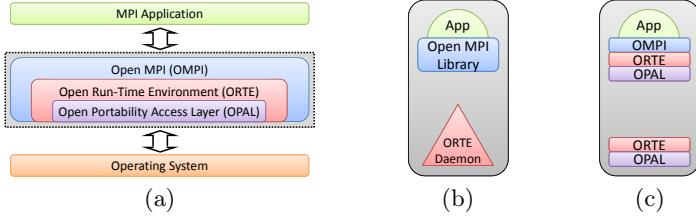


Fig. 2. (a) Open MPI sections interdependency relationship. (b) A typical Open MPI node running an application. (c) Structure of ORTE daemon and Open MPI library.

iii) *component* which is a specific implementation of a particular framework (*e.g.* TCP as transport layer).

Frameworks are organised in three dependent sections: *i*) *Open MPI* (OMPI) which interfaces with applications through MPI API, *ii*) *Open Run-Time Environment* (ORTE) which provides a parallel run-time environment and other services for the upper section, *iii*) *Open Portable Access Layer* (OPAL) which is an abstraction layer to some operating system function. Figure 2a shows the interdependency between application, operating system and Open MPI sections.

During application launching an *ORTE daemon* is instantiated in each node. These daemons communicate between themselves in order to create a parallel environment. Once the parallel environment is operational, applications are launched and start computation. A typical Open MPI node is shown in figure 2b. Structurally, as depicted in figure 2c, the ORTE daemon is composed by ORTE and OPAL sections. In turn, the library code which assists the application is composed of OMPI, ORTE and OPAL sections.

OMPI provides a three-layer framework stack for MPI communication. On top of this stack is the *Point-to-point Management Layer* (PML) framework which allows wrapper stacking. The lower layer is the *Byte Transfer Layer* (BTL) framework which components implement specific communication drivers. In the middle, there is the *BML Management Layer* (BML) framework whose provides caching for lower components.

Open MPI implements a manual fault tolerance mechanism based on a command line tool which dispatches a checkpoint request to the *Snapshot Coordinator* (SnapC) which starts checkpoint coordination. Applications call the *Checkpoint/Restart Coordination Protocol* (CRCP) in order to synchronise pending messages between them. Finally, CRCP calls the *Checkpoint Restart Service* (CRS) which instantiates the checkpoint library in order to create the checkpoint file [3]. Files are transferred through the *File Manager* (FileM) framework, and all these communications use the *Out of Band* (OOB) framework.

3 Integrating RADIC into Open MPI

To implement RADIC into Open MPI several decisions which affect performance and operation should be taken into account. Since RADIC has no implementation

constraints it is flexible when it is implemented and it allows new state saving techniques to be incorporated.

Due to the observer's behaviour, it has been integrated as a PML wrapper component. This decision assures the existence of one observer for each application process. On the other hand, protectors have been integrated on the ORTE daemon. In order to define initial observer/protector mapping a simple algorithm has been used. From the observer's point of view its protector is defined as the next logical node. In case the observer is running on the last node, its protector is located on the first node (*i.e.* observers running on node 2 use node 3's protector and observers running on node n use the first node's protector).

In the following we describe how RADIC fault tolerance tasks have been integrated into Open MPI, highlighting decision points and challenges found on the way. The resulting implementation of this work was named RADIC/OMPI. With the exception of FileM, no other fault tolerance framework has been used to take checkpoints. Inter and intra-process coordination is highly coupled across several frameworks which hinder modifications to make it uncoordinated.

3.1 Uncoordinated Checkpointing Operation

A well-known problem related to uncoordinated checkpointing refers to communication channels used by MPI. They must be closed, or at least, unused during checkpoint operation. It ensures that no in-transit messages will be truncated or lost. To avoid this situation observers avoid checkpointing while calls to MPI communication functions are in course.

Checkpointing is initiated through a call-back function triggered by a timer (checkpoint interval). This function verifies if no communication is in course and blocks subsequent requests in order to call the checkpoint library for checkpoint file creation. Once the checkpoint file is created, it is transferred to the protector using FileM framework. After checkpoint file transmission, the observer goes back to allowing message transfers. A checkpoint reception induces a garbage collection of message logs stored on the protector.

3.2 Message Logging Operation

Coding the observer as a PML wrapper assures that all point-to-point and collective communications performed by applications could be triggered. Furthermore, MPI calls are distributed between different frameworks. Thus, Open MPI's request subset has been modified to perform a log of calls to functions like `MPI_Test` and `MPI_Wait`.

The log of MPI communication functions are sent to protectors using *Out Of Band* (OOB) framework. The protector stores log data in a memory structure or a log file on disk, according to RADIC fault tolerance configuration.

3.3 Fault Detection Mechanism

First, on RADIC, faults are detected during communication tries, which starts an error management procedure. It supposes modifications on lower layers in order

to throw up those errors to PML. RADIC also defines a heartbeat/watchdog mechanism between protectors, which is useful in scenarios where faults cannot be detected by application communication errors.

Protectors set timers in order to send heartbeats to the next logical node (*e.g.* node 1 send heartbeats to node 2) which reset its watchdog timer. Once heartbeats cannot be sent or the watchdog timer expires a recovery procedure is triggered. It has been implemented on the command processor of ORTE daemons.

3.4 Fault Management

When a process fails Open MPI's default action is to dispatch application finalization. It is not in compliance with RADIC behaviour. To avoid application finalization a new error manager component has been implemented which permits protectors to start the recovery procedure.

In order to avoid failures for applications during communication attempts the observer handles errors generated by lower layers. Thence sender's observer contacts the destination application's protector in order to query the application state. At this moment, if the protector confirms the error, it starts the recovery procedure immediately and answers the observer ordering a communication retry. Errors could increase message delivering latency. Furthermore, the observer never returns an error to the application process.

3.5 Recovery Procedure and System Reconfiguration

Recovery procedure is performed in three phases. The first one is managed by the protector which restarts the application process from its previous checkpoint. The second phase is the log processing, in which the recovered observer draws messages from the log and discards repeated ones. Finally, the third phase corresponds to reconfiguration in which, *i*) the recovered observer choose a new protector to send its checkpoint and, *ii*) all protectors affected by the failure re-establish the heartbeat/watchdog mechanism.

Failed process restarting is similar to initial application launching but instead of using the original application filename the checkpoint library is used to restart it from the previous checkpoint. Due to the faulty process migration to another node, contact information stored in the BTL cache of survivor processes is not reliable. When these processes try to communicate with the recovered one they will fail. Thus, its respective protector will be contacted and will answer with the new recovered process location. Hence, the application's cache can be updated on demand without a global synchronisation. This procedure has required changes in lower MPI communication layers.

RADIC architecture does not impose where a failed application processes should be recovered. Actually, a failed process is recovered on the node where its critical data is stored.

3.6 Conflicts between RADIC Architecture Concepts and Open MPI Characteristics

In order to achieve scalability, RADIC architecture is completely distributed and uses no collective operations. In turn, Open MPI uses a collective operation which disseminates contact information during application checkpointing (PRE-PARE/CONTINUE/RESTART). In order to avoid such collective operations, protectors and observers use a query protocol to discover the new location of recovered processes. This procedure is performed independently and on demand by each observer.

This RADIC implementation over Open MPI provides support solely to MPI-1 applications, since it does not support dynamic process creation.

4 Experimental Evaluation

This experimental evaluation analyses the behaviour of new fault tolerance operations integrated into Open MPI. Protection operations such as checkpointing and message logging are analysed in isolation. In order to analyse the impact of these operations on applications they have been executed in a fault-free environment as well as in the presence of faults.

This paper presents three evaluations of RADIC implementation over Open MPI, *i*) latency introduced by the log operation according to message size, *ii*) checkpointing performance according to application process size, *iii*) execution time in either fault-free and faulty scenarios. First analysis uses *NetPIPE* while others use applications from the *NAS Parallel Benchmark*.

Experiments run on a 32 node Linux cluster equipped with two Dual-Core Intel Xeon processors running at 2.66GHz. Each node has 12 GBytes of main memory and a 160 GByte SATA disk. Nodes are interconnected via two Gigabit Ethernet interfaces.

4.1 Message Logging Performance

The objective of this evaluation is to provide a magnitude order about the overhead introduced by message logging. In order to evaluate message logging performance the NetPIPE evaluator has been used. Measurements were taken using asynchronous transmissions in only one direction. On RADIC configuration, checkpointing has been disabled.

Figure 3a depicts message delivery latency when using one communication channel. The lower curve is presented for comparison reasons and represents the MPI communication latency without logging. The upper curve represents the MPI communication latency while performing a log. The gap existent between both curves is the message logging overhead.

Since Open MPI can perform load balance between available channels for MPI communication, the analysis has been made using two channels also. As shown in figure 3b, once the message size gets bigger than the TCP packet limit

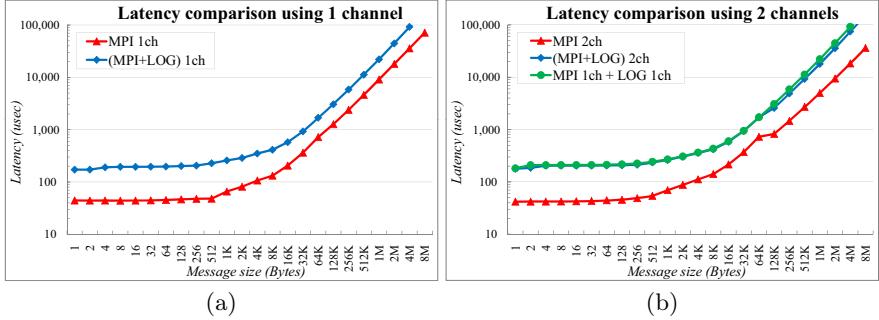


Fig. 3. Latency with and without logging while using one channel (a) and two (b)

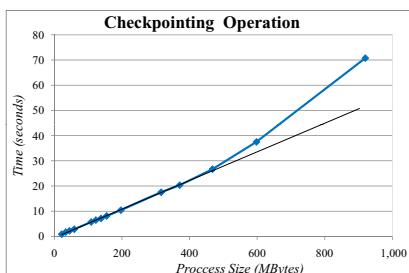
(64 KBytes) it can be load balanced. Log traffic can not be load balanced because the framework used to send the log does not provide this feature.

The OOB framework does not assure that data is available on destination after leaving `send_buffer` function. Thus, an acknowledgement message is necessary. Consequently, an additional step has been introduced on the logging procedure, increasing its overhead. Because of this, the overhead introduced is the time needed to send the log to the protector plus the latency of the acknowledgement message.

4.2 Checkpointing Performance

Checkpointing procedure is composed of two operations: file creation and transfer. The time needed to fork a process and copy it to another node varies according to process size, hence, checkpointing performance depends on process size.

Figure 4 depicts checkpointing performance for different process sizes. Values are expressed in seconds and megabytes. For small process sizes checkpointing performs linearly. After network saturation the time needed to transfer files starts to grow. Since, checkpointing is uncoordinated and storage is distributed, checkpointing one process should not impact on other processes.



Benchmark	Process Size (MB)	Time to Checkpoint (s)
LU A 4 nodes	22	0.893
LU B 4 nodes	59	2.860
SP B 4 nodes	109	5.704
LU C 4 nodes	197	10.41
FT B 4 nodes	468	26.68
IS C 4 nodes	598	37.51
MG C 4 nodes	920	70.75

Fig. 4. Checkpointing performance according process size. Values are expressed in seconds and megabytes.

4.3 Analysis of the Impact of Fault Tolerance on NAS

To analyse the impact of fault tolerance NAS applications have been used. Class C and D of BT, LU and SP were chosen because they require significant time to complete. Since class D applications run for between 20 minutes and 2 hours, to simulate a faulty scenario, a probability of fault has been defined as 1 fault after every 30 minutes. These failures have been introduced killing the application process and its ORTE daemon.

To configure the checkpoint interval the following model[4] has been used $I = \sqrt{\frac{2k_0}{\alpha}}$. Where I is the optimal checkpoint interval, k_0 is the time spent to create and transfer checkpoint, and α is the probability of failure. Table 1 presents the calculated checkpoint intervals used to execute class D of NAS applications. Values are expressed in minutes and megabytes.

Figure 5 depicts the performance of NAS applications while using Open MPI with and without RADIC. For comparison purposes the running time of class D applications running in a fault-free environment are also presented.

Table 1. Checkpoint intervals used to execute class D of NAS applications. Values are expressed in minutes and megabytes.

App	#	Running Time	Process Size	Ckpt. Interval	App	#	Running Time	Process Size	Ckpt. Interval
BT D	16	43.79	1,980	21.58	LU D	8	103.84	1,747	19.46
	25	29.58	1,400	16.28		16	49.69	1,061	13.13
SP D	16	55.01	1,715	19.17		32	20.63	722	9.91
	25	40.82	1,251	14.90					

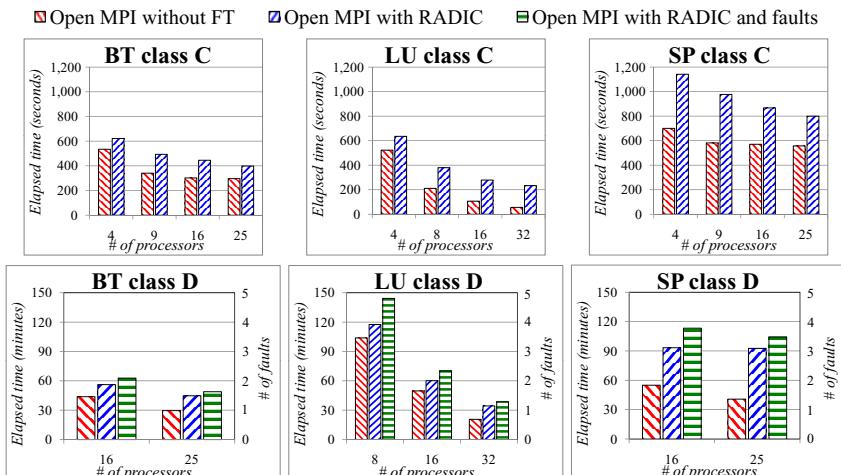


Fig. 5. Performance of NAS applications class C and D while using Open MPI with and without RADIC fault tolerance

The difference between executions with and without fault tolerance is the overhead introduced by fault tolerance protection tasks. Class C applications are too short to justify the use of fault tolerance and in this scenario they are not affected by faults. Nevertheless, class C applications depict better the overhead introduced by fault tolerance.

According to BT and SP class C executions without fault tolerance, there is no gain in performance for more than 9 nodes. Meanwhile, due to process size reduction the checkpointing overhead decreases. In turn, LU class C without fault tolerance scales well from 4 to 32 nodes. However, while using fault tolerance the message logging procedure mitigates the scaling gain.

On SP class D execution message logging makes the application communication bound. Thus, no gain is obtained increasing the number of nodes from 16 to 25 while using fault tolerance. In turn, without fault tolerance the application experiences a modest gain. Meanwhile, class D of BT and LU with fault tolerance scale better than SP. It occurs because applications maintain their computation bound behaviour while performing fault tolerance operations.

5 Related Work

Much effort has gone into providing fault tolerant MPI libraries. Proposals normally rely on rollback/recovery techniques. For state saving, those techniques perform, in the great majority of cases, coordinated checkpointing or uncoordinated checkpointing combined with message logging.

MPICH-V2[5] assures transparency and is automatic, but it has significant differences with RADIC/OMPI. The first difference is the message logging approach. MPICH-V2 performs sender-based message logging while RADIC/OMPI performs receiver-based. The second and major difference refers to the strategy used to store critical data. MPICH-V2 relies on stable resources to store critical data, while RADIC/OMPI does not require any special and/or stable resource on a parallel computer. Another difference between MPICH-V2 and RADIC/OMPI concerns the distributed mechanism used for state saving and recovering. Theoretically, the distributed approach grants to RADIC/OMPI more scalability than MPICH-V2, without the need to introduce any additional resources.

As aforementioned, Open MPI already implements a fault tolerance mechanism [3]. Nevertheless, neither state saving nor recovery are performed automatically. They are manual operations carried out by the user or administrator. Additionally, there is another important difference with respect to the checkpointing strategy. The Open MPI strategy relies on coordinated checkpointing

Table 2. Characteristics of fault tolerant MPI libraries

Library	Detection	Recovery	Storage	Fault Tolerance Strategy
MPICH-V2	automatic	automatic	centralized	uncoord ckpt+logging
Open MPI	n/a	manual	n/a	coord ckpt
RADIC/OMPI	automatic	automatic	distributed	uncoord ckpt+logging

while RADIC/OMPI uses uncoordinated, which represents an important scalability issue. Even more, Open MPI strategy does not define any storage policy. It is up to the user or administrator to assure availability of critical data.

Table 2 summarises RADIC/OMPI, MPICH-V2 and Open MPI original characteristics.

6 Conclusion and Future Works

This paper has presented RADIC/OMPI which is the integration of RADIC fault tolerance architecture into Open MPI. It grants to Open MPI new fault tolerance capabilities like uncoordinated checkpointing combined with pessimistic receiver-based message logging. Thus, Open MPI now can perform fault protection, detection and recovery automatically and transparently for application developers and parallel machine administrators.

Experimental evaluation makes clear that to achieve a better performance in message logging the high priority communication framework used to transfer messages at MPI level (PML) should be available for message logging operations. Furthermore, in applications which present computation bound behaviour the overhead introduced by message logging is acceptable. The distributed characteristic of RADIC assures that while increasing the number of nodes the overhead introduced by fault tolerance tasks does not increase as well. In this sense RADIC/OMPI assures the application's scalability.

Future work should consider the use of spare nodes according RADIC2[6] specification. This improvement aims to maintain original performance and process mapping as well as provides interfaces to process migration and preventive maintenance.

References

1. Duarte, A., Rexachs, D., Luque, E.: Increasing the cluster availability using RADIC. *Cluster Computing*, 1–8 (2006)
2. Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B.W., Lumsdaine, A., Castain, R.H., Daniel, D.J., Graham, R.L., Woodall, T.S.: Open MPI: Goals, concept, and design of a next generation MPI implementation. In: Kranzlmüller, D., Kacsuk, P., Dongarra, J. (eds.) EuroPVM/MPI 2004. LNCS, vol. 3241, pp. 97–104. Springer, Heidelberg (2004), <http://www.springerlink.com/content/rdd34k92rx3nqx1>
3. Hursey, J., Squyres, J., Mattox, T., Lumsdaine, A.: The Design and Implementation of Checkpoint/Restart Process Fault Tolerance for Open MPI. In: IPDPS, March 26–30, 2007, pp. 1–8 (2007)
4. Gropp, W., Lusk, E.: Fault Tolerance in Message Passing Interface Programs. *Int. J. High Perform. Comput. Appl.* 18(3), 363–372 (2004)
5. Bouteiller, A., Cappello, F., Herault, T., Krawezik, K., Lemarinier, P., Magniette, M.: MPICH-V2: a Fault Tolerant MPI for Volatile Nodes based on Pessimistic Sender Based Message Logging. *Supercomputing* 25 (2003)
6. Santos, G., Duarte, A., Rexachs, D.I., Luque, E.: Providing non-stop service for message-passing based parallel applications with RADIC. In: Luque, E., Margalef, T., Benítez, D. (eds.) Euro-Par 2008. LNCS, vol. 5168, pp. 58–67. Springer, Heidelberg (2008)

In-Memory Checkpointing for MPI Programs by XOR-Based Double-Erasure Codes^{*}

Gang Wang, Xiaoguang Liu, Ang Li, and Fan Zhang

Nankai-Baidu Joint Lab, College of Information Technical Science,
Nankai University, 94 Weijin Road, Tianjin, 300071, China
wgzwp@163.com, liuxg74@yahoo.com.cn,
{megathere, zhangfan555}@gmail.com

Abstract. Today, the scale of High performance computing (HPC) systems is much larger than ever. This brings a challenge to fault tolerance of HPC systems. MPI (Message Passing Interface) is one of the most important programming tools for HPC. There are quite a few fault-tolerant extensions for MPI, such as MPICH-V, StarFish, FT-MPI and so on. Most of them are based on on-disk checkpointing. In this paper, we apply two kinds of XOR-based double-erasure codes - RDP (Row-Diagonal Parity) and B-Code to in-memory checkpointing for MPI programs. We develop scalable checkpointing/recovery algorithms which embed erasure code encoding/decoding computation into MPI collective communications operations. The experiments show that the scalable algorithms decrease communication overhead and balance computation effectively. Our approach provides highly reliable, fast in-memory checkpointing for MPI programs.

Keywords: MPI, fault-tolerant, erasure codes, collective communication.

1 Introduction

Today, the scale of High performance computing (HPC) systems is much larger than ever. 54.8% of Top500 systems are composed of 129-512 processors in June 2003. In November 2007, 53.6% of Top500 systems are composed of 1025-2048 processors. The fastest system in 2007 consists of more than twenty thousand processors [1].

A challenge to the systems of such a scale is how to deal with hardware and software failures. Literal [2] reports the reliability of three leading-edge HPC systems: LANL ASCIS Q (8192 CPUs) has a MTBI (Mean Time Between Interrupts) of 6.5 hours, LLNL ASCI White (8192 CPUs) has a MTBF (Mean Time Between Failures) of only 5.0 hours, and Pittsburgh Lemieux (3016 CPUs) has a MTBI of 9.7 hours. In order to deal with such frequent failures, some fault tolerance mechanisms must be considered in both hardware and software design.

* This paper is supported partly by the National High Technology Research and Development Program of China (2008AA01Z401), RFDP of China (20070055054), and Science and Technology Development Plan of Tianjin (08JCYBJC13000).

MPI (Message Passing Interface) [3] is one of the most important programming tools for HPC. Fault-tolerant MPI is obviously necessary for users to run jobs reliably on large HPC systems. Some fault-tolerant extensions had been developed for MPI [4, 5, 6, 7]. Their common fault tolerant method is checkpointing. We have studied MPI in-memory checkpointing technique based on erasure codes [10]. In this paper, we apply two kinds of XOR-based double-erasure codes - RDP and B-Code to in-memory checkpointing. Scalable checkpointing/recovery algorithms combining erasure code encoding/decoding and collective communication operations are developed. The experiments show that the scalable algorithms decrease communication overhead and balance computation effectively.

2 Fault Tolerant MPI and Checkpointing

Today's HPC applications typically run for several days, even several weeks or several months. To deal with the frequent system failures, checkpointing technique is generally used. Checkpoints - the states of processes are written into nonvolatile storage (often hard disk) periodically. If a process failure occurs, the last checkpoint is read and then all (surviving and re-spawned) processes roll back to the latest state. The current MPI specification doesn't refer to any fault tolerance strategy. Therefore, quite a few extensions had been developed to help programmers handle failures more conveniently and efficiently. Cocheck [4] is the first checkpointing extension for MPI. It sits on top of the message passing library. Its shortcoming is global synchronization and implementation complexity. Starfish [5] implements checkpointing at a much lower level. It uses strict atomic group communication protocols, and thus avoids the message flush protocol of Co-Check. MPICH-V uses a coordinated checkpoint and distributed message log hybrid mechanism [6]. Several MPICH-V architectures with different features are developed.

Our work is based on FT-MPI [7] which is another important MPI fault tolerant implementation. It extends some of the semantics of MPI for allowing the application the possibility to survive process failures. FT-MPI provides four "communicator modes" to deal with the status of MPI objects after recovery and two "communication modes" to deal with on the fly messages. FT-MPI handles failures typically in three steps: 1) the run-time environment discovers failures; 2) all surviving processes are notified about these events; 3) the MPI library and the run-time environment are recovered, and then the application recovers its data itself. So, programmers are responsible for checkpoints recording in the fault-free mode and application data recovery after MPI system level recovery.

Apparently, disk write/read operations are the major source of overhead in on-disk checkpoint systems [8]. Diskless checkpointing technique, or so-called in-memory checkpointing, stores checkpoints in the main memory instead of hard disks. Because checkpoints are stored in the address space of the local process, process/node failures induce checkpoint loss. Generally erasure coding techniques are used to protect checkpoints. An erasure code for storage systems is a scheme that encodes the content on n data disks into m check disks so that the system is resilient to any m disk failures [11]. In checkpointing context, processes correspond to disks, n working processes (responsible for original HPC tasks and checkpoints maintenance) correspond to n

data disks, and m redundant processes (dedicated for check information maintenance) correspond to m check disks. It seems that diskless checkpointing introduces extra communication and computation overhead. But in fact, in order to tolerate node failures, on-disk checkpointing often relies on reliable distributed or parallel storage systems which generally are also based on erasure codes [8].

There are two kinds of in-memory checkpointing techniques: one treats checkpoint data as bit-streams [8] and the other treats checkpoint data as its original type [9] (generally floating-point numbers in scientific applications). The advantage of the former is that it can introduce erasure coding schemes smoothly and has no round-off errors. The advantage of the latter is that it is suitable for heterogeneous architectures and can eliminate local checkpoints for some applications. Our work uses the former strategy. However it can be translated into the latter strategy easily.

3 Erasure Codes

The most common used erasure codes in diskless checkpointing are mirroring and simple parity [8]. They are also the most popular codes used in storage systems known as RAID-1 and RAID-4/5 [12]. As the size of HPC systems becomes larger and larger, multi-erasure codes are necessary to achieve high reliability. But unfortunately, there is no consensus on the best coding technique for $n, m > 1$.

The best known multi-erasure code is Reed-Solomon code [13]. It is the only known multi-erasure code suitable for arbitrary n and m . RS code is based on Galois Field, thus the computational complexity is a serious problem. A real number field version of RS code has been used in diskless checkpointing [9].

Another kind of multi-erasure is so-called parity array codes. They arrange the data and parity symbols into an array, and divide symbols into overlapping parity groups, hence the name. Array codes are inherently XOR-based, thus are far superior to RS codes in computational complexity. In this paper, we focus on RDP (Row-Diagonal Parity) - a double-erasure horizontal code with dependent symbols [14]. “Horizontal” means that some disks contain only data symbols and the others contain only parity symbols. Its opposite - “Vertical” means that the data and parity symbols are stored together. Fig 1.a shows the 6-disk RDP code. A standard RDP code with $(p+1)$ disks can be described by a $(p-1) \times (p+1)$ array. The parity groups are organized along horizontal and skew diagonal directions. D_{ij} denotes the data symbol that participates in the i^{th} horizontal parity P_i and the j^{th} diagonal parity Q_j . D_i denotes the data symbol

disk0	disk1	disk2	disk3	disk4	disk5
D_{00}	D_{01}	D_{02}	D_{03}	P_0	Q_0
D_{11}	D_{12}	D_{13}	D_1	P_{10}	Q_1
D_{22}	D_{23}	D_2	D_{20}	P_{21}	Q_2
D_{33}	D_3	D_{30}	D_{31}	P_{32}	Q_3

a. The 6-disk RDP code

disk0	disk1	disk2	disk3	disk4	disk5	disk6
D_{15}	D_{25}	D_{01}	D_{02}	D_{03}	D_{04}	D_{05}
D_{24}	D_{34}	D_{35}	D_{45}	D_{12}	D_{13}	D_{14}
P_0	P_1	P_2	P_3	P_4	P_5	D_{23}

b. A 7-disk B-Code

Fig. 1. RDP code and B-Code

that participates in the only i^{th} horizontal parity. P_{ij} denotes the horizontal parity that also acts as a data member of another diagonal parity Q_j - they are “dependent parity symbols”. RDP achieves optimal encoding performance and good decoding performance. Note that p must be a prime number. But this limitation can be removed by deleting some data disks (assuming they contain nothing but zeros). Another advantage of RDP is that it has been generalized to more than 2 erasures [15].

Another array code used in our work is B-Code [16]. It’s a double-erasure vertical code. B-Code has no prime size limitation because it is constructed by perfect one-factorizations (P1F) of complete graphs. There is a widely believed conjecture in graph theory: every complete graph with an even number of vertices has a P1F [17]. Fig 1.b shows a 7-disk B-Code. D_{ij} denotes the data symbol that participates in parities P_i and P_j . The number of disks p is an odd number. The last disk contains $(p-1)/2$ data symbols. Every other disk contains $(p-1)/2-1$ data symbols and 1 parity symbol. We use B_{2n+1} denotes this kind of B-Codes. Deleting the last disk produces a B-Code with even number of disks. This kind of B-Codes is denoted by B_{2n} . That is to say, B-Code exists for all sizes if P1F conjecture holds. B-Code achieves optimal encoding and decoding performance. Another advantage of B-Code is its inherent distributed structure. Because the parities are scattered over all disks, communication and computation workload are naturally distributed evenly.

We chose RDP and B-Code because they both have good encoding/decoding performance and perfect parameter flexibility - they can be applied to any number of MPI processes - this is obviously significant for the practice.

4 Scalable Checkpointing and Recovery Algorithms

4.1 Encoding and Decoding

FT-MPI system is responsible for failure detection and MPI environment rebuilding. We just need to add checkpointing and recovery functions into user applications. In our scenario, checkpointing and recovery are essentially erasure codes encoding and decoding respectively. So we first examine RDP and B-Code encoding/decoding.

RDP encoding is straightforward. The data symbols in the same row (with the same in-disk offset) are XORed into the horizontal parity symbol in the same row, and then the data symbols and the horizontal parity symbol in the i^{th} skew diagonal (the sum of the row index and the column index equals to i , $0 \leq i \leq p-2$) are XORed into the i^{th} diagonal parity symbol.

Unlike RDP, B-Code doesn’t guarantee regular structure. So we can’t determine the two parity symbols of a data symbol according to its row and column indices directly. We store the mapping from the data symbols to the parity symbols into a table g . $g[i, j, k]$ stores the index of the i^{th} parity symbols of the j^{th} data symbol in the k^{th} disk. Given a B-Code instance, we can calculate g easily. Therefore encoding is performed by XORing every data symbol into all of its parity symbols according to g .

Decoding is somewhat complicated. Both RDP and B-Code encoding can be described by a matrix-vector multiplication, therefore decoding is just a matrix inversion followed by a matrix-vector multiplication [18]. But this method inherently has

non-optimal computational complexity and is hard to be parallelized. A better way is chained decoding.

Single-erasures in a RDP coding system are easy to recover. If one of the parity disks fails, decoding is just (half) encoding. Otherwise, the failed data disk is recovered through the horizontal parity disk. Double-erasures involving the diagonal parity disk are also trivial - another failed disk is recovered through horizontal parity groups and then the diagonal parity disk is re-encoded. The most complicated double-erasures are those excluding the diagonal parity disk. In this case, the horizontal parity disk is regarded as a data disk. Observing Fig 1.a, we can see that each data disk except the first one touches only $(p-2)$ diagonal parities (touching a parity means that contains symbols belonging to the parity), and every data disk misses different diagonal parity. Thus, a pair of data disks including the first data disk touches $(p-2)$ diagonal parities twice and another diagonal parity once; a disk pair excluding the first data disk touches $(p-3)$ diagonal parities twice and other 2 diagonal parities once. Anyway, parity group(s) losing only one symbol always exists. We can start decoding from this kind of parities, and then recover the lost symbols using horizontal and diagonal parity groups alternatively. For example, if a double-erasure (disk0, disk1) occurs in the RDP coding system shown in Fig 1.a, we recover D_{00} by XORing all surviving symbols in the parity group Q_0 , then recover D_{01} using P_0 , and then recover D_{11} using Q_1 , and so on, finally recover D_3 using P_3 . If a double-erasure excluding the first data disk occurs, decoding goes along two paths. Because RDP has a very regular structure, we can deduce starting points and directions of the decoding chains easily.

B-Code decoding also can be done in chained method. “1-missing” parities and decoding chains are determined according to the mapping table g . For example, the two decoding chains in the double-erasure (disk0, disk1) in the B-Code system shown in Fig 1.b are “ $D_{34} \rightarrow D_{24} \rightarrow D_{25} \rightarrow D_{15} \rightarrow P_1$ ” and “ P_0 ”.

4.2 Checkpointing and Recovery Algorithms

The last section outlines the basic RDP and B-Code encoding/decoding methods. In fault-tolerant MPI context, they should be translated into a distributed style, particularly should be merged into MPI collective communication primitives. In this section, we only focus on checkpointing systems based on standard RDP codes and B_{2n} . Those based on shortened RDP codes and B_{2n+1} can be dealt with in a similar way. Moreover, we only concern with double-erasures excluding the diagonal parity disk, because other double-erasures and single-erasures are much easier.

A plain idea is appointing a root process (generally a redundant process or a re-spawned process) to execute all encoding/decoding computation. Fig 2.a depicts plain RDP checkpointing algorithm. The first redundant process gathers all checkpoint data from n working processes, then calculates horizontal and diagonal parities locally, and finally sends diagonal parities to the second redundant process. This algorithm can be transformed into recovery algorithm simply by exchanging the role of the re-spawned processes and the redundant processes and replacing local encoding by local decoding. Suppose each working process contributes s bytes checkpoint data and linear time gather algorithm [19] is used. Because per data word cost of RDP encoding/decoding is about 2 XOR operations, the time complexity of plain RDP

checkpoint/recovery algorithm is $O(sn)$. The algorithm also requires an extra buffer of size $O(sn)$ which is obviously induced by local encoding/decoding.

Fig 2.b shows the plain B-Code Checkpoint algorithm. The first working process collects checkpoint data, then encodes them into parities, and finally scatters parities over all processes. This algorithm also can be converted into recovery algorithm. The time complexity and extra memory requirement are $O(sn)$ too.

Apparently, the plain algorithms lead to serious load imbalance and unnecessary communication traffic. In order to distribute computation workload evenly and minimize communication traffic, a natural idea is embedding encoding/decoding into MPI collective communication operations. This idea is actually feasible. The basic operation in encoding and decoding is XORing data from all processes into parity at a single destination process. This is in fact a typical *all-to-one reduction* [19]. Therefore a RDP checkpointing can be accomplished by two successive reductions. Fig 3.a shows a checkpointing procedure in a fault-tolerant system based on the 6-disk RDP code. Horizontal parity reduction is trivial. Diagonal parity reduction requires a buffer pre-processing. The i^{th} process shifts its buffer to the up by i steps (with wraparound) to align data with parity. Because every process except the first one misses a diagonal parity, the corresponding area in the buffer is zeroed. If logarithmic time reduction algorithm [19] is used, the time complexity is $O(slogn)$. Total extra memory requirement is also $O(sn)$, but per process extra memory requirement is only $O(s)$.

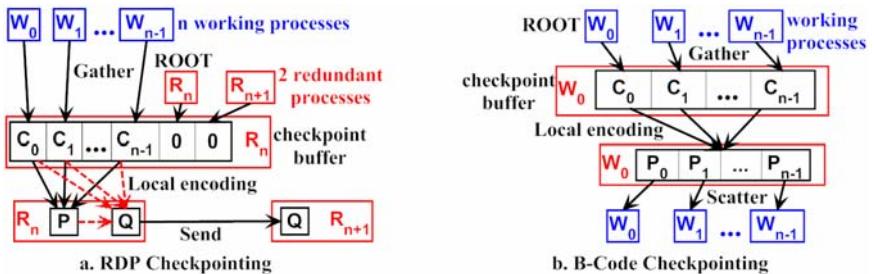


Fig. 2. Plain Checkpointing Algorithm

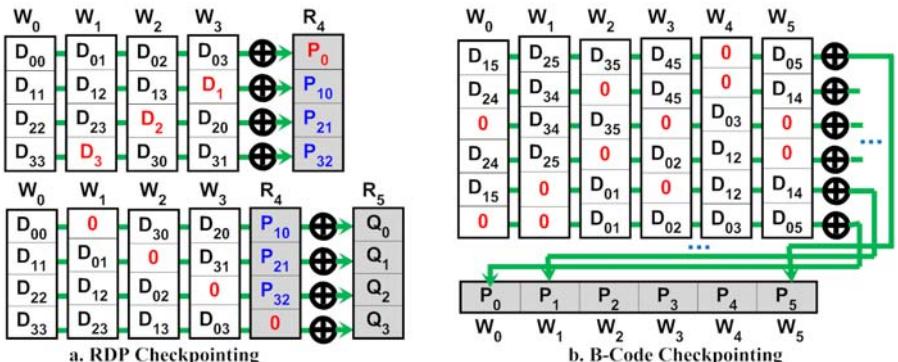


Fig. 3. Scalable Checkpointing Algorithm

Fig 3.b illustrates the scalable B-Code checkpointing algorithm. This example is based on the B-Code shown in Fig 1.b. Buffers must be preprocessed too. Every data packet (data symbol) is duplicated and the two copies are aligned with its two parities respectively. The buffer areas corresponding to missing parities are zeroed too. Unlike RDP, B-Code uses *all-to-all reduction* [19] instead of all-to-one reduction because parities should be distributed across all processes. The time complexity is $O(s+\log n)$. Per process extra memory requirement is $O(s)$.

Designing scalable recovery algorithms has an obstacle - chained decoding is inherently a serial procedure. So we must reorganize the computation to exploit its potential concurrency. The notations so-called *syndromes* are used here. \hat{P}_i and \hat{Q}_i denote the XOR sum of all surviving symbols in P_i and Q_j respectively. They equal to the sum of all lost symbols in the same parity group. Observing the double-erasure (disk0, disk1) in a 6-disk RDP coding system, \hat{Q}_0 is just D_{00} , D_{01} is decoded by XORing \hat{P}_0 and D_{00} , $D_{11}=\hat{Q}_1$ XOR D_{01} , and so on. Although each step depends on the last one, but in fact syndromes can be calculated independently. So we can calculate all syndromes by two all-to-one reduction, then the root process (one of the re-spawned process) performs decoding locally (this step is not suitable for parallelization because of high communication overhead), and finally the root process sends recovered checkpoint data to another re-spawned process. This is an $O(s\log n)$ algorithm. The space complexity is $O(s)$. Scalable B-Code recovery algorithm is similar. But only one all-to-one reduction is executed to calculate syndromes.

5 Experimental Evaluation

We implemented our algorithms in FT-MPI. For RDP, the scalable algorithms were used. For B-Code, we chose the plain algorithms. We also implemented RAID-4 and RAID-5 based checkpointing for comparison. RAID-4 checkpointing and RAID-4/5 recovery were implemented by an all-to-one reduction. RAID-5 checkpointing was implemented by an all-to-all reduction.

We implemented in-memory checkpointing at the application level. We designed a common interface for erasure code based checkpointing. The checkpointing and recovery algorithms were implemented as helper functions. When checkpointing or recovery needs to be performed, the interface function packs user checkpoint data into a buffer first, and then calls the helper function to execute actual checkpointing or recovery. This framework simplifies new codes incorporating. We can simply implement a set of helper functions for a new code.

We mainly tested the performance overhead of our XOR erasure codes based fault tolerance approach using the Preconditioned Conjugate Gradient (PCG) application. The underlying PCG parallel program with FT-MPI is from Innovative Computing Laboratory (ICL) at University of Tennessee. Namely, it is just the one used in [9]. The input instance used is BCSSTK23 [20] which is a 3134*3134 symmetric matrix with 45178 nonzero entries.

All experiments were performed on a cluster of 8 single-core 3.06 GHz Intel Celeron nodes. The nodes are connected with a Gigabit Ethernet. Each node has 512MB of memory and runs Red Hat Enterprise Linux AS release 4 (Nahant Update 2). The FT-MPI runtime library version is 1.0.1. We ran the PCG program for 100000

iterations. For PCG with checkpoint, we took checkpoint every 25 iterations. That is to say, 4000 checkpoints were taken in each run. For PCG with recovery, we simulated a single or double process failure by killing the first process or the first two processes at the 50000-th iteration. MPI_Wtime was used to measure the execution time. Each data point is the average of 3 runs.

The purpose of the first set of experiments is to measure the performance overhead of taking checkpoints. For RAID-4, 6 working processes and 1 redundant process are invoked. For RDP, 6 working processes and 2 redundant processes are used. For RAID-5 and B-Code, 6 working processes are used. Fig 4 shows the execution time of PCG w/o checkpoint, and the overhead of taking checkpoint. As expected, the single-erasure codes have lower overhead than the double-erasure codes. RAID-5 has higher overhead than RAID-4. The reason is that the checkpoint data size s is relatively small which favor all-to-one reduction than all-to-all reduction. The overhead of the plain B-Code algorithm is obviously higher than that of the scalable RDP algorithm. It seems that the checkpoint overhead of our approach is considerably worse than the result published in [9]. But please note, the checkpoint interval in our experiments is only about 30ms, while that in [9] ranges from 24s to 330s. It takes our algorithms 0.875ms~2.5ms to take a checkpoint. While the time of the single- and double-erasure algorithms presented in [9] ranges from 205ms to 500ms. After stripping out the impact of system size and input size, our bitwise, XOR-based approach is still superior to floating-point arithmetic coding approach used in [9].

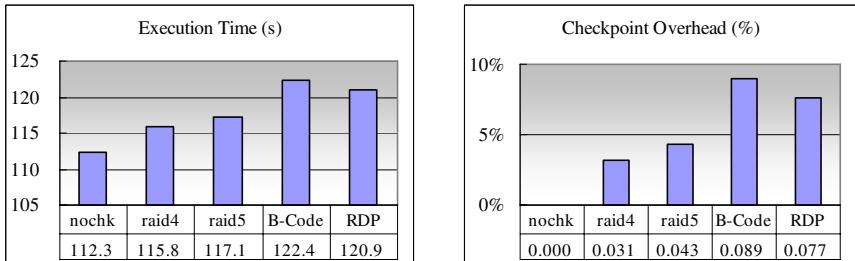


Fig. 4. PCG Checkpointing Overhead

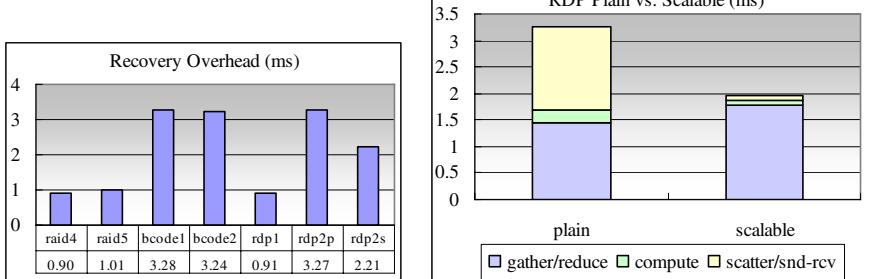


Fig. 5. PCG Recovery Overhead

Fig. 6. RDP Plain vs. Scalable

Fig 5 shows the overhead of recovery. “bcode1” and “bcode2” denote single- and double-failure recovery based on B-Code respectively. “rdp1” denotes RDP based single-failure recovery. “rdp2p” and “rdp2s” denote RDP plain and scalable double-failure recovery respectively. We can see that the three single-failure recovery algorithms have the lowest overhead, because they only perform one all-to-one reduction. Compared with the plain algorithm, the scalable RDP double-failure recovery algorithm decreases overhead remarkably. It is also superior to the plain B-Code algorithm. It is worth while to note that our recovery algorithms are almost as fast as their checkpointing buddies. However the recovery algorithms presented in [9] are much slower than their corresponding checkpointing algorithms..

Fig 6 shows the detailed comparison between the plain RDP double-failure recovery algorithm and the scalable algorithm. Obviously, although the reduction step of the scalable algorithm is slightly slower than the gather step of the plain algorithm because the former undertakes numerous XOR operations, but the next two steps of the scalable algorithm are much faster than those of the plain algorithm.

6 Conclusion and Future Work

In this paper, we made a preliminary attempt to applied XOR-based erasure codes to in-memory checkpointing for MPI programs. Our main contributions include: 1) introduces two XOR-based double-erasure codes - RDP and B-Code into fault-tolerant MPI scenario; 2) incorporates encoding/decoding computation into MPI collective communication primitives. Our fault-tolerant approach is resilient to any two node/process failures. The experiments show that the scalable algorithms decrease communication overhead and balance computation effectively. Our approach is superior to related work in checkpointing and recovery overhead.

Evaluating our approach in larger systems using more different MPI applications is the principle work in the future. Optimizing computation and communication further is another important work. A possible way is to design new collective communication operations because some encoding/decoding processes conform to *all-to-k* patterns instead of standard all-to-one or all-to-all. Using multi-erasure codes to tolerate more than two failures is obviously a valuable work. In addition, translating our approach into floating-point arithmetic style will improve its applicability. For MPI applications with local communication patterns, we plan to try erasure codes with good locality.

References

1. <http://www.top500.org>
2. Wu-Chun, F.: The Importance of Being Low Power in High Performance Computing. Cyberinfrastructure Technology Watch Quarterly 1(3), 12–21 (2005)
3. Message Passing Interface Forum: MPI: A Message Passing Interface Standard. Technical report, University of Tennessee (1994)
4. Stellner, G.: CoCheck: Checkpointing and Process Migration for MPI. In: 10th International Parallel Processing Symposium, Honolulu, USA, pp. 526–531 (1996)

5. Agbaria, A., Friedman, R.: Starfish: Fault-Tolerant Dynamic MPI Programs on Clusters of Workstations. In: 8th IEEE International Symposium on High Performance Distributed Computing, Redondo Beach, California, USA, pp. 167–176 (1999)
6. Bosilca, G., Bouteiller, A., Cappello, F., Djilali, S., Fedak, G., Germain, C., Herault, T., Lemarinier, P., Lodygensky, O., Magniette, F., Neri, V., Selikhov, A.: MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes. In: 2002 ACM/IEEE conference on Supercomputing, Baltimore, Maryland, USA, pp. 1–18 (2002)
7. Fagg, G.E., Dongarra, J.: FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. In: Dongarra, J., Kacsuk, P., Podhorszki, N. (eds.) PVM/MPI 2000. LNCS, vol. 1908, pp. 346–353. Springer, Heidelberg (2000)
8. Plank, J.S., Li, K., Puening, M.A.: Diskless checkpointing. IEEE Trans. Parallel Distrib. Syst. 9(10), 972–986 (1998)
9. Chen, Z., Fagg, G., Gabriel, E., Langou, J., Angskun, T., Bosilca, G., Dongarra, J.: Fault Tolerant High Performance Computing by a Coding Approach. In: 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Chicago, IL, USA, pp. 213–223 (2005)
10. Liu, X.G., Wang, G., Zhang, Y., Li, A., Xie, F.: The Performance Of Erasure Codes Used In FT-MPI. In: 2nd International Forum on Information Technology and Applications, Chengdu, China (2005)
11. Plank, J.S.: Erasure Codes for Storage Applications. Tutorial. In: 4th Usenix Conference on File and Storage Technologies, San Francisco, CA, USA (2005)
12. Chen, P.M., Lee, E.K., Gibson, G.A., Katz, R.H., Patterson, D.A.: RAID: High-Performance, Reliable Secondary Storage. ACM Computing Surveys 26(2), 143–185 (1994)
13. Plank, J.S.: A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-like Systems. Software - Practice & Experience 27(9), 995–1012 (1997)
14. Corbett, P., English, B., Goel, A., Greanac, T., Kleiman, S., Leong, J., Sankar, S.: Row-Diagonal Parity for Double Disk Failure Correction. In: 3rd USENIX Conference on File and Storage Technologies, San Francisco, CA, USA, pp. 1–14 (2004)
15. Blaum, M.: A Family of MDS Array Codes with Minimal Number of Encoding Operations. In: 2006 IEEE International Symposium on Information Theory, Washington, USA, pp. 2784–2788 (2006)
16. Xu, L., Bohossian, V., Bruck, J., Wagner, D.G.: Low-Density MDS Codes and Factors of Complete Graphs. IEEE Trans. on Information Theory 45(6), 1817–1826 (1999)
17. Colbourn, C.J., Dinitz, J.H., et al.: Handbook of Combinatorial Designs, 2nd edn. CRC Press, Boca Raton (2007)
18. Plank, J.S.: The RAID-6 Liberation Codes. In: 6th USENIX Conference on File and Storage Technologies, San Francisco, USA, pp. 97–110 (2008)
19. Grama, A., Gupta, A., Karypis, G., Kumar, V.: Introduction to Parallel Computing, 2nd edn. Addison Wesley, Edinburgh Gate (2003)
20. <http://math.nist.gov/MatrixMarket/data/Harwell-Boeing/bcsstruc3/bcsstk23.html>

MPC-MPI: An MPI Implementation Reducing the Overall Memory Consumption

Marc Pérache, Patrick Carribault, and Hervé Jourdren

CEA, DAM, DIF F-91297 Arpajon, France

{marc.perache,patrick.carribault,herve.jourdren}@cea.fr

Abstract. Message-Passing Interface (MPI) has become a standard for parallel applications in high-performance computing. Within a single cluster node, MPI implementations benefit from the shared memory to speed-up intra-node communications while the underlying network protocol is exploited to communicate between nodes. However, it requires the allocation of additional buffers leading to a memory-consumption overhead. This may become an issue on future clusters with reduced memory amount per core. In this article, we propose an MPI implementation built upon the MPC framework called MPC-MPI reducing the overall memory footprint. We obtained up to 47% of memory gain on benchmarks and a real-world application.

Keywords: Message passing, Memory consumption, High-performance computing, Multithreading.

1 Introduction

Message-Passing Interface [1] (MPI) has become a major standard API for SPMD¹ programming model in the high-performance computing (HPC) area. It provides a large set of functions abstracting the communications between several *tasks*, would these tasks be on the same address space or on different nodes of a cluster. Within a shared-memory environment, MPI runtimes benefit from the global memory available by using buffers stored inside a memory-segment shared among the tasks (or processes). The size of these buffers is mainly implementation-dependent but it may have a huge impact on the final performance of the running application.

The amount of memory per core tends to decrease in recent high-performance computers. Currently, the Tera-10 machine² provides 3GB of memory per core with a total of about 10,000 cores. With its 1.6 million cores, the future Sequoia machine will only have on average 1GB of memory available per core. One explanation of this trend could be that the number of cores is still increasing, thanks to Moore's law, while the memory space cannot grow the same speed. The

¹ Single Program Multiple Data.

² As of November 2008, Tera-10 is ranked 48th in the TOP500 list <http://www.top500.org/> and was 5th when it was installed, in June 2006.

memory per core is therefore becoming a bottleneck for efficient scalability of future high-performance applications. MPI libraries have to adapt their memory consumption to guarantee a correct overall scalability on future supercomputers.

In this paper, we introduce MPC-MPI, an extension to the MPC framework [2] (MultiProcessor Communications). MPC-MPI provides a full MPI-compatible API with a lower memory footprint compared to state-of-the-art MPI runtimes. The MPC framework implements an $M \times N$ thread library including a NUMA-aware and thread-aware memory allocator, and a scheduler optimized for communications. MPC-MPI is developed upon the MPC framework by providing a compatibility with the MPI API version 1.3 [1] where each task is a single thread handled by the user-level scheduler. Thanks to this implementation, we reduced the whole memory consumption on benchmarks and real-world applications by up to 47%.

This article is organized as follows: Section 2 details the related work and Section 3 summarizes the current features of the MPC framework. Then, Section 4 explains how we extended MPC with MPC-MPI. Memory consumption optimizations are detailed in Section 5. Section 6 presents experiments comparing MPC-MPI to state-of-the-art MPI implementations before concluding in Section 7.

2 Related Work

A significant part of the memory consumption of MPI libraries is due to additional buffers required to perform efficient communications. This section describes standard approaches to handle the issue.

2.1 Standard MPI Approaches

According to the MPI standard, MPI tasks are processes. In MPI runtimes, the message-passing operations rely on copy methods from the source buffer to the destination one. There are four main approaches to accomplish this copy.

Two-copy method: The 2-copy method relies on the allocation of a shared-memory segment between the communicating tasks. First of all, this approach requires a copy from the original buffer to the shared-memory segment. Then, an additional copy is performed from this segment to the destination buffer. If a message is larger than the memory buffer, it has to be split. The size and the number of these segments is a tradeoff between performance and memory consumption. The best-performing variant allocates a shared buffer for each pair of processes leading to a total of N^2 buffers where N is the number of tasks. Even though this approach is not scalable in memory, it is used in MPICH Nemesis fastbox [3] on nodes with low amount of cores. A more-suitable approach for scalability involves only one buffer per task, as MPICH Nemesis uses for large nodes.

Two-copy with Rendez-vous method: The rendez-vous method is a variant of the 2-copy method using a few larger additional buffers to realize the intermediate copy. This method improves performance for large messages, avoiding message splitting; but it requires a synchronization of involved tasks to choose a free buffer. This method increases the memory consumption.

One-copy method: The 1-copy method is designed for large-message communications when the MPI library is able to perform a direct copy from the source buffer to the destination. This can be done through a kernel module performing physical-address memory copy. This method is efficient in terms of bandwidth but it requires a system call that increases the communication latency. The main advantage of this method is that it does not require additional buffers. This method is used in MPIBull.

Inter-node communications: Network Interface Controller (NIC) libraries require per process data structure allocations. The size of these data is often related to the number of neighbors of each process. Thus, the memory consumption is N^2 where N is the number of processes. Whereas some processes may share a node, NIC libraries are not able to share these data due to the process model that requires the separation of the memory address spaces. Nevertheless, some solutions have been proposed to reduce the memory footprint related to the NIC and the NIC library capabilities [4,5].

2.2 Process Virtualization

Unlike standard MPI approaches that map tasks to processes, *process virtualization* [6,7,8] benefits from shared-memory available on each cluster node by dissociating tasks from processes: tasks are mapped to threads. It allows efficient 1-copy method or straight-forward optimizations of inter-node communications. Nevertheless, process virtualization requires to restrict the use of global variables because many processes now share a unique address space.

3 The MPC Framework

MPC-MPI is an extension of the MultiProcessor Communications (MPC) framework [2]. This environment provides a unified parallel runtime built to improve the scalability and performance of applications running on clusters of large multiprocessor/multicore NUMA nodes. MPC uses process virtualization as its execution model and provides dedicated message-passing functions to enable task communications. The MPC framework is divided into 3 parts: the thread library, the memory allocator and the communication layer.

3.1 Thread Library

MPC comes with its own MxN thread library [9,10]. MxN thread libraries provide lightweight user-level threads that are mapped to kernel threads thanks to a user-level thread scheduler. One key advantage of the MxN approach is the ability to

optimize the user-level thread scheduler and thus to create and schedule a *very* large number of threads with a reduced overhead. The MPC thread scheduler provides a polling method that avoids busy-waiting and keeps a high level of reactivity for communications, even when the number of tasks is much larger than the number of available cores. Furthermore, collective communications are integrated into the thread scheduler to enable efficient *barrier*, *reduction* and *broadcast* operations.

3.2 Memory Allocation

The MPC thread library is linked to a dedicated *NUMA-aware* and *thread-aware* memory allocator. It implements a *per-thread heap* [11,12] to avoid contention during allocation and to maintain data locality on NUMA nodes. Each new data allocation is first performed by a lock-free algorithm on the thread private heap. If this local private heap is unable to provide a new memory block, the requesting thread queries a *large page* to the *second-level global heap* with a synchronization scheme. A *large page* is a parametrized number of system pages. Memory deallocation is locally performed in each private heap. When a large page is totally free, it is returned to the *second-level global heap* with a lock-free method. Pages in *second-level global heap* are virtual and are not necessarily backed by physical pages.

3.3 Communications

Finally, MPC provides basic mechanisms for intra-node and inter-node communications with a specific API. Intra-node communications involve two tasks in a same process (MPC uses one process per node). These tasks use the optimized thread-scheduler polling method and thread-scheduler integrated collectives to communicate with each other. As far as inter-node communications are concerned, MPC used up to now an underlying MPI library. This method allows portability and efficiency but hampers aggressive communication optimizations.

4 MPC-MPI Extension

This paper introduces MPC-MPI, an extension to the MPC framework that provides a whole MPI interface for both intra-node (tasks sharing a same address space) and inter-node communications (tasks located on different nodes). This section deals with the implementation choices of MPC-MPI.

4.1 MPI Tasks and Intra-node Communications

The MPC framework already provides basic blocks to handle message passing between tasks. Therefore, MPC-MPI maps MPI tasks to user-level threads instead of processes. Point-to-point communications have been optimized to reduce the memory footprint (see Section 5) whereas collective communications

already reached good performance and low memory consumption. MPC-MPI currently exposes an almost-full compatibility with MPI 1.3 [1]: it passes 82% of the MPICH test suite.³ The missing features are inter-communicators and the ability to cancel a message.

Nevertheless, compatibility issues remain: MPC-MPI is not 100% MPI compliant due to the process virtualization. Indeed, the user application has to be *thread-safe* because several MPI tasks may be associated to the same process and will share global variables. To ease the migration from standard MPI source code to MPC-MPI application, we modified the GCC compiler to warn the programmer for each global-scope variable (privatizing or removing such variables is one way to guarantee the thread safety of a program).

4.2 Inter-node Communications

The MPC framework relies on an external MPI library to deal with inter-node message exchange. MPC-MPI now implements its own *inter-node communication layer* based on direct access to the NICs as far as high-performance networks are concerned. This new layer enables optimizations according to communication patterns. For example, it reduces the overhead of adding the MPC-MPI message headers required for communications. So far, MPC-MPI supports three network protocols: TCP, Elan(Quadrics) and InfiniBand but there are no restrictions to extend this part to other networks.

5 Memory Consumption Optimization

MPC-MPI reduces the memory consumption of MPI applications thanks to process virtualization, intra-node/inter-node communications and memory-management optimizations. This section depicts these optimizations.

5.1 Intra-node Optimizations

The optimizations of memory requirements for intra-node communications mainly rely on the removal of temporary buffers. The unified address space among tasks within a node allows MPC-MPI to use the 1-copy buffer-free method without requiring any system call. That is why the latency of MPC-MPI communications remains low even for small messages. Nevertheless, for optimization purposes, tiny messages (messages smaller than 128B) may use a 2-copy method on a 2KB per-task buffer. This optimization reduces the latency for tiny messages. Finally, the 1-copy method has been extended to handle non-contiguous derived datatypes without additional packing and unpacking operations.

5.2 Inter-node Optimizations

The second category of memory optimizations are related to the inter-node message exchange. With MPC-MPI, a process handles several tasks and only one

³ http://www.mcs.anl.gov/research/projects/mpi/mpi_tests/tsuite.html.

process is usually spawned on each node. Thus, network buffers and structures are shared among all the tasks of the process. The underlying NIC library only sees one process and, therefore one identifier for communication between nodes. This enables further optimizations at the NIC level e.g., packet aggregation.

5.3 Memory Allocation Optimizations

Finally, the MPC memory allocator has been optimized to recycle memory chunks between tasks. Many high-performance applications allocate temporary objects that have very short lifetimes. With a standard memory allocator, these allocations are buffered to avoid a system call each time such event occurs. MPC-MPI uses smaller buffers in each thread-private heap but it also maintains a shared pool of memory pages in the global heap to avoid system calls. It enables intra-node page sharing between tasks without the system calls required in the multiprocess approach.

6 Experiments

This section describes some experimental results of MPC-MPI, conducted on two architectures: a dual-socket Intel Nehalem-EP (2×4 cores) with 12GB of main memory running Red Hat Enterprise Linux Client release 5.2 and an octo-socket Intel Itanium Montecito (8×2 cores) with 48GB of memory running Bull Linux AS4 V5.1 (one node of the CEA/DAM Tera-10 machine). To evaluate the MPI performance of MPC-MPI, the Intel MPI Benchmarks⁴ have been used with the option `-off_cache` to avoid cache re-usage and, therefore, to provide realistic throughput. The tests compare MPC-MPI to MPICH 2 1.1a2 Nemesis on the Nehalem architecture and MPIBull 1.6.5 on the Itanium Tera-10 machine.⁵ These experiments are divided into 3 categories: (i) point-to-point communication benchmarks, (ii) collective communication benchmarks and (iii) results on our real-world application.

6.1 Point-to-Point Benchmarks

Figure 1 provides the latency and the bandwidth of the MPC-MPI library on the standard pingpong benchmark, with two tasks communicating among a total of 2, 8 or 16 tasks. MPC-MPI is a bit slower than MPICH Nemesis on the Nehalem architecture when the number of MPI tasks is low. But with a total of 8 tasks or with 16 hyperthreaded tasks per node, MPC-MPI keeps a constant latency and bandwidth whereas MPICH Nemesis slows down. As far as the Itanium Tera-10 machine is concerned, MPC-MPI is a bit slower than MPIBull. This gap is mainly due to a lack of Itanium-specific optimizations in MPC-MPI.

Figure 2 depicts the memory consumption of MPC-MPI compared to other implementations. The optimizations described in this paper lead to memory gain

⁴ <http://www.intel.com/cd/software/products/asmo-na/eng/219848.htm>.

⁵ MPIBull is the vendor MPI available on the Tera-10 machine.

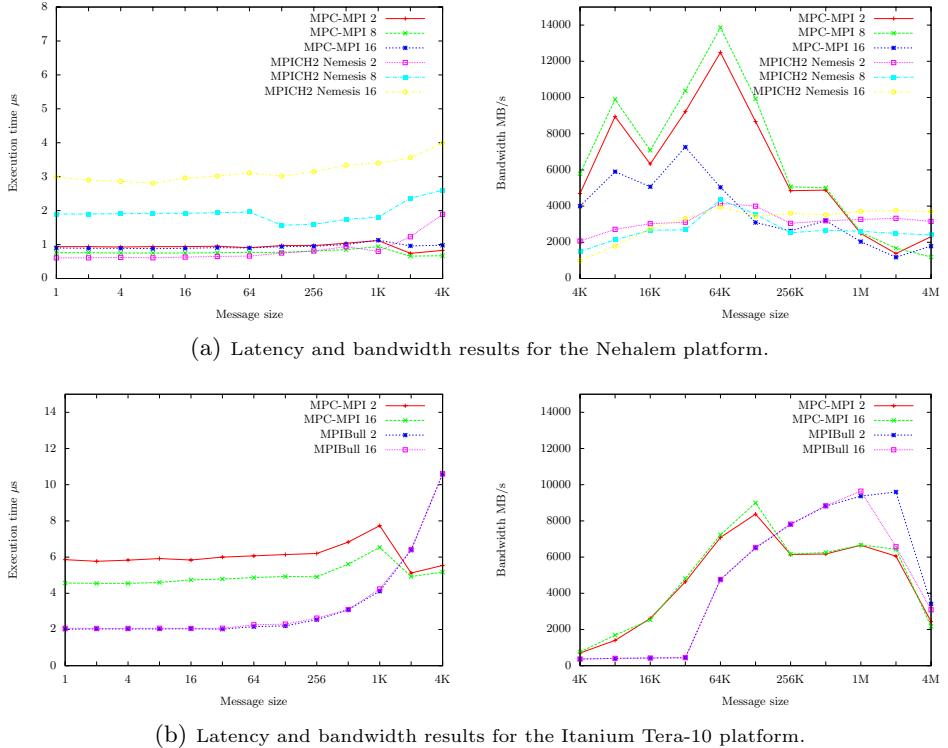


Fig. 1. Performance of the point-to-point communication benchmark

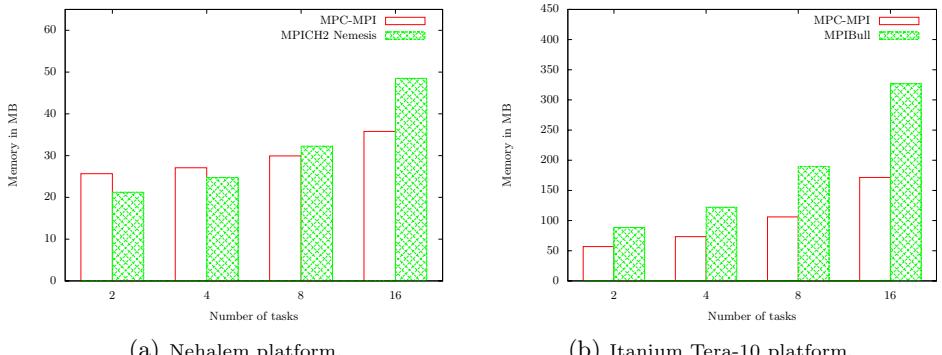


Fig. 2. Memory consumption of the point-to-point communication benchmark

when the node is fully used, even on a simple test such as a pingpong. MPC-MPI allows to reduce the overall memory consumption by up to 26% on the Nehalem compared to MPICH and up to 47% on the Itanium compared to MPIBull.

This pingpong benchmark illustrates the rather good performance of MPC-MPI compared to other MPI libraries in terms of latency and bandwidth. It also

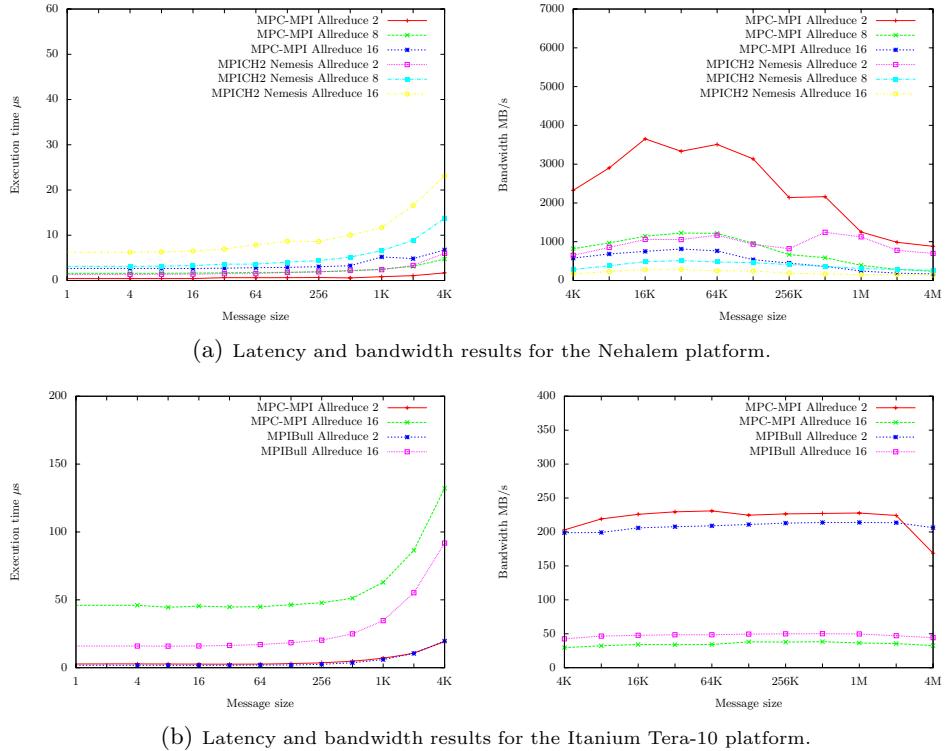


Fig. 3. Performance of the allreduce communication benchmark

illustrates the benefits of MPC-MPI memory optimizations that lead to save a significant amount of memory, even on this simple test.

6.2 Collective Benchmarks

The second benchmark category is related to collective communications. Figure 3 shows the latency and bandwidth of the MPC-MPI library on the Allreduce benchmark. MPC-MPI is a bit faster than MPICH Nemesis on the Nehalem architecture whatever the number of tasks is. On the Itanium Tera-10 machine, MPC-MPI is still a bit slower than MPIBull.

Figure 4 illustrates the memory consumption of MPC-MPI compared to other implementations. The results with the Allreduce benchmark are similar to the pingpong ones. MPC-MPI saves up to 31% of the overall memory on the Nehalem compared to MPICH and up to 32% compared to MPIBull on Itanium.

6.3 Application

MPC-MPI has been used with the HERA application on the Tera-10 supercomputer. HERA is a large multi-physics 2D/3D AMR hydrocode platform [13].

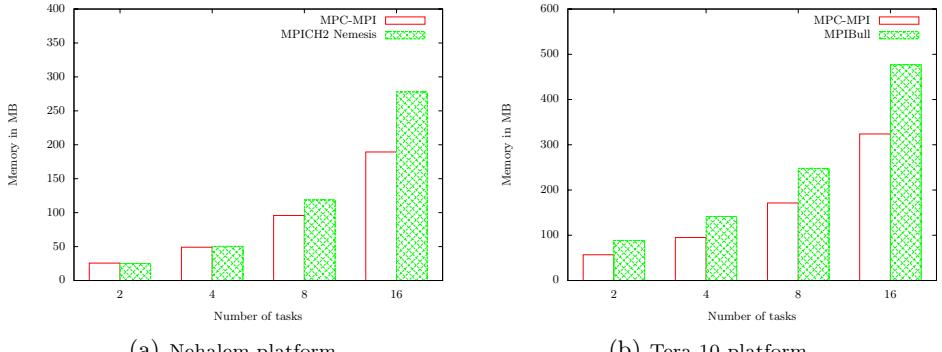


Fig. 4. Memory consumption of the allreduce communication benchmark

On a typical ablation front fluid-flow problem, with 3-temperature multifluid hydrodynamics (explicit Godunov-type CFD solver) coupled to 3-temperature diffusion (iterative Newton-Krylov solver + preconditioned conjugate-gradient method), MPC-MPI allows a 47% reduction of memory consumption (1,495MB for MPC-MPI instead of 2,818MB for MPIBull) on a 16-core Tera-10 NUMA node, for just 8% overhead compared to native MPIBull (about 400,000 cells, with aggressive Adaptive Mesh Refinement and dynamic load balancing). This large C++ code uses number of derived types and collective communications during its execution, demonstrating the robustness of the MPC-MPI library.

7 Conclusion and Future Work

In this paper we presented MPC-MPI, an extension of the MPC framework providing an MPI-compatible API. This extension reduces the overall memory consumption of MPI applications, thanks to process virtualization and a large number of optimizations concerning communications (intra-node and inter-node) and memory management. Experiments on standard benchmarks and a large application show significant memory gains (up to 47%) with only a small performance slowdown compared to vendor MPI (MPIBull on Itanium Tera-10) and a state-of-the-art MPI implementation (MPICH Nemesis on Nehalem). Furthermore, optimizations of inter-node communications do not hamper further optimizations proposed by specialized high-performance communication NIC libraries on dedicated hardware.

There are still missing features for a full MPI 1.3 API, features to be implemented in the near future. Some work has also to be done concerning thread-safety required by MPC-MPI. So far, a modified GCC compiler is provided to highlight global variables of the application, but no tool is available to transform the application code.

The MPC-MPI extension of MPC⁶ is the first step to provide an optimized hybrid MPI/thread runtime system on Petaflops machines, addressing more specifically the *stringent* memory constraints appearing on such computer architectures.

References

1. Message Passing Interface Forum: MPI: A message passing interface standard (1994)
2. Péache, M., Jourdren, H., Namyst, R.: MPC: A unified parallel runtime for clusters of NUMA machines. In: Luque, E., Margalef, T., Benítez, D. (eds.) Euro-Par 2008. LNCS, vol. 5168, pp. 78–88. Springer, Heidelberg (2008)
3. Buntinas, D., Mercier, G., Gropp, W.: Design and evaluation of nemesis, a scalable, low-latency, message-passing communication subsystem. In: CCGRID 2006: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid (2006)
4. Sur, S., Koop, M.J., Panda, D.K.: High-performance and scalable MPI over InfiniBand with reduced memory usage: an in-depth performance analysis. In: SC 2006: Proceedings of the 2006 ACM/IEEE conference on Supercomputing (2006)
5. Koop, M.J., Jones, T., Panda, D.K.: Reducing connection memory requirements of MPI for InfiniBand clusters: A message coalescing approach. In: CCGRID 2007: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid (2007)
6. Kalé, L.: The virtualization model of parallel programming: runtime optimizations and the state of art. In: LACSI (2002)
7. Huang, C., Lawlor, O., Kalé, L.V.: Adaptive MPI. In: Rauchwerger, L. (ed.) LCPC 2003. LNCS, vol. 2958, pp. 306–322. Springer, Heidelberg (2004)
8. Tang, H., Yang, T.: Optimizing threaded MPI execution on SMP clusters. In: ICS 2001: Proceedings of the 15th International Conference on Supercomputing (2001)
9. Namyst, R., Méhaut, J.F.: PM2: Parallel multithreaded machine. a computing environment for distributed architectures. In: Parallel Computing, ParCo 1995 (1995)
10. Abt, B., Desai, S., Howell, D., Perez-Gonzalez, I., McCracken, D.: Next Generation POSIX Threading Project (2002)
11. Berger, E., McKinley, K., Blumofe, R., Wilson, P.: Hoard: a scalable memory allocator for multithreaded applications. In: International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX) (2000)
12. Berger, E., Zorn, B., McKinley, K.: Composing high-performance memory allocators. In: Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation (2001)
13. Jourdren, H.: HERA: A hydrodynamic AMR platform for multi-physics simulations. In: Adaptive Mesh Refinement - Theory and Application, LNCSE (2005)

⁶ MPC, including the MPC-MPI extension, is available at <http://mpc.sourceforge.net>.

Towards an Efficient Process Placement Policy for MPI Applications in Multicore Environments

Guillaume Mercier and Jérôme Clet-Ortega

Université de Bordeaux - INRIA - LaBRI
351, cours de la Libération F-33405 Talence cedex
`{guillaume.mercier,jerome.clet-ortega}@labri.fr`

Abstract. This paper presents a method to efficiently place MPI processes on multicore machines. Since MPI implementations often feature efficient supports for both shared-memory and network communication, an adequate placement policy is a crucial step to improve applications performance. As a case study, we show the results obtained for several NAS computing kernels and explain how the policy influences overall performance. In particular, we found out that a policy merely increasing the intranode communication ratio is not enough and that cache utilization is also an influential factor. A more sophisticated policy (eg. one taking into account the architecture's memory structure) is required to observe performance improvements.

Keywords: Message-Passing, Multicore architectures, Process placement.

1 Introduction

In the last decade, parallel computer architectures have evolved dramatically. Clusters have shifted from an assembly of uniprocessor machines interconnected by a single network to a complex and highly hierarchical structure. Nodes are now composed of several multicore processors sharing memory banks physically scattered across the node. Major CPU manufacturers like AMD (HyperTransport) and Intel (Quick Path Interconnect) follow this trend. The memory access time depends on the location of both the core and the memory bank. This is often designed as the Non-Uniform Memory Access (NUMA) effect. The memory hierarchy is also more complex due to the increase of cache levels. This sharing of memory resources depends on the CPU architecture and differs from a manufacturer to another. As for the interconnection network, multirail systems where several high-speed NICs are connected to a node are sometimes also encountered.

A real challenge for a parallel applications is to exploit such architectures at their full potential. In order to achieve the best performance, many factors must be taken into consideration and studied. The first one is to make use of an implementation of the MPI specifications [1] able to efficiently take advantage of a multicore environment. Whilst the MPI standard is architecture-independent, it is an implementation's task to bridge the gap between the hardware's performance and the application's. Indeed, recent MPI-2 implementations such as

Open MPI [2] or MPICH2 [3] fulfill this purpose and offer a very satisfactory performance level on multicore architectures.

However, in order for an MPI implementation to fully exploit the underlying hardware, the MPI application processes have to be placed carefully on the cores of the target architecture. This placement policy has to be defined by both the application's communication pattern and the hardware's characteristics. For instance, if some application processes communicate more frequently than others, they should be regrouped and placed on the same multicore node. By doing so, the amount of intranode communication increases and the application global performance will improve since intranode communication (shared memory) is faster than internode communication (network).

In this paper we expose the method and software tools we employed to allow an MPI application to better take advantage of a multicore environment. We will show a performance improvement not due to modifications of the MPI implementation itself but rather due to a relevant process placement. The rest of this paper is organized as follows: Section 2 describes how process placement is determined and on which set of tools and algorithms it relies. Experimental results are presented in Section 3 with performance figures for some of the NAS computing kernels. Section 4 lists related works in this area of MPI process placement and discusses some issues raised by information gathering. Section 5 concludes this paper and opens future perspectives.

2 Computation of a Relevant MPI Process Placement

In order to place the MPI processes in a relevant fashion, we have to gather information about the target architecture and the application's communication pattern. Once both are available we analyze them and determine the best possible placement. The criterion choice should follow a user-defined strategy. However, the current scope of this work does not encompass all MPI applications.

2.1 Hypotheses about MPI Applications and Their Execution Environment

In the rest of this paper we consider *static* MPI applications. By static, we mean that the application does not use any of the dynamic processes features offered by MPI-2. We also exclude hybrid MPI applications that rely on multithreading features (such as OpenMP directives). The number of computing entities (threads and processes) is therefore guaranteed to remain constant during an application's execution. We also consider the target machine to be fully dedicated to the MPI application. All cores are usable by the MPI processes with the restriction that only a single MPI process runs on a given core. As a consequence of these points, a static mapping between the MPI processes and the CPU cores can be computed before launching the application. This placement will not need to be modified during the application's execution.

2.2 Gathering the Hardware's Information

As previously explained, clusters of NUMA nodes are hierarchically structured. For instance, figure 1 shows the architecture of an AMD Opteron-based compute node. This compute node is composed of four dies with two cores each. Each die possesses a set of main memory banks attached to it. A core features its own Level 1 cache (not shown) and Level 2 cache, not shared with the other core on the same die. A core located on a die can access the main memory of any other die but the access time increases as the physical distance between the core and the memory bank lengthens. A network interface card (NIC) can be attached to a bus connected to Die #0 and Die #1. Since the cores located on these dies are physically closer to the I/O bus, one might expect faster network transactions for processes mapped on these cores.

Some tools can provide us with the needed information (e.g libtopology or Portable Linux Processor Affinity [4] available for Linux). However, such tools are not portable and accurate enough over a wide spectrum of operating systems.

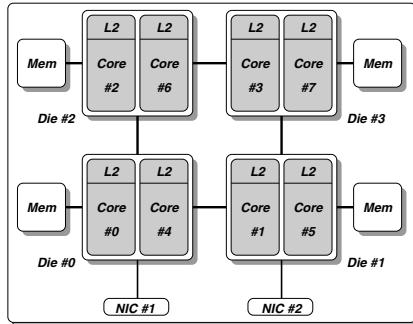


Fig. 1. An exemple of hierarchical compute node: an 8-cores Opteron

Machine:
NUMANode + Die: Node#0(8GB) Die#0
L2Cache + Core + L1Cache + SMTproc : L2#0(1MB) Core#0 L1#0(64kB) CPU#0
L2Cache + Core + L1Cache + SMTproc : L2#4(1MB) Core#1 L1#4(64kB) CPU#4
NUMANode + Die: Node#1(8GB) Die#1
L2Cache + Core + L1Cache + SMTproc : L2#1(1MB) Core#0 L1#1(64kB) CPU#1
L2Cache + Core + L1Cache + SMTproc : L2#5(1MB) Core#1 L1#5(64kB) CPU#5
NUMANode + Die: Node#2(8GB) Die#2
L2Cache + Core + L1Cache + SMTproc : L2#2(1MB) Core#0 L1#2(64kB) CPU#2
L2Cache + Core + L1Cache + SMTproc : L2#6(1MB) Core#1 L1#6(64kB) CPU#6
NUMANode + Die: Node#3(8GB) Die#3
L2Cache + Core + L1Cache + SMTproc : L2#3(1MB) Core#0 L1#3(64kB) CPU#3
L2Cache + Core + L1Cache + SMTproc : L2#7(1MB) Core#1 L1#7(64kB) CPU#7

Fig. 2. Hardware information generated by PM²'s topology discovery mechanism

$$Machine = \begin{pmatrix} 0 & 100 & 100 & 10 & 1000 & 100 & 100 & 10 \\ 100 & 0 & 10 & 100 & 100 & 1000 & 10 & 100 \\ 100 & 10 & 0 & 100 & 100 & 10 & 1000 & 100 \\ 10 & 100 & 100 & 0 & 10 & 100 & 100 & 1000 \\ 1000 & 100 & 100 & 10 & 0 & 100 & 100 & 10 \\ 100 & 1000 & 10 & 100 & 100 & 0 & 10 & 100 \\ 100 & 10 & 1000 & 100 & 100 & 10 & 0 & 100 \\ 10 & 100 & 100 & 1000 & 10 & 100 & 100 & 0 \end{pmatrix}$$

Fig. 3. Matrix representation of the hardware's information from figure 2. ($Machine(i,j)$ represents the potential of communication between cores #i and #j).

To gather the hardware's information, we used a topology discovery mechanism implemented in the PM² runtime system [5]. This feature is employed to better schedule threads in multicore environments [6]. Currently, this mechanism only deals with CPUs and does not deliver information about networks and I/O buses. For instance, when applied to the Opteron node as depicted by figure 1, the PM² topology discovery mechanism outputs the information displayed by figure 2.

We generate from this output a data structure that other tools will use to compute the placement. This data structure is a graph with vertices representing the CPU cores and weighted edges. This graph is complete and non-oriented. The weight affected to each edge increases as more elements of the memory hierarchy are shared between cores. It is also affected by the NUMA effect.

Figure 3 shows the various weight values chosen for the Opteron compute node (figure 1). We decided to put the largest weight for cores on the same die because they directly share memory banks. The next largest weight value corresponds to NUMA effects: for instance core #0 is able to access the memory attached to Die #2 and Die #1 faster than the memory attached to Die #3. When the machine is made of several compute nodes, we build a larger graph and affect a larger weight for cores located within the same compute node. The smallest weight value thus corresponds to communication using the network.

2.3 Collecting the Application's Communication Pattern Data

The second piece of information deals with the application's communication pattern. Each application possesses its own pattern influenced by the number of participating processes. Our chosen characterization criterion for this pattern is the amount of MPI data exchanged between processes. Therefore, we need to compute this amount for each pair of processes in the application.

Several sophisticated tools are provided for MPI application tracing and analysis, such as the MPI Parallel Environment (MPE). However, they do not provide all the necessary information. For instance, MPE is able to trace all calls to MPI routines by the application. By analyzing such a trace and focusing on the point-to-point calls, we can have hints about the communication pattern. This approach is simple and requires to configure the MPI implementation with MPE support and to link the application with the appropriate libraries. This is

$$Application = \begin{pmatrix} 0 & 1000 & 10 & 1 & 100 & 1 & 1 & 1 \\ 1000 & 0 & 1000 & 1 & 1 & 100 & 1 & 1 \\ 10 & 1000 & 0 & 1000 & 1 & 1 & 100 & 1 \\ 1 & 1 & 1000 & 0 & 1 & 1 & 1 & 100 \\ 100 & 1 & 1 & 1 & 0 & 1000 & 10 & 1 \\ 1 & 100 & 1 & 1 & 1000 & 0 & 1000 & 1 \\ 1 & 1 & 100 & 1 & 10 & 1000 & 0 & 1000 \\ 1 & 1 & 1 & 100 & 1 & 1 & 1000 & 0 \end{pmatrix}$$

Fig. 4. Matrix representation of NAS LU (class B, 8 processes) communication pattern. ($Application(i,j)$ represents the amount of communication between processes #i and #j)

limited by two factors: first, the trace generated can be potentially very large and second, the amount of data exchanged in collective operations is not taken into account.

As a consequence we found simpler and more accurate to modify an MPI implementation to collect the desired information. By modifying directly an implementation, we reduce drastically the trace size and take into account collective communication operations. In order to get a generic (that is, not implementation-specific) information, we trace only the size of MPI user data exchanged between processes. All costs induced by the implementation's internal protocols are not counted. As for the hardware's data, we represent this communication pattern with a complete, non-oriented graph with weighted edges. In this case, the weight value increases as the amount of data exchanged between MPI processes grows. An example is depicted by figure 4: the matrix represents the communication pattern for one NAS benchmark (lu.B.8).

2.4 Mapping a MPI Process Rank to a CPU Core Number

The final step is to extract an embedding of the application's graph from the target machine's graph. We use the *Scotch* software [7] to solve this *NP* graph problem. Scotch applies graph theory, with a divide and conquer approach, to scientific computing problems such as graph and mesh partitioning, static mapping, and sparse matrix ordering. In our case, we use the ability to construct a static mapping. Scotch implements dual recursive bipartitioning algorithms to perform this task [8] and computes static mappings for graphs larger than 2^{32} vertices. This ensures that we can create a mapping for all MPI applications, regardless of their size.

For instance, table 1 gives the static mapping computed by Scotch in the case of the NAS lu.B.8 benchmark (figure 4) launched on the Opteron compute node (figure 3). Using this static mapping, we finally generate a specific command line

Table 1. Resulting mapping for application NAS lu.B.8 on the Opteron compute node

MPI_COMM_WORLD Rank	0	1	2	3	4	5	6	7
Core Number	3	7	4	0	6	2	5	1

fully customized for each (*Application, Target Machine*) couple. Practically, each MPI process is affected to its dedicated core with the `numactl` command.

3 Evaluation: A Case Study with NAS Computing Kernels

In this section, we present some results obtained by applying the method previously described on several NAS computing kernels. We first describe our experimental environment, then show the results and finally comment on them.

3.1 Experimental Environment and NAS Benchmarks Choice

We carried out experiments on a 10-nodes cluster called *Borderline*, part of the Grid5000 testbed [9]. *Borderline* nodes are similar to the exemple depicted by figure 1: each node features four dies (a 2.6 GHz AMD Opteron 2218) with two cores each. A core possesses its own Level 2 cache (1 MBytes) not shared with the other core on the same die. The total amount of memory is 32 GBytes per node (8 GBytes per die). A Myrinet 10G NIC is attached on a bus to Die #0.

We benchmark some of the NAS computing kernels in order to assess the relevance of our placement method and policy. We first make a run of all NAS benchmarks to gather their respective communication pattern data, as described in section 2.3. Since our placement technique is currently based only on the global amount of data exchanged, we only run tests where this amount is significant: 64-processes jobs of classes C and D. Likewise, we test only applications with irregular communication patterns. Indeed, they are likely to be the most influenced by the placement of processes in a multicore environment. Among all NAS kernels, only BT, CG, LU, MG and SP have an irregular communication pattern. In the case of CG, some processes do not communicate with others. But when communication occurs, the global amount of data exchanged for each pair of processes is of same magnitude. In the case of BT, LU, MG and SP these amounts differ dramatically from one pair to the other.

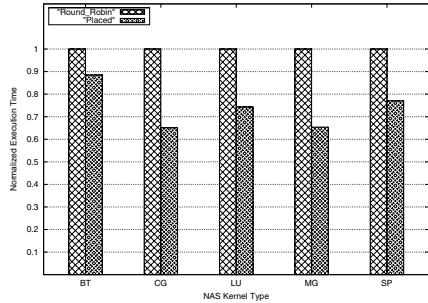
3.2 NAS Performance Results for Two MPI Implementations

In order to confirm that the method used and the data gathered are implementation-independent, we expose performance comparisons for two different MPI implementations. This first one is MPICH2-Nemesis, which relies on a very efficient intranode communication system using shared-memory [10]. We configured Nemesis to use its most recent MX support (available in the MPICH2 1.1 release). The other MPI implementation is MPICH2-MX [11] designed and implemented by Myricom. This implementation also features an efficient intranode communication system based on an in-kernel mechanism.

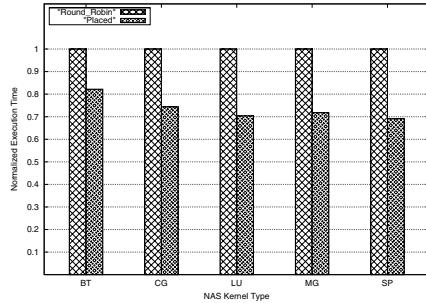
Table 2, figures 5 and 6 show the NAS performance for two different placement policies. The first placement policy is a simple *Round-Robin*-type policy where the process number i is executed on the node number n where $n \equiv i \pmod{8}$ (in our case each node features eight cores). This type of placement is commonly used when launching MPI applications without any knowledge of their communication

Table 2. MPICH2-Nemesis:MX execution times in seconds for two different placement policies

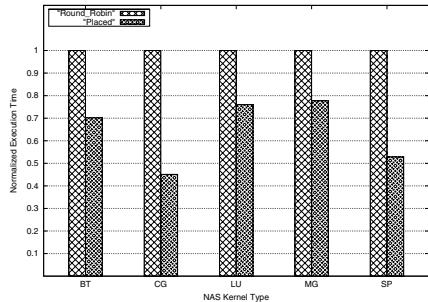
	BT		CG		LU		MG		SP	
	C	D	C	D	C	D	C	D	C	D
Round-Robin	51.6	1038.6	23.9	1177.5	45.5	1356.5	5.6	119.2	78.6	2015.63
Placed	45.6	851.7	15.6	848.4	33.6	938.3	3.7	85.3	60.2	1386.8



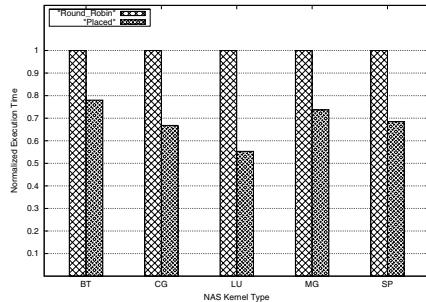
(a) Class C - 64 Processes



(b) Class D - 64 Processes

Fig. 5. Process placements comparison for MPICH2-Nemesis:MX

(a) Class C - 64 Processes



(b) Class D - 64 Processes

Fig. 6. Process placements comparison for MPICH2-MX

pattern. With this scheme, the operating system chooses on which core a process is executed. The other policy, called *Placed* is the one determined thanks to the method described in section 2. In the case of MPICH2-Nemesis:MX, there is a significant performance gap between the *Placed* policy and the *Round-Robin* policy. This result applies to both C and D classes. As our results indicate, this improvement for Classes C and D kernels is roughly of 25%. In the case of Myricom's MPICH2-MX, this gap is even larger: execution times are 34% faster when our *Placed* policy is enforced. One might note that in the case of the CG kernel (class C), performance is much better with our custom placement than with the *Round-Robin* one.

3.3 NAS Performance and Placement Policies Analysis

The *Placed* policy regroups MPI processes on the same node as much as possible. The ratio of intranode communication (using shared-memory) versus internode communication (using the Myrinet network) increases as Table 3 shows. However, questions remain: is this performance gap solely due to this increase of intranode communication ratio in the application? Do others factors – such as memory hierarchy and structure – influence performance too? In order to answer these questions, we run the NAS kernels with three other placement policies. The first new policy is the same as the *Round Robin* policy, except that each process is bound to a particular core with the `numactl` command. In this way, the operating system cannot change a process location when scheduling occurs. The Level 2 cache utilization is therefore better. In this case, the intranode communication ratio is not improved. The second new policy regroups the MPI processes on the nodes as much as possible (like in the *Placed* policy) but instead of binding each MPI process on its dedicated core, we let the operating system choose the placement (like in the regular *Round-Robin* policy). By doing so, the overall ratio of intranode communication versus internode communication is increased but we do not take anymore into account factors such as NUMA effects, memory hierarchy or die sharing by processes. Cache utilization will be negatively impacted when processes are placed according to this policy. The last policy also regroups process on the compute nodes but we force the intranode placement to be suboptimal: in this case we place the processes that communicate the most on opposite dies in the compute node. This last policy and the *Regroup* policy share a common point in that process placement within

Table 3. Overall intranode communication ratios in NAS kernels for two different placement policies

	BT		CG		LU		MG		SP	
	C	D	C	D	C	D	C	D	C	D
Round-Robin	56%	56%	33%	33%	66%	66%	62%	70%	55%	55%
Placed	69%	69%	85%	85%	82%	82%	73%	80%	69%	69%

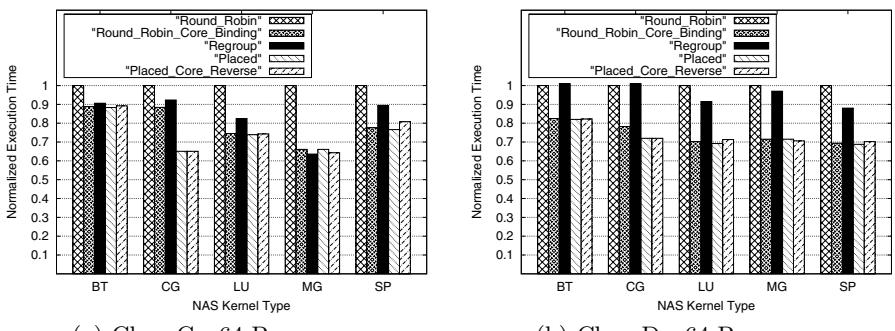


Fig. 7. Process placements comparison for MPICH2-Nemesis:MX

the node is very poor. The difference between the two is that with the *Placed Core Reverse* policy, cache utilization is better since processes are pinned to a core. Figure 7 shows the results: both *Round-Robin* and *Placed* policies are the same as in section 3.2, while *Round-Robin Core Binding*, *Regroup* and *Placed Core Reverse* represent the new placement policies described above. The results indicate that merely improving the intranode communication ratio is not enough to deliver better performance. Increasing this ratio without taking into account cache utilization leads to suboptimal results. Globally, our *Placed* policy delivers slightly better results than the others policies. The CG kernel is the benchmark for which the performance improvement is the most noticeable. These results suggest that cache utilization has a great influence on NAS performance. and advocates for a placement policy that takes architectural factors into consideration. However, we think that the Opteron compute nodes we used have a too small NUMA effect for it to impact applications performance. Also, the results obtained with the *Placed Core Reverse* policy show that despite the increase of traffic on the memory bus (induced by this policy), the impact on performance is limited on the compute nodes we used. Last, Level 2 cache is not shared between cores on the same die as explained in section 2.2. Those three aspects lead us to think that our placement policy could perform even better on nodes featuring more cores, with substantial NUMA effects, and where cache is effectively shared between cores. Finally, since we do not know the theoretical achievable performance of this particular cluster, we cannot assess how close to the optimal placement we actually are.

4 Related Works and Discussion

An adequate process placement is a necessity to fully take advantage of multicore environments. The MPI standard itself offers a set of routines that allow the applications developers to create and manipulate *topologies*. Using such topologies, MPI processes could be placed according application's communication pattern. However, only a subset of existing MPI applications makes use of these topology creation and management features. Also, as [12] points out, not all MPI implementations support these efficiently. At last, this mapping between the *virtual* topology created and the *physical* topology (the issue on which we focus in this paper) is outside the scope of the MPI standard. Our approach – deploying the MPI processes according to a matching between the application's communication pattern and the machine's architecture – is more generic and can benefit to any MPI application. Actually, this approach is not MPI-dependent, but rather *message-passing*-dependent. It can be applied even to non-MPI applications as long as the necessary information is collectable.

Placing MPI processes according to the underlying target architecture using graph theory has already been explored. Several vendor MPI implementations, such as the ones provided by Hewlett-Packard [13] or by IBM (according to [14]) make use of such mechanism. [12] also formalizes the problem with graphs. In these cases, however, the algorithm computing the final mapping is MPI implementation-specific. In our framework, we rely on an external piece of

software (Scotch) fully tailored for graph computations. This ensures both performance and scalability. Scotch is indeed able to work on very large classes of graphs, larger than the maximum number of MPI processes present in any MPI application.

As explained in section 2.2, architecture’s information is gathered thanks to PM²’s topology discovery mechanism. But this feature is fully embedded in this software stack and rather inconvenient to use. Such a topology discovery feature would be very useful for many other applications, especially MPI process managers. Hybrid MPI+OpenMP applications could also take advantage from it[15]. There is a clear need for a portable and accurate tool dedicated to topology discovery. We already started to work on this specific point.

The last issue we would like to address regards the gathering of applications communication patterns. In order to get the needed information to create the mapping, we have to execute a prior run of the applications compiled with a modified MPI implementation fitting our needs. Other works in this area also use the same coarse scheme. They also emphasize that tracing tools could be enhanced to address the specific issue of getting the amount of data exchanged between two given processes, as HP’s *Light Weight Instrumentation* [13] does. The necessary information could be an *estimate* of the communication flows that would allow us to *rank* pairs of processes accordingly. However, the ability to perform this prior run does not systematically exists. Other solutions should be investigated. For instance, relying on an application’s programmer’s knowledge is also possible but once again far from always possible. Could this kind of information be computed at *compile* time? Would it be possible to pass an option to the `mpicc` compiler that would *automatically* generate the customized command line? These are issues we would like to address in the near future.

5 Conclusion and Future Work

In this paper, we expose a method that leads an MPI application to better exploit its target architecture, especially complex and hierarchical multicore environments. This method relies on gathering information about the underlying hardware and the application’s communication pattern. This information is then used to create a mapping between MPI process ranks and each node’s core numbers. Finally an application-specific command line is generated. Our method uses free, open-source software and is not tightly integrated within a particular MPI implementation. Being able to place the processes according to the machine’s topology increases performance. We analyzed several placement policies and found out that merely increasing the intranode communication ratio in an application is not enough to deliver more performance: an adequate cache utilization is also mandatory to enhance performance. The current results are mitigated: our sophisticated placement policy improves MPI performance applications but should yield even better results on more complex architectures. We look forward to run our experiments on a different (and more complex) class of hardware.

Possible future works are numerous: as we previously stated, we began to work on a software tool that would allow us to easily extract hardware’s information.

Integrating information about I/O buses or even GPUs should also be considered. We would like to relax some constraints about the type of MPI applications falling into the scope of this work. We do consider *static* MPI applications, that is, applications where the amount of computing entities remains constant throughout the execution. This excludes both applications spawning new MPI processes and multithreaded ones. Since hybrid programming, mixing message-passing and multithreading, is considered as a possible way to better program multicore architectures, we plan to address the issue of MPI processes placement when OpenMP parallel sections appear in MPI processes. Another interesting direction would be to refine the application data regarding its communication pattern. For now, we did only consider a *spatial* pattern. But what about the *temporal* pattern? Indeed, the data exchanges occurring between a given pair of processes may vary during the application's execution. In order to take this phenomenon into account we will have to isolate application *time slices* and remap the MPI processes during such time slices. What granularity for slices would be the most beneficial to performance? Also, we would have to modify the mapping during execution. For intranode communication this task is easy but for internode communication we would have to migrate processes from one node to the other. Using virtual machines in this context might be a way to implement this. Also, some other influential factors such as contention on the memory and I/O buses could be taken more into consideration. This would lead to an even more refined placement policy but requires to comprehend thoroughly the target application. Generating MPI-implementation dependent information could also be considered in order to increase the policy's accuracy. Last, we plan to investigate the feasibility of gathering the application's information at compile time.

Acknowledgments. Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see <https://www.grid5000.fr>).

The authors wish to thank Sébastien Fourestier, our Scotch Guru.

References

1. Message Passing Interface Forum: MPI-2: Extensions to the message-passing interface (1997), <http://www.mpi-forum.org/docs/mpi-20.ps>
2. Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B.W., Lumsdaine, A., Castain, R.H., Daniel, D.J., Graham, R.L., Woodall, T.S.: Open MPI: Goals, concept, and design of a next generation MPI implementation. In: Kranzlmüller, D., Kacsuk, P., Dongarra, J. (eds.) EuroPVM/MPI 2004. LNCS, vol. 3241, pp. 97–104. Springer, Heidelberg (2004)
3. Argonne National Laboratory: MPICH2 (2004), <http://www.mcs.anl.gov/mpi/>
4. Squyres, J., et al.: Portable Linux Processor Affinity (2008), <http://www.open-mpi.org/projects/plpa/>
5. Namyst, R., Denneulin, Y., Geib, J.-M., Méhaut, J.-F.: Utilisation des processus légers pour le calcul parallèle distribué: l'approche PM2. Calculateurs Parallèles, Réseaux et Systèmes répartis 10, 237–258 (1998)

6. Thibault, S., Namyst, R., Wacrenier, P.-A.: Building Portable Thread Schedulers for Hierarchical Multiprocessors: the BubbleSched Framework. In: EuroPar, Rennes, France. ACM, New York (2007)
7. Pellegrini, F.: Scotch and LibScotch 5.1 User's Guide. ScAlApplix project, INRIA Bordeaux – Sud-Ouest, ENSEIRB & LaBRI, UMR CNRS 5800 (2008), <http://www.labri.fr/perso/pelegrin/scotch/>
8. Pellegrini, F.: Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs. In: Proceedings of SHPCC 1994, Knoxville, pp. 486–493. IEEE, Los Alamitos (1994)
9. Bolze, R., Cappello, F., Caron, E., Daydé, M., Despres, F., Jeannot, E., Jégou, Y., Lantri, S., Leduc, J., Melab, N., Mornet, G., Namyst, R., Primet, P., Quetier, B., Richard, O., Talbi, E.-G., Touche, I.: Grid 5000: a large scale and highly reconfigurable experimental Grid testbed. International Journal of High Performance Computing Applications 20, 481–494 (2006)
10. Buntinas, D., Mercier, G., Gropp, W.: Implementation and Evaluation of Shared-Memory Communication and Synchronization Operations in MPICH2 using the Nemesis Communication Subsystem. In: Parallel Computing, Selected Papers from EuroPVM/MPI 2006, vol. 33, pp. 634–644 (2007)
11. Myricom: MPICH2-MX (2009), <http://www.myri.com/scs/download-mpichmx.html>
12. Träff, J.L.: Implementing the MPI process topology mechanism. In: Supercomputing 2002: Proceedings of the 2002 ACM/IEEE conference on Supercomputing, pp. 1–14. IEEE Computer Society Press, Los Alamitos (2002)
13. Solt, D.: A profile based approach for topology aware MPI rank placement (2007), http://www.tlc2.uh.edu/hpcc07/Schedule/speakers/hpcc_hp-mpi_solt.ppt
14. Duesterwald, E., Wisniewski, R.W., Sweeney, P.F., Cascaval, G., Smith, S.E.: Method and System for Optimizing Communication in MPI Programs for an Execution Environment (2008), <http://www.faqs.org/patents/app/20080288957>
15. Rabenseifner, R., Hager, G., Jost, G.: Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes. In: Proceedings of the 17th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP 2009), Weimar, Germany, pp. 427–436 (2009)

Dynamic Communicators in MPI

Richard L. Graham and Rainer Keller

Oak Ridge National Laboratory
Computer Science and Mathematics Department
`{rlgraham,keller}@ornl.gov`

Abstract. This paper describes a proposal to add support for dynamic communicators to the MPI standard. This adds the ability to grow or shrink a specified communicator, under well defined conditions. The goal is to make it possible for a new class of applications – long-running, mission-critical, loosely coupled applications, running in a highly dynamic environment – to use MPI libraries for their communication needs, and to enable HPC applications to adjust to changing system resources. Implementation analysis indicates that performance impact on existing high-performance MPI applications should be minimal, or non-existent. The changes to MPI implementations are expected to be well compartmentalized.

1 Introduction

The Message Passing Interface (MPI) [3] is the ubiquitous choice for user-level communications in High Performance Computing (HPC). This standard serves as a basis for the implementation of high-performance communication systems enabling simulation codes to fully benefit from the capabilities of modern, high-performance networks, such as InfiniBand [1].

Communication systems targeting parallel applications define a notion of process groups. The Parallel Virtual Machine (PVM) specification [5] provides support for dynamic process groups, but PVM communications are not as efficient as MPI communications. The Unified Parallel C (UPC) and Co-Array Fortran are current language specifications that incorporate the notion of parallelism. The UPC specification [7] team definition is heavily influenced by the MPI standard, and is static in nature. The concept of process groups, or teams, was considered for the Fortran 2008 language standard [4], but has not been included in this specification.

The MPI communicator brings together the notion of a partitioned communication context and the MPI process groups that may communicate in this partitioned space, with communications occurring between an MPI rank in the local group and one or more ranks in the remote group. Two types of communicators are defined, intra-communicators where the local and remote groups are identical, and inter-communicators, where the local and remote groups are disjoint. All communications occur in the context of a communicator. Point-To-Point and collective communications explicitly reference the communicator in all

communications, while one-sided communications references the communicator through the window handle, and MPI file I/O operations references this through the file handle.

In the current MPI standard the groups associated with the communicators are static, fixed for life-time of the communicator. To change the MPI processes involved in a given communication pattern a new communicator must be constructed. Communicator creation is collective over the processes in the groups forming the new communicator, providing a time for all processes in the communicator to exchange information and initialize optimized communications. For example, the static nature of the communicator allows MPI implementations to cache the communication patterns used for collective operations, such as a broadcast, and initialize cached scratch buffers for such communications.

The static nature of the MPI communicator is a good match for current HPC computing environments where machine resources are allocated as static resource pools, with MPI implementations taking advantage of the communication optimizations this model provides. HPC simulation codes have also adapted to this mode of operation. However, there are other classes of parallel applications that run in much more dynamic environments, that can't easily adapt to the current communicator model used by MPI. In these environments the number of processes and the resources used by these change considerably over time. In general, these tend to be loosely coupled client/server types of applications where the clients rarely communicate with each other, and their use of collective communications is non-existent, or limited to server management activities. In general these include long running mission critical services, such as Condor based applications [6], applications with dynamic loads during runtime, such as bio-informatics applications [2,8], or even dynamic client/server applications like on-line gaming sites, to mention a few. Solution providers in these spaces would like to be able to take advantage of the large amount of effort going into creating portable and high-performance MPI library communication sub-systems, but require support for the dynamic process environment. In addition, to respond to changing machine resources, HPC application developers have expressed interest in MPI support for changing the size of existing communicators.

This paper describes a proposed extension to the MPI communicator concept that preserves the performance optimization opportunities inherent in the current standard, but also allows communicators to be dynamic in nature.

2 Dynamic Communicators

This paper proposes to add to the MPI standard support for dynamic communicators, defined to be communicators that vary in size over their healthy lifetime. Currently, MPI dynamic process support results in the creation of new communicators when adding MPI processes. In contrast, the current proposal is to change the size of an already existing communicator. Support is proposed for two classes of applications – loosely and tightly coupled. For the loosely coupled

applications, a mechanism with relaxed communicator consistency is proposed. At a given point in time members of the communicator may, for a finite amount of time, have different views of the communicator. Support for tightly coupled applications is aimed at providing high-performance communications support. To do so, communicators are kept consistent over all ranks while these are in use.

Providing support to shrink a communicator enables the support for a class of applications not currently offered by the MPI standard. Support for increasing communicator size may be viewed as syntactic candy, but does increase MPI's ease of use for applications needing such capabilities. Overall, this proposal sets the stage for MPI applications to be much more adaptable to changing application needs and system resources.

Support for tightly coupled and loosely coupled communicators is fundamentally different and is described in the following subsections.

2.1 Loosely Coupled Dynamic Communicators

The loosely coupled dynamic communicator model assumes lazy notification when the size of a communicator changes. As part of normal operations, a communication's target may vanish, leaving the library unable to complete requested operations while needing to continue normal library operations. For example, a sender may initiate communication to a rank that no longer exists, requiring the library to return an appropriate warning to the application, and continue running, even if the user has set the error handler to be `MPI_ERRORS_ARE_FATAL`.

A communicator may grow either by a request from an existing rank to grow the communicator, or by an pre-existing process requesting to join the communicator.

A communicator grow by one of it's members making a call to the routine:

`MPI_COMM_GROW (communicator_handle, n_to_grow, callback_func)`

INOUT	<code>communicator_handle</code>	communicator (handle)
IN	<code>n_to_grow</code>	number of ranks to add to the communicator (integer)
IN	<code>callback_func</code>	callback function for change notification (handle)

Unless `MPI_COMM_WORLD` is the communicator being expanded, the newly created ranks are also members of a new `MPI_COMM_WORLD`, whose members are the newly created MPI processes. A callback mechanism, described below, is used to notify other ranks in the communicator of the change. The newly created processes need to obtain the new communicator handle after the call to `MPI_INIT` or `MPI_INIT_THREAD` completes. The function

MPI_COMM_GET_RESIZED_HANDLE (communicator_handle)

OUT	communicator_handle	communicator (handle)
-----	---------------------	-----------------------

is introduced to obtain the handle to the communicator grown by the MPI_COMM_RESIZE routine. No input parameters are needed, as the newly created MPI process will have only the two default communicators: MPI_COMM_WORLD and MPI_COMM_SELF and, if not derived from MPI_COMM_WORLD, the resized communicator. If the expanded communicator is MPI_COMM_WORLD this is the handle returned, otherwise the handle to the third communicator is returned.

For an MPI process to connect to an existing communicator, it first needs to open a connection to the MPI server specified by `port_name`. To become a member of an existing communicator it uses the call

MPI_COMM_CONNECT_TO (port_name, communicator_name, newcomm)

IN	port_name	Network address (string, used only on root)
IN	communicator_name	Name of communicator to join (string)
OUT	newcomm	Communicator (handle)

requiring the communicator to be named. The list of existing communicators that may be accessed by the function

MPI_COMM_GET_COMMUNICATORS (port_name, comm_names, count)

IN	port_name	Network address (string, used only on root)
OUT	comm_names	Array of communicators (strings)
OUT	count	Number of communicators (integer)

Communicator size increases with each process connecting to it, with its assigned rank being the first available rank, starting at rank zero. An MPI process may not connect to a communicator of which it is already a member.

A rank can leave a communicator using the routine

MPI_COMM_EXIT (communicator)

INOUT	communicator	Communicator (handle)
-------	--------------	-----------------------

with the predefined handle MPI_COMM_ALL signaling MPI to cleanly exit from any connected communicator.

Change Notification. When processes join or leave a communicator, the other ranks in the communicator will be notified lazily. Since notification needs to be on a per communicator basis for layered library support, we propose two mechanisms, notification using a library initiated message delivered to each rank in

the communicator, or via a callback mechanism. In the first case the notification will occur if the application posted a receive with the pre-defined tag `MPI_NOTIFY_COMMUNICATOR_CHANGE`. The return buffer will include the communicator handle and rank information of the newly added ranks, and may return information on as many newly created ranks as the input buffer will hold. In the second case, a user defined callback registered upon communicator construction will be invoked, providing the ranks of the processes that either left or joined the communicator. The callback will be provided with communicator-handle, process rank, and whether this rank joined or left the communicator. Each rank it notified in-order of all changes to a given rank. Only local work is allowed in the callback routine.

`MPI_COMM_CHANGE_CALLBACK` (communicator, num_join, joined, num_left, left)

IN	communicator	Communicator that changed (handle)
IN	num_join	Number of ranks joining (integer)
IN	join	Array of ranks joining (integer)
IN	num_left	Number of ranks leaving (integer)
IN	left	Array of ranks leaving (integer)

Performance Considerations. Special attention need be given to communications performed in the context of loosely coupled communicators. One must consider how this impacts static communicator communications. Collective operations need special attention, as each rank in a communicator may have a different view of the communicator. It would be tempting to restrict the collective operations supported, however there is no good reason to do so. A best effort is assumed to complete these operations, with each process including the ranks it is aware of at the start of the collective operation. Since communicator composition may change as the collective operation progresses, algorithms need to be ready to handle such situations and not deadlock. If this happens, the return code indicates this, with a user-defined callback function called just before the collective operation returns reporting which ranks did not participate in the call. For example, if after a broadcast is issued some of the ranks that the root attempts to reach exit the communicator, the return code will indicate this, with the callback routine listing the ranks that did not get this data. It is clear that the performance of collective operations for such communicators will not be as good as that of static MPI collective operations, however for these loosely coupled applications, the performance of the collective operations is not as important as the convenience of using library provided collective operations.

The impact of support for dynamic communicators on the performance of point-to-point communications also requires careful consideration. Associating a given set of collective operations with a communicator is common practice. Therefore, it is possible to implement collective operations targeting dynamic communicators while not affecting static communicator collective operations.

However, the authors are unaware of implementations with communicator specific point-to-point, one-sided, or MPI I/O communications, and as such the performance implications on MPI library as a whole must be considered. For point-to-point and one-sided communications one must consider the (1) inability to complete such communication due to the target exiting the communicator, and (2) the impact of variable communicator size on access to internal library data structures. The first is not a performance problem as the initiator must already handle error conditions. Local MPI completion semantics continue to be sufficient, as the target exits a communicator at it's initiative, with no further data delivery expected. The second item could have negative performance impact on communications using static communicators, as reallocating data structures may change their memory location. To avoid accessing stale data, these data structures need to be accessed atomically. It should be possible to keep the number of such data structures small, perhaps even as low as a single data structure, making the performance cost similar to that of providing a thread safe MPI library. Another approach could be at communicator creation to set an upper limit on it's size, or even providing support for loosely coupled communicators as a compile time option.

2.2 Tightly Coupled Dynamic Communicators

All ranks in a tightly coupled dynamic communicators share the same view of the communicator while it is in use. To keep such communicators consistent, the routine for changing communicator size is defined to be collective, and there is not need to explicitly declare these as a dynamic communicators. These becomes a dynamic communicators, of size `requested_size` when the following function is called:

`MPI_COMM_RESIZE_ALL (communicator_handle, requested_size, removed_ranks)`

INOUT	<code>communicator_handle</code>	communicator (handle)
IN	<code>requested_size</code>	new size (integer)
IN	<code>removed_ranks</code>	array of ranks to be removed (struct)

This function may be called on any communicator but `MPI_COMM_SELF`, which is not allowed to change it's size. The list of ranks to be removed is relevant only when the group is being shrunk, and even then is optional. It includes the group (local or remote) and the rank within this group of the processes to be removed. In it's absence (NULL pointer) the highest order ranks of the local communicator will be removed. This list must be consistent across all ranks in the communicator.

When shrinking a communicator the resulting ranks in the redefined communicator of size M will be dense, with values in the range of 0 to $M - 1$, possibly changing a process's rank within the communicator.

When a communicator is grown new processes are created, and unless MPI_COMM_WORLD is the communicator being expanded, the newly created ranks are members of a new MPI_COMM_WORLD, whose members are the newly created MPI processes. As in the case of the loosely coupled communicator, the newly created MPI processes obtain the communicator handle for the expanded communicator with the function MPI_COMM_GET_RESIZED_HANDLE, as defined on page 119. If the processes were not started as the result of resizing a communicator, the handle MPI_COMM_NULL is returned. If the return handle is not the null communicator handle, this handle is used as input to the routine MPI_COMM_RESIZE_ALL, to complete the new communicator initialization.

The collective nature of the routine provides a synchronization point at which an implementation can redo any communicator based optimization, avoiding loss of communications performance. However, this does require completing all outstanding communications before resizing the communicator, forming well defined communication epochs.

3 Example of the Loosely-Coupled Communicator

In Algorithm 1 a simple application is presented to show the usage of the loosely-coupled dynamic communicators. As the solution is dynamically evolving, the

```

1 MPI_INIT(...);
2 MPI_COMM_GET_RESIZED_HANDLE(comm);
   /* If we are one of the grown processes, receive from root */
3 if comm != MPI_COMM_NULL then
4   MPI_RECV(part of the tree to work on, 0, ..., comm);
5 MPI_COMM_DUP(MPI_COMM_SELF, selfdup);
6 while no solution found do
   /* Generate branches in a tree */
   /* If load is too high, start process and send part of tree */
7   if load-imbalance then
8     MPI_COMM_GROW(selfdup, 1, NULL);
9     MPI_SEND(part of the tree, 1, ..., selfdup);
10 end
   /* Process tree, then check for updates from grown processes */
11 MPI_IPROBE(MPI_ANY_SOURCE, MPI_NOTIFY_COMMUNICATOR_CHANGE, selfdup,
   flag, status);
12 if flag then
13   MPI_RECV(solution, status.MPI_SOURCE, ..., selfdup);
14 end
15 if comm != MPI_COMM_NULL then
16   MPI_SEND(solution, 0, ..., comm);
17 MPI_COMM_EXIT(comm);

```

Algorithm 1. Simple dynamic tree-based algorithm

computing process generates solutions to be verified in a tree. After trying to prune the tree, if the load per process is still considered too high, the process dynamically creates processes, locally known to him. Then this process solves the sub-tree assigned to it – being able to again spawn other processes on the fly.

4 Summary

This paper proposes adding support for the concept of dynamic communicators to the MPI standard. The goal of this addition is to support new classes of MPI applications, e. g. long-running, loosely coupled application running in a dynamic environment, and to allow HPC applications to adjust to changing system resources. Performance impact on existing MPI applications is minimal. It may prove non-existent after implementation. Future work is planned to implement this new feature set.

References

1. Infiniband Trade Association. Infiniband architecture specification vol 1. release 1.2 (2004)
2. Cortés, A., Planas, M., Millán, J.L., Ripoll, A., Senar, M.Á., Luque, E.: Applying load balancing in data parallel applications using DASUD. In: Dongarra, J., Laforenza, D., Orlando, S. (eds.) EuroPVM/MPI 2003. LNCS, vol. 2840, pp. 237–241. Springer, Heidelberg (2003)
3. The MPI Forum. Mpi: A message-passing interface standard - version 2.1 (June 2008)
4. Reid, J.: The new features of fortran 2008. SIGPLAN Fortran Forum 27(2), 8–21 (2008)
5. Sunderam, V.S.: Pvm: A framework for parallel distributed computing. Concurrency: Practice and Experience 2, 315–339 (1990)
6. Thain, D., Tannenbaum, T., Livny, M.: Distributed computing in practice: the condor experience. Concurrency - Practice and Experience 17(2-4), 323–356 (2005)
7. UPC Consortium. UPC Language Specifications, v1.2 (2005)
8. Villmann, T., Hammer, B., Seiffert, U.: Perspectives of self-adapted self-organizing clustering in organic computing. In: Ijspeert, A.J., Masuzawa, T., Kusumoto, S. (eds.) BioADIT 2006. LNCS, vol. 3853, pp. 141–159. Springer, Heidelberg (2006)

VolpexMPI: An MPI Library for Execution of Parallel Applications on Volatile Nodes

Troy LeBlanc, Rakhi Anand, Edgar Gabriel, and Jaspal Subhlok

Department of Computer Science, University of Houston
`{tpleblan,rakhi,gabriel,jaspal}@cs.uh.edu`

Abstract. The objective of this research is to convert ordinary idle PCs into virtual clusters for executing parallel applications. The paper introduces VolpexMPI that is designed to enable seamless forward application progress in the presence of frequent node failures as well as dynamically changing networks speeds and node execution speeds. Process replication is employed to provide robustness in such volatile environments. The central challenge in VolpexMPI design is to efficiently and automatically manage dynamically varying number of process replicas in different states of execution progress. The key fault tolerance technique employed is fully distributed sender based logging. The paper presents the design and a prototype implementation of VolpexMPI. Preliminary results validate that the overhead of providing robustness is modest for applications having a favorable ratio of communication to computation and a low degree of communication.

1 Introduction

Idle desktop computers represent an immense pool of unused computation, communication, and data storage capacity [1,2]. The advent of multi-core CPUs and increasing deployment of Gigabit capacity interconnects have made mainstream institutional networks an increasingly attractive platform for executing scientific codes as “guest” applications. Idle desktops have been successfully used to run sequential and master-slave task parallel codes, most notably under Condor [3] and BOINC [4]. In the recent past, some of the largest pools of commercial compute resources, specifically Amazon [5] and Google [6], have opened up part of their computation farms for public computing. Often these computers are very busy on a few occasions (e.g. Christmas shopping) and underutilized the rest of the time. This new phenomenon is often referred to as “cloud computing”.

However, a very small fraction of idle PCs are used for such guest computing and the usage is largely limited to sequential and “bag of tasks” parallel applications. In particular, we are not aware of any MPI implementation that is used widely to support execution on idle desktops. Harnessing idle PCs for communicating parallel programs presents significant challenges. The nodes have varying compute, communication, and storage capacity and their availability can change frequently and without warning as a result of, say, a new host application, a reboot or shutdown, or just a user mouse click. Further, the nodes are

connected with a shared network where available latency and available bandwidth can vary. Because of these properties, we refer to such nodes as *volatile* and parallel computing on volatile nodes is challenging.

This paper introduces VolPEEx (Parallel Execution on Volatile Nodes) MPI that represents a comprehensive and scalable solution to execution of parallel scientific applications on virtual clusters composed of volatile ordinary PC nodes. The key features of our approach are the following:

1. *Controlled redundancy*: A process can be initiated as two (or more) replicas. The execution model is designed such that the application progresses at the speed of the fastest replica of each process, and is unaffected by the failure or slowdown of other replicas. (Replicas may also be formed by checkpoint based restart of potentially failed or slow processes, but this aspect is not implemented yet).
2. *Receiver based direct communication*: The communication framework supports direct node to node communication with a *pull* model: the sending processes buffer data objects locally and receiving processes contact one of the replicas of the sending process to get the data object.
3. *Distributed sender based logging*: Messages sent are implicitly logged at the sender and are available for delivery to process instances that are lagging due to slow execution or recreation from a checkpoint.

VolpexMPI is designed for applications with moderate communication requirements and is expected to scale to 100s of nodes on institutional LANs. Certainly many parallel applications will not run effectively on ordinary desktops under VolpexMPI (or any other framework) because of memory and communication requirements that can only be met with dedicated clusters. In particular, for an application to run effectively on volatile nodes, it must have a low communication degree and limited sensitivity to latency. It has been shown that many scientific applications have a low degree stencil as the dominant communication pattern [7,8]. Hence, we believe that many parallel applications are good candidates and an important goal of this project is to identify the extent of applicability of this approach.

An example motivating application is Replica Exchange Molecular Dynamics (REMD) formulation [9] where each node runs a piece of molecular simulation at a different temperature using the AMBER program [10]. At certain time steps, communication occurs between neighboring nodes based on the Metropolis criterion, in case a given parameter is less than or equal to zero. REMD requires low volume loosely coupled communication making it a good candidate for VolpexMPI. It is currently implemented in the Volpex environment [11] but not yet ported to MPI.

This paper focuses on the design, implementation and validation of VolpexMPI. The implementation works on clusters and PC grids. Preliminary results presented for a commodity PC cluster compare VolpexMPI to Open MPI and analyze the overhead of replication and node failure.

2 Fault Tolerance in MPI

The MPI specifications are rather vague about failure scenarios. In recent years MPI implementations have been developed to deal with process and network failures. Fault tolerant methods supported by various implementations of MPI can be divided into three categories: 1) extending the semantics of MPI, 2) check-point restart mechanism, and 3) replication techniques.

FT-MPI [12] is the best known representative of the approach of extension of semantics for failure management. The specification of FT-MPI defines the status of the MPI handles and messages in case of a process failure. FT-MPI has the ability to either replace a failed process, or continue execution without it. The library deals only with MPI-level recovery, but lets the application manage the recovery of user level data items in a performance efficient manner [13]. However, it requires significant modifications to the application, and thus does not provide a transparent fault-tolerance mechanism.

MPICH-V [14] belongs to the category of MPI libraries that employ checkpoint-restart mechanisms for fault tolerance. It is based on uncoordinated checkpointing and pessimistic message logging. The library stores all communications of the system on reliable media through the usage of a *channel memory*. In case of a process failure, MPICH-V is capable of restarting the failed application process from the last checkpoint and replay all messages to that process. Similarly to MPICH-V, RADICMPI [15] also fundamentally relies on checkpointing MPI processes, but tries to avoid any central instance or single point of failure within its overall design. Although some of the conceptual aspects of VolpexMPI are similar to MPICH-V and RADICMPI, there are key architectural differences. Neither of these two MPI libraries are designed to run multiple replicas of an MPI process. Thus, while VolpexMPI can continue the execution of an application seamlessly in case of a process failure if replicas are available for that process, MPICH-V and RADICMPI will have to deal with the overhead generated by restarting processes from an earlier checkpoint. Also, message logging in VolpexMPI is fully distributed on host nodes themselves and there is no equivalent of channel memories.

MPI/FT [16] provides transparent fault tolerance by replicating MPI processes and introducing a central coordinator. The library is able to recognize malicious data by using a global voting algorithm among replicas. However, this feature also leads to an exponential increase in the number of messages with the number of replicas: each replica sends every message to all destination replicas. VolpexMPI avoids the penalty resulting from this communication scheme by ensuring that a message is pulled from exactly one sender with receiver initiated communication. P2P-MPI [17] is also based on replication techniques where each set of process replicas maintain a master replica that distributes messages. Fault detection is done using a gossip-style protocol [18], which has the ability to scale well and provides timely detection of failures. P2P-MPI also takes advantage of locality awareness and co-allocation strategies. A key difference is that, unlike P2P-MPI, VolpexMPI utilizes a pull based model for data communication that ensures that the application advances at the speed of the fastest replica for each process.

VolpexMPI also employs replication for fault tolerance. It has been shown, that check-pointing offers a good solution when failures are infrequent whereas replication offers better performance when failure rates are high [19]. VolpexMPI is designed to balance replication and checkpoint-restart, although the current implementation is limited to replication.

3 VolpexMPI Design

VolpexMPI is an MPI library implemented from scratch focusing on fault tolerance using process replication. As of today, the library supports around 40 functions of the MPI-1 specification. The design of the library is centered around five major building blocks, namely the MPI API layer, the point-to-point communication module, a buffer management module, a replica selection module and a data transfer module.

The point-to-point communication module of VolpexMPI has to be designed for MPI processes with multiple replicas in the system. This is required in order to handle the main challenge of grids built from idle PCs, namely the fact that processes are considered fundamentally unreliable. A process might go away for no obvious reason, such as the owner pressing a button on the keyboard. From the communication perspective, the library has two main goals: (I) avoid increasing the number of messages on the fly by a factor of $n_{replicas} \times n_{processes}$, i.e., every process sending each message to every replica, and (II) make the progress of the application correspond to the fastest replica for each process.

In order to meet the first goal, the communication model of VolpexMPI deploys a receiver initiated message exchange between processes where data is pulled by the receiver from the sender. In this model, the sending node only buffers the content of a message locally, along with the message envelope. Furthermore, it posts for every replica of the corresponding receiver rank, a non-blocking, non-expiring receive operation. When contacted by a receiver process about a message, a sender participates in the transfer of the message if it is buffered, otherwise informs the receiver that the message is not available.

The receiving process polls a potential sender and waits then for the data item or a notification that the data is not available. As of today, VolpexMPI does not support wildcard receive operations as an efficient implementation poses a significant challenge. A straight-forward implementation of `MPI_ANY_SOURCE` receive operations is possible, but the performance would be significantly degraded compared to non-wildcard receive operations.

Since different replicas can be in different execution states, a message matching scheme has to be employed to identify which message is being requested by a receiver. For deterministic execution, a simple scheme that timestamps messages by counting the number of messages exchanged between pairs of processes is applied based on the tuple [communicator id, message tag, sender rank, receiver rank]. These timestamps are also used to monitor the progress of individual process replicas for resource management. Furthermore, a late replica can retrieve an older message with a matching logical timestamp, which allows restart of a process from a checkpoint.

The buffer management module provides the functionality to store and retrieve an MPI message based on the tuple described above. An important question is whether the message buffers on the sender processes must be maintained for the duration of execution or whether they can be cleared at some point. From the logical perspective, a message buffer can never be cleared due to the fact that, even if all replicas of a particular rank have received a given message, all of them might fail to finish the execution. Thus, a new replica of that process might have to be started, which would have to retrieve all messages. Our current approach employs a circular buffer where the oldest log entry is removed when the buffer is full. The long-term goal is to coordinate the size of the circular buffer with checkpoints of individual processes, which will allow guaranteed restarts with a bounded buffer size.

In order to meet the goal that the progress of an application correspond to the fastest replica for each process, the library has to provide an algorithm which allows a process to generate an order in which to contact the sender replicas. This is the main functionality provided by the replica-selection module. The algorithm utilized by the replica-selection module has to handle two seemingly contradicting goals: on one hand, it would be beneficial to contact the “fastest” replica from the performance perspective. On the other hand, the library does not want to slow-down the fastest replica by making it handle significantly larger number of messages, especially when a message is available from another replica. The specific goal, therefore, is to determine a replica which is “close” to the execution state of the receiver process. Currently the library utilizes a simple approach which groups replicas into ‘teams’. A receiver tries to contact the first the replica within its team, and only contacts a replica of another team if its own replica does not response within a given time slot. This is, however, a topic of active research with more sophisticated algorithms in the process of being implemented.

The data transfer module of VolpexMPI relies on a socket library utilizing non-blocking sockets. In the context of VolpexMPI, the relevant characteristics of this socket library are the ability to handle failed processes, on-demand connection setup in order to minimize the number of network connections, an event delivery system integrated into the regular progress engine of the socket library and the notion of timeouts for both communication operations and connection establishment. The latter feature will be used in future versions of VolpexMPI to identify replicas which are lagging significant.

The startup of a VolpexMPI application utilizes a customized `mpirun` program which takes the desired replication level as a parameter, in addition to the number of MPI processes, the name of the executable, and a list of hosts where the processes shall be started. As of today, the startup mechanism relies on secure shell operations. However, we anticipate to extend this section of the code in the near future by customized BOINC or CONDOR functions. `mpirun` has furthermore the functionality to inform all MPI processes about their rank, the team they belong to, as well as the information required by an MPI process to contact any other process within this application.

4 Experiments and Results

This section describes the experiments with the VolpexMPI library and the results obtained. VolpexMPI has been deployed on a small cluster as well as pool of desktop PCs. Although VolpexMPI is designed for PC grids and volunteer environments, experimental results are shown for a regular cluster in order to determine the fundamental performance characteristics of VolpexMPI in a stable and reproducible environment. The cluster utilizes 29 compute nodes, 24 of them having a 2.2 GHz dual core AMD Opteron processor, and 5 nodes having two 2.2GHz quad-core AMD Opteron processors. Each node has 1 GB main memory per core and network connected by 4xInfiniBand as well as a 48 port Linksys GE switch. For evaluation we utilize the Gigabit Ethernet network interconnect of the cluster to compare VolpexMPI run times to Open MPI [20] v1.2.6. and examine the impact of replication and failure on performance.

First, we document the impact of the VolpexMPI design on the latency and the bandwidth of communication operations. For this, we ran a simple ping-pong benchmark using both Open MPI and VolpexMPI on the cluster. The results shown in Figure 1 indicate, that the receiver based communication scheme used by VolpexMPI can achieve close to 80% of the bandwidth achieved by Open MPI. The latency for a 4 byte message increases from roughly 0.5ms with Open MPI to 1.8ms with VolpexMPI. This is not surprising as receiver based communication requires a ping-pong exchange before the actual message exchange.

Next, the NAS Parallel Benchmarks (NPBs) are executed for various process counts and data class set sizes. For each experiment, the run times were captured as established and reported in the NPB with the normal `MPI_Wtime` function calls for start and stop times. Since VolpexMPI targets the execution of applications with moderate number of processes, we present results obtained for 8 process and 16 process scenarios.

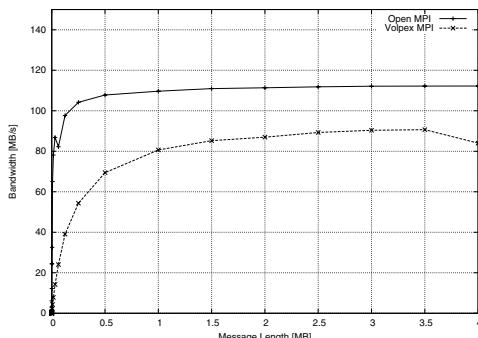


Fig. 1. Bandwidth comparison of OpenMPI and VolpexMPI using a Ping-Pong Benchmark

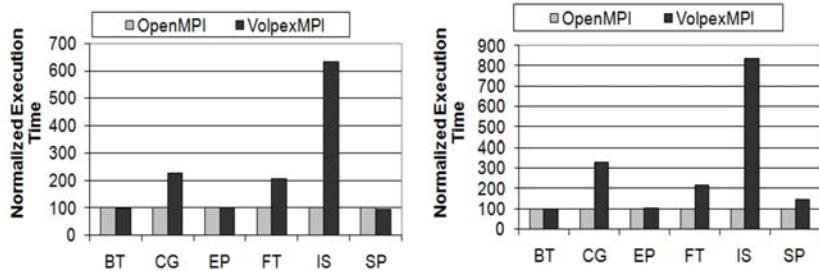


Fig. 2. Comparison of OpenMPI to VolpexMPI for Class B NAS Parallel Benchmarks using 8 Processes (left) and 16 Processes (right)

Figure 2 shows results for runs of 8 processes (left) and 16 processes (right) utilizing the Class B data sets for six of the NPBs. We have excluded LU and MG from our experiments due to their use of `MPI_ANY_SOURCE` which is not currently supported in VolpexMPI. These reference executions did not employ redundancy (x1). The run times for Open MPI are shown for comparison in the bar graph. All times are noted as normalized execution times with a reference time of 100 for Open MPI. The overhead incurred in the VolpexMPI implementation is virtually non-existent for BT, SP, and EP for the 8 and 16 processes, except that SP shows a noticeable overhead of 45% for 16 processes. The overhead for CG, FT and IS is significantly higher due to a variety of reasons, such as a greater use of collective calls such as `MPI_Alltoall(v)`, and in the case of IS, a ratio of computation to communication which is unfavorable to higher-latency environments. This also documents the fact that the class of applications considered suitable for execution with VolpexMPI have to follow a sparse communication scheme, i.e., a process should optimally only communicate with a small number of other processes, and should have a favorable communication to computation ratio. These requirements broadly hold for BT, SP and EP, but not necessarily for CG, FT and IS.

Next, we document the effect of executing an application with multiple copies of each MPI process. The left part of Figure 3 shows the normalized execution times of VolpexMPI for the 8 process NPBs running with no (same as single) redundancy (x1), double redundancy (x2) and triple redundancy (x3). The results indicate that, for most benchmarks the overhead due to redundant execution is minimal if no failure occurs, i.e. executing multiple copies of each MPI processes does not impose a significant performance penalty in the VolpexMPI scheme/model. Note, that this is a significant improvement over the replication based related work in the field. The benchmarks that show some sensitivity to replication are CG and SP, and the reasons are currently under investigation. Since Open MPI is not designed for utilizing redundant nodes, there are no directly comparable results for the double redundancy (x2) and triple redundancy (x3) runs.

Finally, we document the performance impact of a process failure for the NAS Parallel Benchmarks when using VolpexMPI. For this, we inserted into the source code of each benchmark some statements which terminate the execution of the second replica of rank 1 in `MPI_COMM_WORLD`, emulating a process failure. All

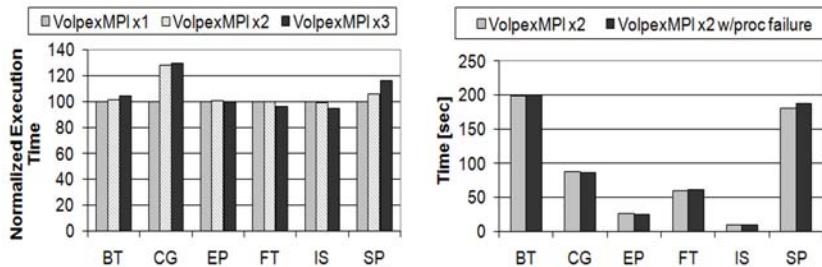


Fig. 3. Comparison of VolpexMPI execution times for 8 MPI processes with varying degree of replication (left) and in case of a process failure (right)

processes communicating with the terminated process will thus have to repost all pending communication operations to the only remaining replica of process 1. This test case represents one of the worst case scenarios for VolpexMPI, since the number of processes communicating with a single process doubles at runtime. Killing more than one process would actually relieve the remaining processes with rank 1, since the number of communication partners is reduced.

The results shown in the right part of Figure 3 show virtually no overhead in the scenario outlined above compared to the fault-free execution of the same benchmark using double redundancy. There are two potential sources for overhead. The first comes from the fact that the surviving process with rank 1 is being queried for data items by the second “team” of processes. Second, detecting the process failure is as of today based on a timeout mechanism or the break-down of a TCP connection. However, since the correct result of the simulation is available as soon as any replica of each process finishes the execution, these overheads might not necessarily show up in the final result.

5 Conclusions

This paper introduces VolpexMPI, an MPI library designed for robust execution of parallel applications on PC grids. The key design goal is efficient execution with replicated processes. The library employs a receiver based communication model between the processes, and a distributed, sender based message logging scheme.

We demonstrated with a prototype implementation the necessity to focus on the right class of applications for Volpex MPI, namely those with a favorable communication to computation ratio and a modest degree of communication. Benchmarks having the required characteristics show only a minor overhead compared to a standard MPI library. More important, utilizing multiple replicas for each process does not impose a notable overhead for the majority of the NAS benchmarks. Also, the NAS Parallel Benchmarks analyzed could successfully survive a process failure without suffering a major performance degradation. Hence, the central functionality and performance goals for VolpexMPI are satisfied.

The ongoing work on VolpexMPI includes developments in algorithms and execution environment. We are working on integrating checkpoint-restart with Volpex-MPI to dynamically manage replication by recreating slow and failed replicas from healthy replicas. We also plan to deploy and evaluate VolpexMPI on a large campus PC grid. Currently CONDOR and BOINC are being investigated as vehicles for integrated deployment.

We are investigating several applications as candidates for execution on PC clusters and building simulation tools to rapidly assess the suitability of an application for Volpex-MPI. In particular, we are in active discussions with a research group at the University of Houston which develops the Replica Exchange Molecular Dynamics (REMD) application [9]. The memory and compute requirements of this application combined with its low but important communication requirements make the application ideally suited for VolpexMPI.

Acknowledgments. Partial support for this work was provided by the National Science Foundation's Computer Systems Research program under Award No. CNS-0834750. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

1. Anderson, D., Fedak, G.: The computation and storage potential of volunteer computing. In: Sixth IEEE International Symposium on Cluster Computing and the Grid (May 2006)
2. Kondo, D., Taufer, M., Brooks, C., Casanova, H., Chien, A.: Characterizing and evaluating desktop grids: An empirical study. In: International Parallel and Distributed Processing Symposium, IPDPS 2004 (April 2004)
3. Thain, D., Tannenbaum, T., Livny, M.: Distributed computing in practice: the condor experience. *Concurrency - Practice and Experience* 17(2-4), 323–356 (2005)
4. Anderson, D.: Boinc: A system for public-resource computing and storage. In: Fifth IEEE/ACM International Workshop on Grid Computing (November 2004)
5. Amazon webservices: Amazon Elastic Compute Cloud, Amazon EC2 (2008), <http://www.amazon.com/gp/browse.html?node=201590011>
6. Google Press Center: Google and IBM Announce University Initiative to Address Internet-Scale Computing Challenges (October 2007), http://www.google.com/intl/en/press/pressrel/20071008_ibm_univ.html
7. Tabe, T., Stout, Q.: The use of the MPI communication library in the NAS Parallel Benchmark. Technical Report CSE-TR-386-99, Department of Computer Science, University of Michigan (November 1999)
8. Kerbyson, D., Barker, K.: Automatic identification of application communication patterns via templates. In: Proc. 18th International Conference on Parallel and Distributed Computing Systems (PDCS 2005), Las Vegas, NV (September 2005)
9. Sugita, Y., Okamoto, Y.: Replica-exchange molecular dynamics method for protein folding. *Chemical Physics Letters* 314, 141–151 (1999)
10. Case, D., Pearlman, D., Caldwell, J.W., Cheatham, T., Ross, W., Simmerling, C., Darden, T., Merz, K., Stanton, R., Cheng, A.: Amber 6 Manual (1999)

11. Kanna, N., Subhlok, J., Gabriel, E., Cheung, M., Anderson, D.: Redundancy tolerant communication on volatile nodes. Technical Report UH-CS-08-17, University of Houston (December 2008)
12. Fagg, G.E., Gabriel, E., Chen, Z., Angskun, T., Bosilca, G., Pjesivac-Grbovic, J., Dongarra, J.J.: Process fault-tolerance: Semantics, design and applications for high performance computing. *International Journal of High Performance Computing Applications* 19, 465–477 (2005)
13. Ltaief, H., Gabriel, E., Garbey, M.: Fault Tolerant Algorithms for Heat Transfer Problems. *Journal of Parallel and Distributed Computing* 68(5), 663–677 (2008)
14. Bouteiller, A., Cappello, F., Herault, T., Krawezik, G., Lemarinier, P., Magniette, F.: Mpich-v2: a fault tolerant mpi for volatile nodes based on pessimistic sender based message logging. In: SC 2003: Proceedings of the 2003 ACM/IEEE conference on Supercomputing, Washington, DC, USA, vol. 25. IEEE Computer Society, Los Alamitos (2003)
15. Duarte, A., Rexachs, D., Luque, E.: An Intelligent Management of Fault Tolerance in Cluster Using RADICMPI. In: Mohr, B., Träff, J.L., Worringen, J., Dongarra, J. (eds.) PVM/MPI 2006. LNCS, vol. 4192, pp. 150–157. Springer, Heidelberg (2006)
16. Batchu, R., Neelamegam, J.P., Cui, Z., Beddu, M., Skjellum, A., Yoginder, D.: Mpi/ft tm: Architecture and taxonomies for fault-tolerant, message-passing middleware for performance-portable parallel computing. In: Proceedings of the 1st IEEE International Symposium of Cluster Computing and the Grid, pp. 26–33 (2001)
17. Genaud, S., Rattanapoka, C.: Large-scale experiment of co-allocation strategies for peer-to-peer supercomputing in p2p-mpi. In: IEEE International Symposium on Parallel and Distributed Processing, 2008. IPDPS 2008, pp. 1–8 (2008)
18. Van Renesse, R., Minsky, Y., Hayden, M.: A gossip-style failure detection service. Technical report, Ithaca, NY, USA (1998)
19. Zheng, R., Subhlok, J.: A quantitative comparison of checkpoint with restart and replication in volatile environments. Technical Report UH-CS-08-06, University of Houston (June 2008)
20. Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B.W., Lumsdaine, A., Castain, R.H., Daniel, D.J., Graham, R.L., Woodall, T.S.: Open MPI: Goals, concept, and design of a next generation MPI implementation. In: Kranzlmüller, D., Kacsuk, P., Dongarra, J. (eds.) EuroPVM/MPI 2004. LNCS, vol. 3241, pp. 97–104. Springer, Heidelberg (2004)

Using Non-blocking I/O Operations in High Performance Computing to Reduce Execution Times

David Buettner, Julian Kunkel, and Thomas Ludwig

Ruprecht-Karls-Universität Heidelberg
Im Neuenheimer Feld 348, 69120 Heidelberg, Germany
mail@buettnerdavid.de
<http://pvs.informatik.uni-heidelberg.de/>

Abstract. As supercomputers become faster, the I/O part of applications can become a real problem in regard to overall execution times. System administrators and developers of hardware or software components reduce execution times by creating new and optimized parts for the supercomputers. While this helps a lot in the struggle to minimize I/O times, adjustment of the execution environment is not the only option to improve overall application behavior.

In this paper we examine if the application programmer can also contribute by making use of non-blocking I/O operations. After an analysis of non-blocking I/O operations and their potential for shortening execution times we present a benchmark which was created and run in order to see if the theoretical promises also hold in practice.

Keywords: MPI-IO, MPICH2, PVFS2, non-blocking I/O, PIOviz, benchmarking.

1 Introduction

Used in various areas of research, especially weather and climate research, but also outside of research in production systems in the areas of finance, geophysics, energy, and others [8], supercomputers provide the mandatory computing power. While the developments over the recent years provide more and more resources, the race towards faster supercomputer seems to go on without an end in sight.

With the new computing power and faster communication networks the applications are getting better at producing big amounts of useful data. Not only does this lead to the need of writing checkpoints during program execution, but also results in the desire to save intermediate results. But the developments concerning parallel I/O were not able to keep up with these rapid improvements. Therefore I/O is becoming a bigger bottleneck in comparison to calculation and communication [12].

Special hardware like RAID-systems and dedicated I/O servers as well as special software like parallel file systems are being developed to reduce this bottleneck and offer a variety of options which can be adjusted to fit the individual

application requirements. Together with the MPI [1] and MPI-2 [2] standards these components provide an easy to use computation environment for the application programmer.

In order to provide maximum performance for a given task, different hardware setups can be chosen for a supercomputer and various optimization options in the different software layers are available. Most of these optimizations are done outside the application such as the choosing of different RAID levels and the use of algorithms like **two-phase-I/O** and **data sieving** in the MPI implementation **MPICH2**. While these methods can speed up application performance they are not the only way to do so. Since in the application itself more information about the manipulated data and its timing are available than to the supporting software layers other optimizations can be used inside the application to minimize its execution time. One of these methods is the use of non-blocking I/O operations. The remainder of this paper is structured as follows: Section 2 will present related work and discuss how these are different to this analysis. Section 3 will introduce the theory behind non-blocking I/O operations and their potential to reduce execution times. A benchmark designed to measure the actual performance gains possible will be presented in section 4 and achieved results with this benchmark will be presented in section 5. Finally the conclusion in section 6 will give an outlook on future work.

2 Related Work and State-of-the-Art

The developments in high performance computing have been accompanied by the creation of a variety of benchmarks. Modeling different application behaviors with respect to CPU usage, communication patterns and I/O behavior, these benchmarks can be used to test how a given system performs for a given set of problems or to help decide which kind of system to acquire.

Nearly all of the existing benchmarks like the **Effective Communication and File-I/O Bandwidth Benchmarks** [14] use the MPI interface for their tests. In addition to the MPI interface some implemented tests use the **POSIX-I/O** functions. This for example is done in the **PRIOMark** [15] benchmark, which is, as far as we know, also the only benchmark which tests non-blocking I/O operations. The **PRIOMark** benchmark depends on the used MPI implementation to support non-blocking I/O operations and focuses on the question of how much the execution times of the I/O part and the calculation part increase when using non-blocking I/O functions. It does not examine the fact that while each part can become longer in execution, overall execution time of the application can be reduced since they are executed in parallel.

Non-blocking communication has been benchmarked in [7] and [16]. The focus in [7] is on the question if a given MPI implementation uses threads internally and how much overhead is induced by doing so.

We propose a benchmark which focuses on the reduction of overall execution time one can achieve by using non-blocking I/O functions. In addition we will present the benchmark results here.

3 The Theory behind Non-blocking I/O Operations

Thanks to modern operating systems and modern hardware a process waiting for a blocking call, e.g. an I/O call, to finish, is being blocked. Another process can make use of the resulting unused CPU cycles. While this works well on personal computers, supercomputers usually do not have more than one production process running on each CPU at a time. So while this sequential process is waiting for the return of an I/O function, CPU cycles are unused.

While it would not necessarily work for every application, in several cases the calculation could go on while data is being written. Since the blocking versions of the I/O calls only return when writing has terminated, the calculation is stopped during this time as shown at the bottom of Fig. 1. Assuming that special hardware like DMA modules is available and that the used MPI implementation supports real non-blocking I/O functions, theory suggests that either I/O or calculation can be hidden entirely behind the other depending on which takes longer. Fig. 1 shows a version where both parts take the same amount of time and presents the non-blocking scenario in which we first assume that the use of non-blocking I/O does not increase the execution time of either part.

Neglecting the fact that the I/O part is not only done by special hardware three scenarios are interesting:

- **I/O > calculation:** The calculation can be hidden entirely behind the I/O phases. The time saved is the time needed for the calculation.
- **I/O < calculation:** The I/O can be hidden entirely behind the calculation phases. The time saved is the time needed for the I/O.
- **I/O = calculation:** Both I/O and calculation take up the equal amount of time and one hides the other. Half of the execution time can be saved.

While overhead of setting up the I/O operations and additional work for the switch between both parts influence the individual execution times, this gives us an upper limit to the performance boost we can expect.

A closer look at the theory suggests that what the real possible achievement will be slightly smaller. Before the actual I/O can be done by hardware like the DMA modules, some CPU cycles must be used to prepare the I/O and also to finish the I/O calls in the end [13].

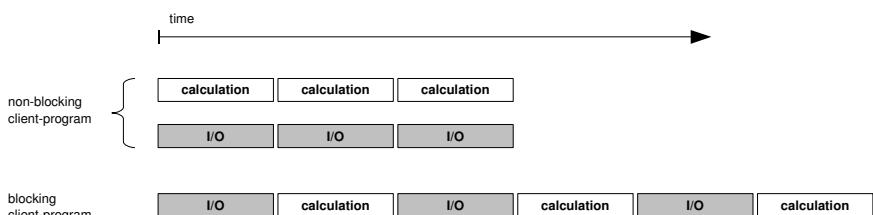


Fig. 1. Theoretical comparison of scenarios with equal times for I/O and calculation

In order to compare the individual results we introduce the **non-blocking efficiency ratio**

$$\frac{\text{time needed for non-blocking I/O version}}{\text{time needed for blocking I/O version}}. \quad (1)$$

Theory suggests that this number should always be greater or equal to 0.5. A number close to 0.5 indicates a good reduction in execution time, a number close to 1 indicates that no gains have been achieved and a number larger than 1 would imply that the execution takes longer when using non-blocking I/O operations.

Furthermore this theory does not say anything about the behavior of the file system that is being used. Assuming that in a high performance computing environment a parallel file system is used, one also has to look at its influence on the application behavior after a change from blocking to non-blocking I/O.

When the blocking version of a write operation returns the written data is available to everyone with access to the file system. It does not, however, mean that all the data has already been written to a hard disc drive. Modern parallel file systems like PVFS2 [5] have a server side cache for the data received by a writing application. This means a blocking I/O call can return whenever all the data has been received on the server side of the file system layer. If all the data fits into the cache the limiting factor of the I/O call is the throughput with which the data can be transferred to the file system. In case that the cache is full due to previous operations or too small for the amount of data being written the limiting factor no longer is the communication network but the speed with which data can actually be stored on the physical hard disc drives.

Assuming we have a program doing I/O and calculation iteratively. In the version using the non-blocking I/O calls one could have the situation where the calculation part and the I/O part take exactly the same amount of time when being run concurrently even with the file system cache being full. This situation is shown in Fig. 2. In the sequential counterpart (Fig. 3), the calculation sequences provide times in which the file system can empty its cache by flushing it to the hard disc drives. So while in our example the first I/O call still takes the same amount of time as in the parallel version the other I/O calls will return faster.

Thus, many factors have an influence on the behavior of an application. While looking at the theory helps us understand what will happen when switching from

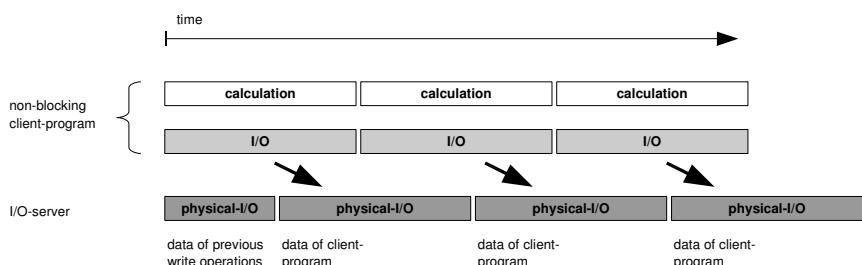


Fig. 2. Non-blocking I/O under assumption of equal times for I/O and calculation

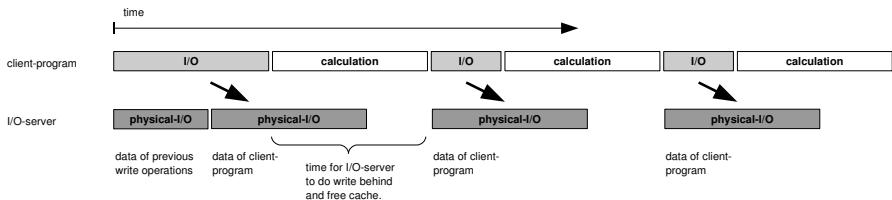


Fig. 3. Sequential execution of calculation and I/O

one method of I/O to the other, the complexity of the system is too big to be sure that the expectations match what really will be happening.

In the following section we present a benchmark which addresses the question of how much an application programmer can really gain by making use of non-blocking I/O and whether it is worth the effort to do so.

4 Design of the Benchmark

The first question when looking at benchmarking non-blocking I/O operations is which kind of functions do already exist that can be used for testing. The MPI-2 standard provides a non-blocking version of its I/O functions but allows them to internally call the blocking version. This is done in older versions of the MPI implementation MPICH2. The latest version at the time of creation of the benchmark was `mpich2-1.0.5p4`. Now non-blocking I/O operations can actually be done in a non-blocking way when using the parallel file system PVFS2 beginning with `mpich2-1.0.7`.

In order to measure the desired non-blocking version we implemented a benchmark and emulated the non-blocking functions `MPI_FILE_IWRITE(...)` and `MPI_WAIT(...)`. This was done by splitting the process into two threads inside the `MPI_FILE_IWRITE(...)` call and joining it inside the `MPI_WAIT(...)` call. The I/O thread then calls the blocking version of the MPI write operation which is now executed concurrently to the calculation thread.

Next we added functionality to fill the file systems I/O cache before the actual test runs. Finally we decided to use the following approach for the test scenarios.

As mentioned earlier we can expect the best reduction in execution time when the I/O part and the calculation take about the same amount of time both in the parallel and the sequential version. However, this scenario is probably not one that exists in reality due to file system behavior.

Choosing a sequential scenario in which both parts occupy the same amount of time includes time for the file system to empty its cache during the calculation part. What we were more interested in was the gain in performance for a situation where the non-blocking version does physical I/O the whole time. Then we compare this situation to the respective blocking version doing the same amount of operations in the calculation parts and writing the same amount of data in the I/O parts. Since the writing to hard disc drive will be the bottleneck in any

system this gives us a scenario which could not be executed faster in any way since the minimum of time needed to complete the task is at least the time needed to store the data. Fig. 2 shows this situation where both I/O and calculation last the whole time. Fig. 3 shows the theoretically expected sequential version of this scenario also suggesting that the I/O parts will get shorter due to the cache effects on the I/O server.

The tests, each run several times to ensure the reliability of the results, are conducted in the following way:

1. For a given amount of data to be written in each iteration it determines the calculations needed to be done to match the I/O time.
2. Fill the I/O server cache with enough data to make sure the following test will hit the hard discs all the time.
3. Run the non-blocking version measuring the time.
4. Delete the written file.
5. Fill the I/O server cache again to guarantee the same start environment for the blocking test run.
6. Run the blocking I/O version measuring the time.
7. Write the results including the measured total times, the average time for each I/O phase, the average time for each calculation phase and the test run parameters to an output file.

5 Discussion of Results

For the following results we ran the benchmark on the computer cluster of the research group **Parallel and Distributed Systems** which consist of 10 nodes containing the following hardware:

- Two Intel Xeon 2GHz CPUs
- Intel Server Board SE7500CW2
- 1GB DDR-RAM
- 1-Gbit/s-Ethernet-Interfaces

In addition the following hardware is provided:

- The nodes are connected through a D-Link DGS-1016 switch. The switch supports 1000 Mbit connections.
- 5 nodes with two 160 GB S-ATA HDDs set up as a RAID-0. These nodes were used for hosting the pvfs2-servers.

The MPI implementation we used was MPICH2 [3] in version mpich2-1.0.5p4. The parallel file system of choice was PVFS2 [5] version 2.6.2. In order to understand what was going on inside the I/O servers we created traces with the tools provided by the PIOviz environment [9] and generated the screen dumps Fig. 4 and Fig. 5 from the traces using Jumpshot [4]. All in all we ran a variety of test scenarios. The variables in the test setups were:

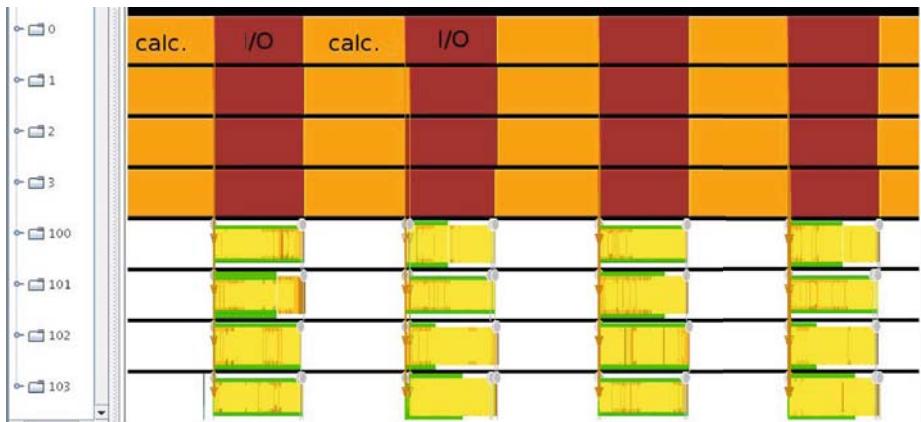


Fig. 4. PIOviz visualization of the benchmark for the blocking I/O phase using 4 I/O servers and 4 client processes

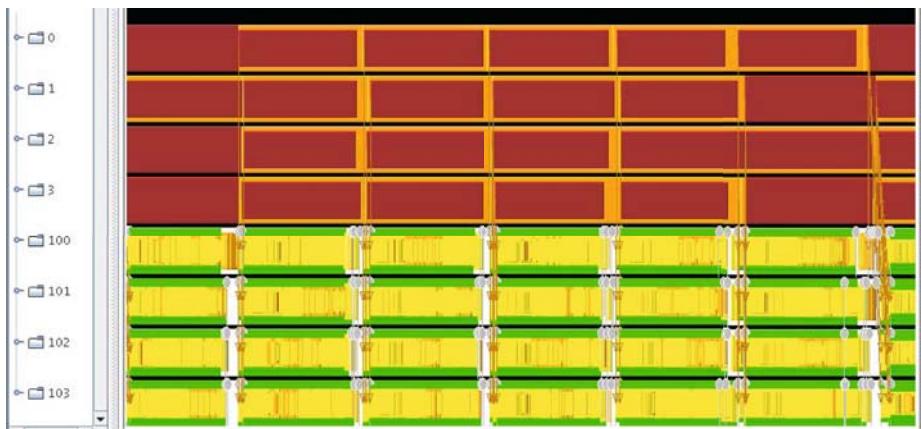


Fig. 5. PIOviz visualization of the benchmark for the non-blocking I/O phase using 4 I/O servers and 4 client processes

- number of CPUs per benchmark process
- number of benchmark processes
- number of I/O servers

For the more interesting tests providing only 1 CPU per benchmark process one of the processors in a node was disabled. The range of the number of benchmark processes and number of I/O servers was 1, 2 and 4 for each of them. All 9 combinations have been tested.

We will present the results of the test using 4 I/O servers and 4 benchmark processes each being run on an individual node in detail here. The other test results are presented in detail in [10].

The 2 CPUs per benchmark process tests show that in a scenario where each thread has its own CPU and the calculation part is not depending on free CPU cycles during the I/O part, we can actually get very close to the expected optimum with the **non-blocking efficiency ratio** (1) at 0.53.

All tests result in the **non-blocking efficiency ratio** being smaller than 0.75. This shows that no matter which setup we chose, at least 25% of the time needed to run the blocking version of the test can be saved by switching to non-blocking I/O operations.

In Fig. 4 and Fig. 5 you can see the traces created for the blocking and non-blocking version respectively. The lines labeled 0 to 3 represent the benchmark processes, the lines labeled 100 to 103 the pvfs2-server processes. In Fig. 4 one can nicely see the calculation phases in dark grey and the I/O phases in light grey. White spots on the pvfs2-server side indicate that no jobs are being executed in the server process.

In Fig. 5 one can see that the two parts really are being executed in parallel when using the emulated non-blocking functions of the benchmark. The pvfs2-server is busy the whole time during the non-blocking test. For the series of tests done for this scenario we can present the following results:

- The **non-blocking efficiency ratio** (1) lies in average at 0.6671. Indicating that over 30% of execution time is being saved by the use of non-blocking I/O.
- The calculation time lengthens by a factor of 1.25 when switching from blocking to non-blocking I/O calls.
- The I/O time lengthens by a factor of 1.29 when switching from blocking to non-blocking I/O calls.

The last two items indicate that while the overall execution time is reduced by using non-blocking I/O functions the individual tasks being performed take longer than in the sequential version.

6 Conclusion and Future Work

While non-blocking I/O has a restricted potential in reducing execution times of applications for parallel computing they can contribute to do so. We showed that in cases where the application programmer has a non-blocking I/O friendly program he can reduce the runtime by over 25%. Considering that running a supercomputer can be quite costly and waiting for results not very productive, the use of non-blocking I/O functions is something application programmer should consider to use.

So far we only looked at write operations and at a limited set of scenarios. In the future other scenarios which determine how much non-blocking read functions can help in the struggle of faster programs should be looked at. Also scenarios of either I/O or calculation intensive programs should be analysed in order to see if the trouble of implementing them with non-blocking functions leads to satisfying results. Finally the results we achieved with the benchmark

can also be used to compare implementations of non-blocking I/O functions in different MPI implementations. This can especially be done with the new version of MPICH2 where this functionality has been added.

References

1. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard (May 1994), <http://www mpi-forum.org>
2. Message Passing Interface Forum: MPI-2: Extensions to the Message-Passing Interface (June 1997), <http://www mpi-forum.org>
3. MPICH2 home page, <http://www-unix.mcs.anl.gov/mpi/mpich2/index.htm>
4. Zaki, O., Lusk, E., Gropp, W., Swider, D.: Toward Scalable Performance Visualization with Jumpshot. *Int. J. High Perform. Comput. Appl.* 13 (1999)
5. The Parallel Virtual File System – Version 2, <http://www.pvfs.org/pvfs2/>
6. The PVFS2 Development Team: PVFS2 Internal Documentation included in the source code package (2006)
7. Thakur, R., Gropp, W.D.: Test Suite for Evaluating Performance of MPI Implementations That Support MPI_THREAD_MULTIPLE. In: Cappello, F., Herault, T., Dongarra, J. (eds.) PVM/MPI 2007. LNCS, vol. 4757, pp. 46–55. Springer, Heidelberg (2007)
8. TOP500.org (April 2009), <http://top500.org/>
9. Kunkel, J.: Performance Analysis of the PVFS2 Persistency Layer, Bachelor's Thesis, Ruprecht-Karls-Universität Heidelberg, Germany (March 2006), <http://pvs.informatik.uni-heidelberg.de/Theeses/2006-kunkel-bsc.pdf>
10. Büttner, D.: Benchmarking of Non-Blocking Input/Output on Compute Clusters, Bachelor's Thesis, Ruprecht-Karls-Universität Heidelberg, Germany (April 2007), <http://www.ub.uni-heidelberg.de/archiv/9217/>
11. Latham, R., Miller, N., Ross, R., Carns, P.: A Next-Generation Parallel File System for Linux Clusters. *LinuxWorld Magazine*, 56–59
12. Gropp, W., Lusk, E., Sterling, T.: Beowulf Cluster Computing with Linux, 2nd edn. MIT Press, Cambridge (2003)
13. Stallings, W.: Betriebssysteme, 4th edn. Pearson Studium, London (2003)
14. Rabenseifner, R., Koniges, A.E.: Effective Communication and File-I/O Bandwidth Benchmarks. In: Cottronis, Y., Dongarra, J. (eds.) PVM/MPI 2001. LNCS, vol. 2131, p. 24. Springer, Heidelberg (2001)
15. Krietemayer, M., Versick, D., Tavangarian, D.: THE PRIOMark PARALLEL I/O-BENCHMARK. In: International Conference on Parallel and Distributed Computing and Networks, Innsbruck, Austria (February 2005)
16. Lawry, W., Wilson, C., Maccabe, A.B., Brightwell, R.: COMB: A Portable Benchmark Suite for Assessing MPI Overlap. In: Proceedings of the 4th IEEE International Conference on Cluster Computing (September 2002)

Conflict Detection Algorithm to Minimize Locking for MPI-IO Atomicity

Saba Sehrish¹, Jun Wang¹, and Rajeev Thakur²

¹ School of Electrical Engineering and Computer Science,
University of Central Florida, Orlando, FL 32816, USA

² Mathematics and Computer Science Division,
Argonne National Laboratory, Argonne, IL 60439, USA

Abstract. Many scientific applications require high-performance concurrent I/O accesses to a file by multiple processes. Those applications rely indirectly on atomic I/O capabilities in order to perform updates to structured datasets, such as those stored in HDF5 format files. Current support for atomicity in MPI-IO is provided by locking around the operations, imposing lock overhead in all situations, even though in many cases these operations are non-overlapping in the file. We propose to isolate non-overlapping accesses from overlapping ones in independent I/O cases, allowing the non-overlapping ones to proceed without imposing lock overhead. To enable this, we have implemented an efficient conflict detection algorithm in MPI-IO using MPI file views and datatypes. We show that our conflict detection scheme incurs minimal overhead on I/O operations, making it an effective mechanism for avoiding locks when they are not needed.

1 Introduction

Some of the scientific applications are I/O intensive and demand high-performance I/O access and bandwidth to store and retrieve their simulation results. A relatively slow procedure to store and retrieve the scientific application results to and from the storage system limits the overall performance despite the ever increasing compute power. Parallel file systems and high-level parallel I/O libraries are provided as a solution to mitigate this lag. Parallel file systems provide the semantics of the local file system but face additional challenges regarding atomicity and consistency. Some parallel file systems such as PVFS [6] and PVFS2 [3] do not support POSIX [4] semantics for atomicity and consistency. These semantics are guaranteed only if there is no data sharing among concurrent processes and are inadequate to describe complex requests in scientific computing applications that are non-contiguous in files. Scientific applications do perform these types of operations [11] [13], so alternative mechanisms for guaranteeing atomicity are necessary, e.g. a locking mechanism.

Atomicity semantics define the outcome of multiple concurrent I/O accesses, at least one of which is a write to a shared or overlapping region. With the advent of parallel I/O libraries, data can be accessed in various complex patterns by

multiple processes. These access patterns can be contiguous or non-contiguous, overlapping or non-overlapping depending upon the application behavior. Locking mechanisms provided by the file system are used to ensure that during a concurrent I/O access, shared data is not being violated. File locks, byte range locks, list locks, and datatype locks are various locking mechanisms that are available to guarantee the atomicity semantics. Adapted from the POSIX semantics, parallel file systems such as GPFS [16] and Lustre [2] provide a byte range locking mechanism. Byte range locks provide an option for guaranteeing atomicity of non-contiguous operations. By locking the entire region, changes can be made by using a read-modify-write sequence. However, this approach does not consider the actual non-contiguous access pattern that may occur in a byte range and introduces *false sharing*. This approach also limits the benefits of parallel I/O that can be gained, by unnecessarily serializing the accesses. To address these particular cases, [5] [10] propose to lock the exact non-contiguous regions within a byte range and maximize the concurrent I/O access.

The overhead of a locking mechanism appears in three forms; first is the communication overhead that is generated while acquiring and releasing the locks (sending the requests to lock server(s)), second is the storage space overhead that is caused by storing the locks during their acquire and release time (a data structure is maintained to store the locks, and for fine grained locks like list locks, this structure can grow very large.), and the third is computation overhead to assign new locks (i.e., the tree data structure is scanned to check if the same lock request is being held by a different process). These overheads can be reduced by making sure that unnecessary locking requests are not generated. An observation is that the locks are requested (e.g. file locks, byte-range locks, list locks, and datatype locks) even if there are no overlaps, when locks are not needed. This observation leads us to a very important question, whether it is possible to isolate the cases where atomicity is required and where it is not, and optimize the locking mechanism.

We propose a scheme to identify the conflicts at the application level, by identifying the concurrent access patterns and overlaps within an application. Our conflict detection algorithm, implemented at the MPI-IO level for independent I/O operations uses file views and datatype decoding to determine the overlaps. Our results show that the conflict detection algorithm incurs a minimal overhead and performs within 3.6% of the ideal case (i.e. concurrent access with no locks). When the file view for a process does not overlap with the file views of other processes, locking is not required; there will be no conflicts. Our approach reduces the locking overhead at minimum for the non-overlapping regions but handles the overlapped regions using either file or byte range locks.

2 Design

We propose a conflict detection algorithm that should be performed before any locking mechanism. Our goal is to optimize the lock acquiring process by providing an efficient conflict detection algorithm beforehand to identify the

overlapping regions, thereby requesting the locks only if there are overlapping regions. The conflict detection algorithm presented in this paper is based on MPI datatypes and file views. Typically, a file read/write request in any MPI-IO program consists of following steps: 1) Create the Data types, 2) Create the File views, and 3) Read/Write Request. The conflict detection is performed when a file view is created (`MPI_File_set_view`). Since it is a collective call, each process can exchange their file views and determine the overlapping regions by comparing offset/block length pairs. Each node acts as a conflict detector for itself.

The file view is created using `MPI_File_set_view`, and then each node decodes the supplied MPI datatype. Decoding a datatype is not straight forward and is a two step procedure using MPI-IO functions. The first step is getting the envelope of the datatype using `MPI_Type_get_envelope` which returns information such as number of displacements and block lengths used to create the datatype. The second step is getting the actual contents in the form of offset/block length using `MPI_Type_get_contents`. The decoded datatypes are exchanged using collective communication functions. The overhead of conflict detection is based on the complexity and size of the datatype. For some datatypes this overhead is as small as exchanging two long integer values, while for others it can consist of a long list of offset/length pairs.

2.1 Conflict Detection

We categorize the derived datatypes in to two broader categories based on their structure. The first category is a *regular datatype*; all the processes have the same block size but a different displacement e.g. `MPI_Type_vector`, `MPI_Type_subarray`. In a subarray, each process accesses a subarray that is defined by the number of dimensions and the starting and ending offsets in each dimension. For a regular datatype, we need to exchange the start and end offset in each dimension for each process. The second category is *irregular datatype*; all the processes may access different block sizes, different patterns and different displacements, e.g. `MPI_Type_hindexed`, where each process accesses a non-contiguous region defined by a list of offsets and corresponding block lengths.

Regular Datatypes: In independent write operations, when there are overlaps among different writes, only one process should perform the write at a time. For regular datatypes, we take the example of `MPI_Type_subarray`. A subarray datatype is defined by the number of dimensions, size of array in each dimension, sizes of subarrays in each dimension and the start positions for each subarray. The start of each subarray and the size in each dimension is used to identify the overlaps between any two consecutive tiles or subarrays as shown in the following equations. A conflicting region is specified by $CR(CO, CL)$, where CO is the conflicting offset and CL is the conflicting length. The displacement of the i^{th} process is specified by $disp_i(x, y)$ for a two-dimensional subarray and the corresponding block length is given by $blklen_i(x, y)$.

$$CO = \max(disp_i(x, y), disp_j(x, y)) \quad (1)$$

If both the displacements i.e. $disp_i$ and $disp_j$ are the same, the CL is given by eq. 2, otherwise eq. 3 is used.

$$CL = \min(blklen_i(x, y), blklen_j(x, y)) \quad (2)$$

$$CL = [\min(disp_i(x, y) + blklen_i(x, y), disp_j(x, y) + blklen_j(x, y))] - CO \quad (3)$$

Irregular Data types: For irregular datatypes, the offset/length pairs are required because each non-contiguous region will be of a different size. Each process compares its own offset/length list against the others. A conflicting region $CR(CO, CL)$ is defined by the following equations, where CO is the starting offset, and CL is the length of the conflicting region. $disp(i)$ is the displacement of the i^{th} process and $blklen(i)$ is the corresponding block length.

$$CO = \max(disp(i), disp(j)) \quad (4)$$

If both the displacements i.e. $disp(i)$ and $disp(j)$ are the same, the CL is given by eq. 5, otherwise eq 6 is used.

$$CL = \min(blklen(i), blklen(j)) \quad (5)$$

$$CL = [\min(disp(i) + blklen(i), disp(j) + blklen(j))] - CO \quad (6)$$

Since, the exact displacement and block length values are used, *false sharing* is eliminated completely. There are more complex datatypes that we categorize as multi-level datatypes, and MPI facilitates the creation of nested datatypes. For example in *noncontig* benchmark, `MPI_Type_contig`, `MPI_Type_vector` and `MPI_Type_struct` are used to create file views. In such cases, we perform multi-level decoding to determine the conflicts. The overhead incurred by conflict detection is evaluated in Section 4 in terms of communication and computation time. The communication overhead is determined by the collective communication calls to exchange datatypes. The computation overhead includes the time to generate and compare the datatypes. It depends on the datatype or the size of the list to be compared. For each process, if the size of the list is N , it will perform a linear compare of order N . The space required is equal to the size of the datatype or the offset/length list.

3 Implementation

We implement the self-detecting locking mechanism in ROMIO, by adding it to `MPI_File_set_view` as shown in the listing 1. Listing 1 also shows how to use the conflict variable in the main program. The decoding process utilizes two function

from MPI-IO library; `MPI_Type_get_envelope` and `MPI_Type_get_contents`. Our initial implementation provides conflict detection support for a few selected data types, `MPI_Type_vector`, `MPI_Type_subarray`, `MPI_Type_hindexed`. Each process exchanges the view information using the `MPI_Allgather` collective communication function. Once the data is ready at each node, it performs the comparison for the conflicts. Our current implementation is tested with PVFS2, which does not support locking. We have used the algorithms presented in [15] for file locks and [17] [14] for byte range locks implementations with PVFS2. For byte range locks, we determine the start and end offsets of the byte range accessed by each process using an existing function in ROMIO i.e. `ADIOI_Calc_my_offset`; it returns the start and end offsets used by `BR_Lock_acquire(br_lock, ...)`.

MPI-IO write functions can be performed collectively or independently. The collective write operations do not require conflict detection because conflicts cannot occur in collective operation. The independent write operations do not communicate with each other to optimize the non-contiguous access and require locking to protect the shared data regions. The blocking independent write functions are `MPI_File_write`, `MPI_File_write_at`, and `MPI_File_write_shared`. Our conflict detection implementation can be used with any locking mechanism and independent write function. et.

4 Performance

In this section, we evaluate our conflict detection algorithm. We quantify the overhead of the conflict detection algorithm, and compare the I/O time of various benchmarks. We evaluate the conflict detection using three different benchmarks for both overlaps and no-overlaps in file access patterns. We compare the time to determine conflicts combined with and without locks and also show the overhead of conflict detection in terms of the communication and computation time. In our experiments, we use the best case of a concurrent write access i.e. when there are no locks, and all processes can perform a write operation concurrently. The worst case used in the experiments is the whole file locks [15], when all writes by different processes become serial. Additionally, we have also used the byte range locks as implemented in [17] [14].

The experimental setup consists of a 16 node cluster, with PVFS2. Each node is a Dell PowerEdge 2 CPU, dual core with 4GB memory and two 500 GB SAS hard drives. PVFS2 has been setup on all 16 nodes, so all nodes serve as I/O nodes and the compute nodes. The network connection is Intel Pro/1000 NIC, and the cluster network consist of Nortel BayStack 5510-48T GigaBit switch. We have used PVFS2 version 2.7.0 and MPICH2-1.0.7 in our experiments.

MPI-Tile IO: MPI-Tile IO [1] is used to write non-overlapping and overlapping tiles.

Non-overlapping Access: The number of dimensions for `MPI_Type_subarray` is 2 in MPI-Tile-IO, and the array of starts gives the x and y position for each tile. The array of sub-sizes returns the size of each tile in both dimensions. The problem size

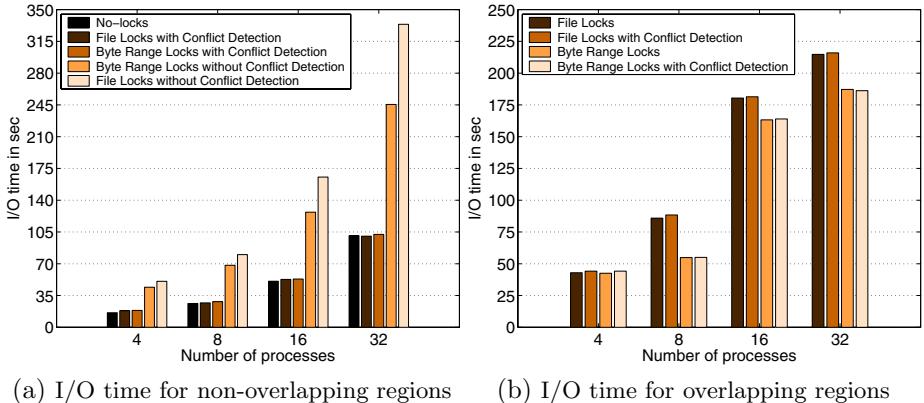


Fig. 1. MPI-Tile-IO: Comparing I/O time for non-overlapping and overlapping regions with conflict detection and locks

consist of an array of 4096×8192 per process, where each element size is 32 bytes. Each process writes 1GB of data, hence if there are 32 processes then the total amount of data written will be 32GB. The 4, 8, 16 and 32 processes are arranged in 2x2, 4x2, 8x2 and 16x2 panels. The I/O time results are shown in Figure 1. I/O time includes the time for `MPI_File_set_view` (also the time for conflict detection), `MPI_File_write` and waiting time if there are file locks or byte range locks.

There are five bars, the first bar shows the best case of no-locks, whereas the last bar shows the worst case. The second and third bar shows the case when conflict detection is performed, no conflicts are reported and the underlying file and byte range locking is disabled as a result. The fourth bar performs byte range locks without using conflict detection. We can see that file and byte range locks combined with conflict detection performs close to the no-locks, i.e. an ideal case for the non-overlapping I/O accesses. The overhead of conflict detection is minimal, because the datatypes exchanged in MPI-Tile-IO consist of start and end offset of each tile accessed by a process. This overhead increases with the number of processes, and the detailed overhead results are shown in Figure 3. File locks have the worst performance because they introduce sequential access and the I/O time increases with the increase in number of processes.

Overlapping Access: We used *overlap-x* and *overlap-y* options in MPI-Tile-IO to generate the overlapped I/O access patterns. The problem size consists of an array of 4096×8192 per process with an overlap of 512×1024 in x and y direction respectively, where each element size is 32 bytes. Each process writes 1GB of data with an overlapping data of 16MB per process. The atomicity semantics guarantee that the 16MB overlapping data will be defined by one process at a time and not contain any data from more than one processes. We compare the I/O time for file locks and byte range locks with and without conflict detection. No-locks results are not provided here, because the output in overlapping region is not defined without locks.

I/O time for the other four cases is shown in the Figure 1. It should be noted that since there are overlaps, locks cannot be avoided, and the purpose of these results is to demonstrate that if a conflict detection is performed before any locking mechanism, the overhead is not significant. The major contribution of our work is for the cases when there are no overlaps and locking is eliminated completely. These results also show that conflict detection can be combined with any other locking mechanism.

S3asim: S3asim is a sequence similarity search algorithm simulator [7], that uses a variety of parameters to adjust the total fragments in the database, sequence size, query count, etc. After each worker finishes its query processing of its fragment, it sends its ordered scores to the master process. The master process merges the ordered scores to its list and once all fragments of an input query have been processed, it sends the locations in the aggregate file to each worker to write the results. Finally, each worker writes the result data to the output file independently when it receives the location from master. The datatype used by workers is `MPI_Type_hindexed`, and is defined by an array of block lengths and displacements.

Our conflict detection algorithm returns no conflict for the S3asim benchmark, and this result is in accordance with [7]. All the workers work on different segments of the database for query search, and a few of them write the results to a shared file. Figure 2 shows three cases; no-locks i.e. none of the locking mechanism was applied, conflict detection i.e. conflict detection algorithm was run in order to determine overlaps, and finally file locks. The conflict detection is performed only for the processes that actually write, and on average it performs within 3.6% of the no-locks case. The file locks show increase in I/O time with increase in number of processes, since there are fewer writers as compared to the number of processes, and the line is not a steep curve.

Conflict Detection Overhead: In this section, we present the overhead incurred by our conflict detection algorithm. Each process participates in two collective communication calls to gather the file view and the starting offset. We measure the communication overhead as the time spent in communicating the required information. This overhead depends on the datatypes and the number of processes that actually perform the write operation. In Figure 3, we show the communication overhead in different benchmarks. It is noticed that in tile-io, each process writes a tile/subarray that may or may not have overlapping regions, but in S3asim only a few workers that find the match perform the write

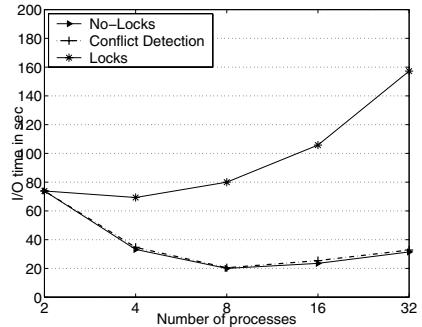


Fig. 2. S3asim: Comparing I/O time when there are no-locks, conflict detection with file locks, and file locks

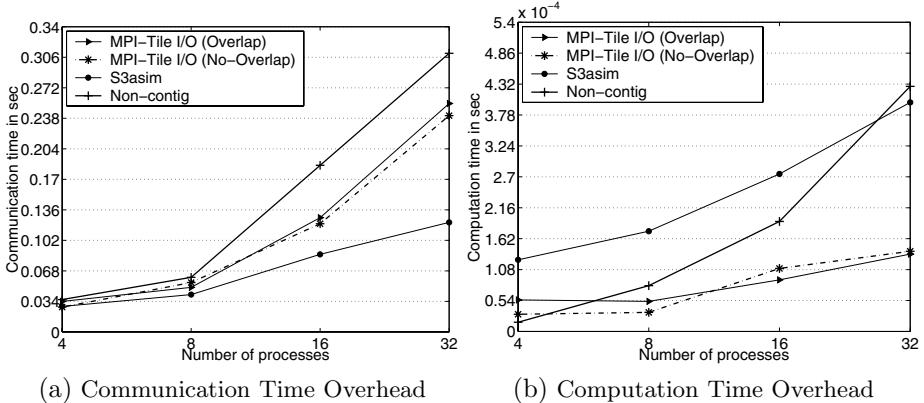


Fig. 3. Communication and Computation Overhead of Conflict Detection for different Benchmarks

operation. This explains the less steep curve for S3asim as compared with tile-io. Non-contig shows the maximum overhead because it has a multi-level datatype and, we need two collective calls to communicate the two levels of datatypes.

After communicating the file views, each process computes the conflicts, which are the results of the comparisons of its own file view with the received ones. The time spent in computing the conflicts is quantified as the computation overhead. This overhead depends on the datatypes and also the size of offset/length pairs generated from the file views. In Figure 3, we also show the computation overhead in various benchmarks. It can be seen that the overhead is minimal in tile-io, the reason being that tile-io writes data logically in sub-arrays, and in order to perform the conflict detection we do not generate the offset/length pairs and use the start and end offsets in each dimension to detect conflicts. For S3asim, the overhead is also minimal because the actual number of workers writing the results is less than the number of processes performing the search. For example, in a $32p$ run, for certain queries only 4 or 5 processes write in the end. S3asim has a few writers but its datatype is more complex and with the increase in number of processes it has more writers, so S3asim has greater overhead with increased number of processes. Noncontiguous benchmark performs four different access patterns, i.e. contiguous/non-contiguous in memory and contiguous/non-contiguous in file. We only present the results when accesses are either non-contiguous in memory or in file. A file size of 2GB is used but we keep per process file size constant, the vector length i.e. the number of elements in vector datatype used is set to 32 and element count i.e. the number of elements in a contiguous chunk to 128. The first derived datatype is `MPI_Type_vector`, and the second derived datatype that is comprised of `MPI_Type_vector` is `MPI_Type_struct`. The overhead of conflict detection for non-contig is shown in Figure 3. Non-contig has a steady increase in computation overhead with number of processes, it is a case of multi-level datatype.

Number of Lock Requests: We emphasize that with conflict detection, if there are no overlaps in the application access pattern, locking can be avoided. Otherwise, only the overlapped region should be locked to guarantee the atomicity of concurrent operations. Finally, we investigate the utilization of our scheme with the existing locking mechanisms. Assuming that there are three locking implementations, i.e. whole-file, byte range and list locks, we compare the number of locks per client in each case with and without conflict detection.

There is one lock per client for the file locks and the byte range locks, but these are coarse grained locks. The non-contiguous access patterns observe false sharing with coarse grained locks. We want to show that a locking mechanism combined with our

approach is effective in reducing the number of lock requests that will be issued for any lock server that needs communication and space on the server to be stored. Many scientific applications have patterns that would require hundred thousands of locks [10]. In Table 1, we show a simple comparison of the number of locks (whole file, byte-range and list) with and without using conflict detection. It should be noticed that the list locks and the datatype locks are very fine grained locks. The locks acquiring process is instigated by a client. The client first calculates which servers to access for the locks; the saving from conflict detection will come in the form of either none or a fewer number of requests. The implementation of conflict detection with list locks [5] is left for future work.

Table 1. Reduction in number of locks

Approach	Number of Locks/Client
Whole-File Locks	1
Byte-Range Locks	1
List Locks	64
Locks with Conflict detection	<i>No-Overlaps: 0 Overlaps: 1 (Whole-file), 1 (Byte-Range), Number of CR (List)</i>

5 Related Work

Researchers have contributed to provide atomicity semantics both at application and file system level. Non-contiguous access patterns and overlapping I/O patterns [8] [9] [12] have been widely studied and the customized locking schemes, process rank ordering and handshaking have been proposed. List locks and datatype locks [5] [10] have maximum concurrency, but they acquire and maintain locks for all regions accessed by a process. We provide conflict detection to find the overlaps before lock requests are issued. The conflict check facilitates the locking mechanism by providing a decision where locks are necessary to guarantee atomicity.

6 Conclusion

We have proposed a scheme to perform conflict detection using file views, and introduce lock free independent write operations if there are no conflicts. We

have implemented our algorithm in ROMIO. In MPI-IO applications atomicity guarantees rely on the file system locks. Our Conflict detection algorithm is able to extract overlapping regions from the file views (for independent operations) created by MPI-IO application with a minimal overhead. It paves the way to the lock-free and scalable approaches of MPI-IO atomicity support.

Listing 1. Pseudocode for Conflict Detection

```
//Pseudocode for Conflict Detection
int Conflict_Detection(MPI_Datatype ftype) {
    //Get the datatype envelope,
    MPI_Type_get_envelope(ftype, &num_ints, &num_adds, &num_dtotypes, &combiner);

    //Get the actual contents of the datatype
    MPI_Type_get_contents(ftype, num_ints, num_adds, num_dtotypes,
                          array_of_ints, array_of_adds, array_of_dtotypes);

    //Gather datatypes from all other nodes
    MPI_Allgather(array_of_ints, num_ints, MPI_INT, ai_all,
                  num_ints, MPI_INT, MPI_COMM_WORLD);
    ...

    //Compare datatypes
    switch(combiner){
        case MPI_COMBINER_SUBARRAY:
        // Compare the elements of ai_all, that contains array of
        starts, and the block lengths are same for all blocks!
        break;
        case MPI_COMBINER_INDEXED:
        case MPI_COMBINER_HINDEXED:
        break;
        ...
    }
    return conflict;
}
```

Acknowledgments

We would like to thank Rob Ross of Argonne National Laboratory for his guidance and valuable comments on this paper. This work is supported in part by the US National Science Foundation under grants CNS-0646910, CNS-0646911, CCF-0621526, CCF-0811413, and the US Department of Energy Early Career Principal Investigator Award DE-FG02-07ER25747.

References

1. <http://www.cs.dartmouth.edu/pario/examples.html>
2. Lustre filesystem, <http://www.lustre.org/>
3. Parallel virtual file system version 2, <http://www.pvfs.org/>
4. IEEE/ANSI std.1003.1. portable operating system interface (POSIX)-part 1: System application program interface (API)[C language] (1996)
5. Aarestad, P.M., Ching, A., Thiruvathukal, G.K., Choudhary, A.N.: Scalable approaches for supporting MPI-IO atomicity. In: CCGRID 2006: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid, Washington, DC, USA, pp. 35–42. IEEE Computer Society Press, Los Alamitos (2006)

6. Carns, P.H., Ligon III, W.B., Ross, R.B., Thakur, R.: PVFS: a parallel file system for linux clusters. In: ALS 2000: Proceedings of the 4th annual Linux Showcase & Conference, Berkeley, CA, USA, p. 28. USENIX Association (2000)
7. Ching, A., Feng, W., Lin, H., Ma, X., Choudhary, A.: Exploring I/O strategies for parallel sequence-search tools with s3asim. *hpdc*, 229–240 (2006)
8. Ching, A., Choudhary, A., Coloma, K., Liao, W.k., Ross, R., Gropp, W.: Noncontiguous I/O accesses through MPI-IO. In: CCGRID 2003: Proceedings of the 3rd International Symposium on Cluster Computing and the Grid, Washington, DC, USA, p. 104. IEEE Computer Society Press, Los Alamitos (2003)
9. Ching, A., Choudhary, A., Liao, W., Ross, R., Gropp, W.: Noncontiguous I/O through PVFS. In: CLUSTER 2002: Proceedings of the IEEE International Conference on Cluster Computing, Washington, DC, USA, p. 405. IEEE Computer Society Press, Los Alamitos (2002)
10. Ching, A., Liao, W.k., Choudhary, A., Ross, R., Ward, L.: Noncontiguous locking techniques for parallel file systems. In: SC 2007: Proceedings of the ACM/IEEE conference on Supercomputing, pp. 1–12. ACM, New York (2007)
11. Crandall, P.E., Aydt, R.A., Chien, A.A., Reed, D.A.: Input/output characteristics of scalable parallel applications. In: Proceedings of Supercomputing 1995. Press (1995)
12. Liao, W.k., Choudhary, A., Coloma, K., Thiruvathukal, G.K., Ward, L., Russell, E., Pundit, N.: Scalable implementations of MPI atomicity for concurrent overlapping I/O. In: International Conference on Parallel Processing, p. 239 (2003)
13. Nieuwejaar, N., Kotz, D., Purakayastha, A., Ellis, C.S., Best, M.L.: File-access characteristics of parallel scientific workloads. *IEEE Trans. Parallel Distrib. Syst.* 7(10), 1075–1089 (1996)
14. Pervez, S., Gopalakrishnan, G.C., Kirby, R.M., Thakur, R., Gropp, W.D.: Formal verification of programs that use MPI one-sided communication. In: Mohr, B., Träff, J.L., Worringen, J., Dongarra, J. (eds.) PVM/MPI 2006. LNCS, vol. 4192, pp. 30–39. Springer, Heidelberg (2006)
15. Ross, R.B., Latham, R., Gropp, W., Thakur, R., Toonen, B.R.: Implementing MPI-IO atomic mode without file system support. In: CCGRID, pp. 1135–1142 (2005)
16. Schmuck, F., Haskin, R.: GPFS: A shared-disk file system for large computing clusters. In: FAST 2002: Proceedings of the 1st USENIX Conference on File and Storage Technologies, Berkeley, CA, USA, vol. 19. USENIX Association (2002)
17. Thakur, R., Ross, R., Latham, R.: Implementing byte-range locks using MPI one-sided communication. In: Di Martino, B., Kranzlmüller, D., Dongarra, J. (eds.) EuroPVM/MPI 2005. LNCS, vol. 3666, pp. 119–128. Springer, Heidelberg (2005)

Exploiting Efficient Transpacking for One-Sided Communication and MPI-IO

Faisal Ghias Mir and Jesper Larsson Träff

NEC Laboratories Europe, NEC Europe Ltd.
Rathausallee 10, D-53757 Sankt Augustin, Germany
`{mir,traff}@it.neclab.eu`

Abstract. Based on a construction of socalled *input-output* datatypes that define a mapping between non-consecutive input and output buffers, we outline an efficient method for copying of structured data. We term this operation *transpacking*, and show how transpacking can be applied for the MPI implementation of one-sided communication and MPI-IO. For one-sided communication via shared-memory, we demonstrate the expected performance improvements by up to a factor of two. For individual MPI-IO, the time to read or write from file dominates the overall time, but even here efficient transpacking can in some scenarios reduce file I/O time considerably. The reported results have been achieved on a single NEC SX-8 vector node.

1 Introduction

In [5] we described a new solution for MPI for the socalled *typed copy problem*: given a non-consecutive input buffer and a non-consecutive output buffer each described by a possibly different derived MPI datatype with the same type signature (in particular, with the same number of basic elements), copy the basic elements from their type map positions of the input buffer to their type map positions of the output buffer. The new approach consists in constructing a special, derived data type, termed an *input-output type*, describing compactly the mapping from input buffer elements to output buffer elements, and using this type as guiding type for a corresponding *transpacking* function that takes care of the actual copying. In [5] a kind of greedy algorithm for constructing input-output types for derived datatypes as defined in MPI was developed. Isolated benchmarking on an NEC SX-8 showed that transpacking can indeed lead to the expected performance improvement of a factor of two over a trivial solution to the typed copy problem in which the input buffer is packed into a consecutive intermediate buffer and unpacked from this into the output buffer.

The transpacking operation has obvious applications inside an MPI implementation, namely for cases where a process has to copy differently typed data to itself (this is called for as part of many collective operations, e.g., `MPI_Alltoall`), in one-sided communication [6, Chapter 11], especially between processes that can access a common shared memory (for instance, by residing on the same SMP node), and in MPI-IO [6, Chapter 13].

In this paper we describe the requirements to a full-fledged, MPI library internal transpacking functionality in more detail, focusing on the details of the interface, such that it can be applied in the above mentioned cases. The functionality has been fully implemented in this fashion, such that we can investigate the possible performance improvements that the MPI user may see. For this we consider one-sided communication and MPI-IO. Benchmarking has been done on an NEC SX-8 node. For the one-sided communication case we report the expected performance gains of about a factor of two for a set of selected datatypes. For MPI-IO the case is less clear, since the total time is ultimately dominated by the actual I/O time. The savings by the efficient typed copy operation are therefore much less spectacular. We nevertheless demonstrate gains of more than 10% in some realistic cases.

We furthermore believe that the basic ideas behind the transpacking functionality as described here will, possibly with some algorithm engineering, be applicable to other systems as well. We also believe that transpacking has applications outside of MPI, and has been investigated in other contexts. For instance, in [4] a different representation of non-consecutive data is employed (FALLS and PITFALLS), and mapping functions on non-consecutive and consecutive layouts are developed.

2 Specification and Implementation of Transpacking

The basic input-output type construction was described in [5]. Given an input MPI type T^i and an output MPI type T_o with the same signature (same basic elements in the same order, see [6, Chapter 4]), the function $\text{IOTYPE}(T^i, T_o)$ returns an *input-output type* T_o^i that directly maps elements of the type map of T^i to elements of the type map of T_o . We note that T_o^i can actually be used to map in both directions, from input to output type and *vice versa*, formally that $T_0^i = \text{IOTYPE}(T^i, T_o)$ and $T_i^o = \text{IOTYPE}(T_o, T^i)$ are structurally equivalent, and write $T_i^o = \bar{T}_o^i$. Given the two types T^i and T_o it suffices to run the input-output type construction only once to obtain maps T_o^i and T_i^o for copying in both directions.

Transpacking itself is done by a special typed copy function $\text{TYPEDCOPY}(\text{src}, \text{srcoff}, \text{srcext}, \text{dst}, \text{dstoff}, \text{dstext}, c, T_o^i)$. This function takes source and destination addresses from each of which a segment starting at a given offset of a given extent is copied. A count of c units of the input-output type T_o^i is copied but only the data falling in the designated segments. This is shown in Figure 1. The function returns the number of (black) basic elements actually copied, that is the number elements satisfying the conditions of being in both of the specified input and output segments. With the typed copy function the MPI one-sided put and get communication operations for `origin` and `target` buffers in shared memory buffers can be implemented by

$$\text{TYPEDCOPY}(\text{origin}, 0, c^o \cdot \text{extent}(T^o), \text{target}, 0, c_t \cdot \text{extent}(T_t), c_t^o \cdot T_t^o)$$

and

$$\text{TYPEDCOPY}(\text{target}, 0, c_t \cdot \text{extent}(T_t), \text{origin}, 0, c^o \cdot \text{extent}(T^o), c_t^o \cdot \bar{T}_t^o)$$

respectively, and only one typed copy function is needed.

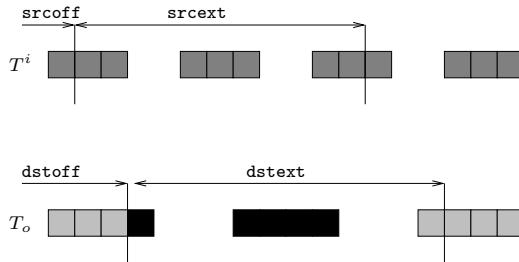


Fig. 1. Effects of a `TYPEDCOPY(src, srcoff, srcext, dst, dstoff, dstext, 1, T_o^i)` call. The elements that fulfill both input and output conditions of lying in the designated segments, and actually copied are shown in black.

The `TYPEDCOPY` function can be efficiently implemented by flattening on the fly techniques [10], and similar [2,7,3,13,8] to limit the number of recursive traversals of T_o^i , enable copy of large sections of evenly strided blocks, improve cache-behavior etc.

This functionality is obviously sufficient for implementing one-sided communication operations, in which a fully specified MPI origin buffer is copied to/from a fully specified target buffer. Use of transpacking in MPI-IO poses new problems that were not discussed in [5].

2.1 Extended Input-Output Type Construction

To motivate the required functionality we consider the *data sieving* technique [9] often used in the implementation of individual, non-collective MPI-IO operations on non-consecutive data. This is illustrated in Figure 2, where an individual MPI-IO read operation is considered. A data sieving implementation allocates an intermediate buffer for transfer of non-consecutive file blocks to and from memory. The file is read/written blockwise, and in each iteration of the loop data are copied between this intermediate buffer and the (non-consecutive) user buffer.

The file access can start at any (non-negative) *offset* in the current file view, where the offset is a multiple of the *elementary type* upon which the *filetype* is constructed. Because of this an input-output type constructed from filetype T^i and memory type T_o as described above [5] does not describe the copying that has to be done from the intermediate file buffer to the user-space memory buffer. The first element to be copied (and the first block in the intermediate buffer) is *not* the first element of T^i but the element found after skipping some offset of elementary types. Starting from here elements are copied to the memory buffer starting from the *first* position of T_o .

To cater for this, more flexibility in the input-output type construction is needed. More specifically, the *extended input-output type construction* function `IOTYPE($s^i, c^i, T^i, s_o, c_o, T_o$)` returns an input-output type T_o^i that maps c^i instances of T^i after skipping the first s^i bytes of T^i onto c_o instances of T_o after

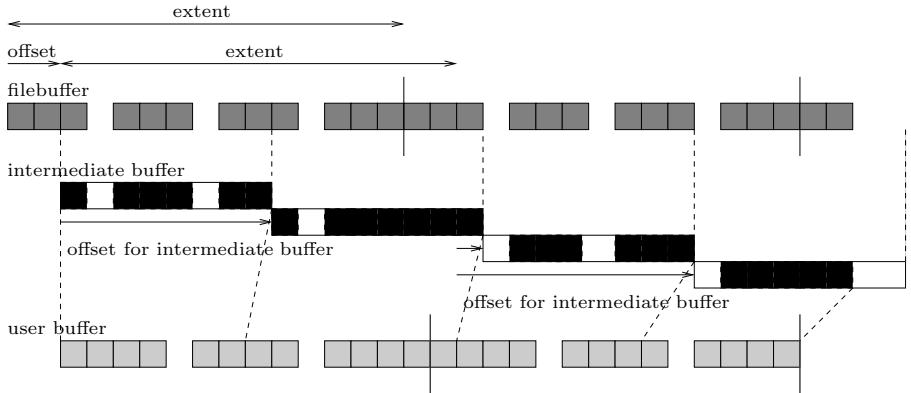


Fig. 2. The basic MPI-IO loop for an individual `MPI_File_read` call of 2 user datatypes starting at offset 2 (elementary types) of the file. File (input) and user (output) type consist of 12 elementary type elements, and the intermediate buffer has room for 8 elements.

skipping the first s_o bytes of T_o . The extended functionality is illustrated in Figure 3.

The basic pre-condition on the call is that $c^i \cdot \text{size}(T^i) = c_o \cdot \text{size}(T_o)$, or more precisely that c^i instances of T^i has the same type signature as c_o instances of T_o . A basic observation for the implementation is that the extent of the types is not affected by the skips, that is $\text{extent}(c^i \cdot T^i) = \text{extent}_{s^i}(c^i \cdot T^i)$. This observation is illustrated in Figure 2. As can be seen from Figure 3 cases a)-c), different input skips give rise to quite different input-output type structures. The structure of case d) is the same as case a), and the structure of case e) the same as case b). This observation shows that it actually suffices to be able to construct input-output types with just one (say) input skip; the output skip can always be set to zero, and the cases where both input and output skip is needed handled by the offsets given in the TYPEDCOPY-function. More concretely, the case e) can be handled by constructing the the type for case b), and copying with an offset of one basic element. The implementation of the extended functionality can be done along the lines described in [5], but has a somewhat more complicated recursive structure. It is worth noting that with $s^i = 0$ and $s_o = 0$ the extended construction coincides with the original algorithm of [5].

We can now use the extended type copy functionality for implementation of individual file read and write calls. For instance, the basic file read loop for implementing the `MPI_File_read(fh, offset, buf, co, To, status)` call would look as follows:

```
// file: file being read (from file handle fh), buf: user buffer
// offset: offset in file (elementary type units)
// intbuf, intext: intermediate buffer with extent
//  $T^i$  is filetype,  $T_o$  user type,  $T_o^i$  the input-output type
```

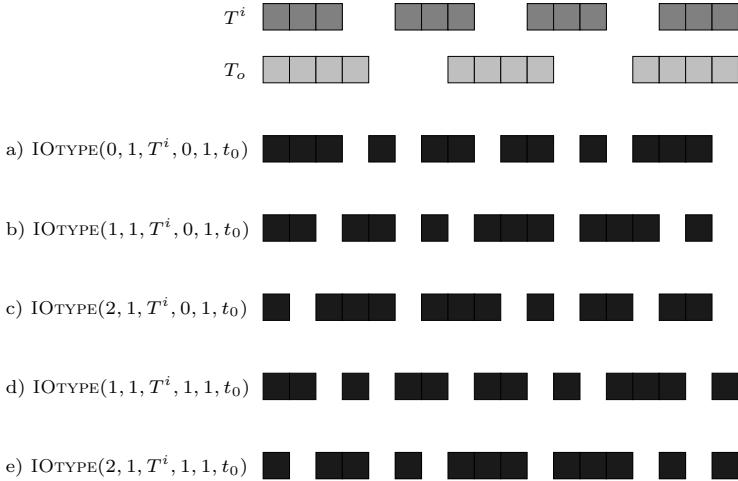


Fig. 3. The extended input-output type construction. Input and output types T^i and T_o are vectors with 4 respectively 3 blocks of 3 respectively 4 basic elements. The figure illustrates the structure of input-output types with different skips. Five input-output types are shown: a) $s^i = 0, s_o = 0$, b) $s^i = 1, s_o = 0$, c) $s^i = 2, s_o = 0$, d) $s^i = 1, s_o = 1$, e) $s^i = 2, s_o = 1$.

```
// For simplicity it is assumed that size( $T^i$ ) = size( $T_o$ )
intoff, bufoff, bufext ← 0, 0, extent
while bufext > 0
    READFILE(file, offset, intbuf, inttext)
     $c \leftarrow \lceil \frac{\text{intoff} + \text{intext}}{\text{extent}(T^i)} \rceil$ 
    TYPEDCOPY(intbuf - intoff, inoff, intext,
              buf + bufoff, 0, bufext, c,  $T_o$ )
     $c \leftarrow \lfloor \frac{\text{intoff} + \text{intext}}{\text{extent}(T^i)} \rfloor$ 
    intoff ← (intoff + intext) mod ( $c \cdot \text{extent}(T^i)$ )
    bufoff ← bufoff +  $c \cdot \text{extent}(T_o)$ 
    bufext ← bufext - bufoff // remaining extent of user buffer
    offset ← offset + intext
endwhile
```

2.2 Type Caching

Input-output type construction can be somewhat costly, although as argued in [5] no worse than $O(|T_o|)$ (which is normally small, although in bad cases the input-output type can blow up and become significantly larger than either input or output type). Thus, transpacking hardly makes sense for very small data instances. Type construction time can be amortized over several typed copy operations by caching the constructed input-output types.

We have implemented a caching mechanism by associating a small cache with each committed MPI datatype. The first typed copy operation on a quadruple

(s^i, T^i, s_o, T_o) constructs T_o^i (and \bar{T}_o^i) with skips s^i and s_o and caches a pointer to this type with both T^i and T_o . Subsequent lookup in the (small) cache of T^i is done by linear search for a triple (T_o, s^i, s_o) . Should the cache for a type become full, the least recently used entry is evicted, and the space for T_o^i freed.

3 Experimental Evaluation

We have implemented the input-output type construction and the transpacking function, and used it for improved implementations of one-sided communication between MPI processes on the same SMP node, and for non-collective file operations. Experiments with synthetic benchmarks have been carried out on a single node of an NEC SX-8 vector processor.

3.1 One-Sided Communication

For one-sided communication a simple, exchange-like benchmark as described in [11] is used, see also [1]. Each process performs one-sided get or put operations with a specified number of neighbors and synchronizes with one of the three synchronization mechanisms. Since one-sided communication benchmarks rarely report on performance using MPI derived datatypes (SKaMPI has the capability to do this [1]) we have extended the exchange benchmark with a number of synthetic datatypes. The exchange benchmark perform a small number of measurements over a spectrum of data sizes, and for each data size records the *best* (minimum) time measured; this value is stable and reproducible (and requires only few iterations to achieve).

We report findings with the following combination of origin and target types, all with MPI_INT as base type:

vector Origin: 3 blocks of 4 elements, stride 7. Target: 4 blocks of 3 elements, stride 5.

indexed Origin: Blocks of 1, 2, 3, 4, 5 elements spaced one element apart.

Target: Blocks of 5, 4, 3, 2, 1 elements spaced one element apart.

nested Nested structures of vectors of indexed type. Origin: 3 blocks of 4 indexed as origin above. Target: 4 blocks of 3 indexed as target above.

The results from an SX-8 with 4 processes, each communicating with all 4 neighbors (including itself) are shown in Figure 4, 5, and 6. The synchronization mechanism is in all cases MPI_Win_fence.

The transpacking implementation is compared to an implementation in which the typed copy is performed by packing and unpacking via an intermediate, consecutive buffer. The transpacking implementation has been run both with input-output type caching enabled and disabled. By the construction of the benchmark, when caching is enabled, the type construction time is completely factored out, since type construction is done only in the very first measurement iteration, and therefore will not be the recorded minimum time. The experiments therefore shows the cost of type construction relative to communication time.

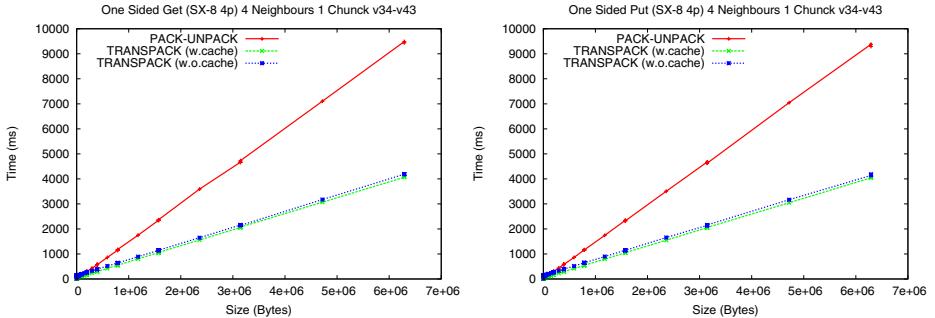


Fig. 4. SX-8 one-sided communication performance, time in microseconds as a function of number of bytes transferred. Vector, get and put, fence synchronization, transpacking with and without type caching.

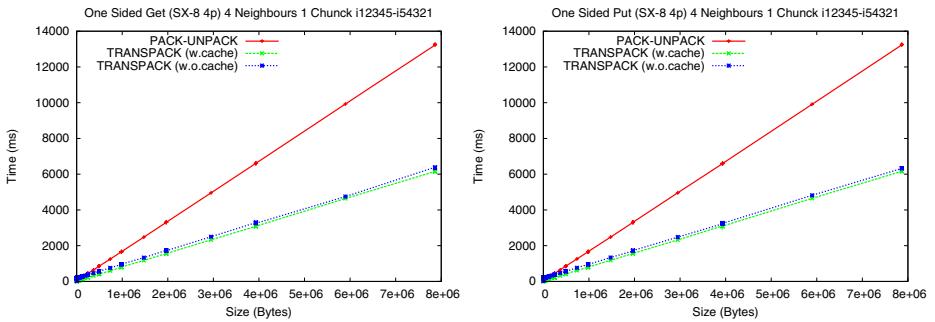


Fig. 5. SX-8 one-sided communication performance, time in microseconds as a function of number of bytes transferred. Indexed, get and put, fence synchronization, transpacking with and without type caching.

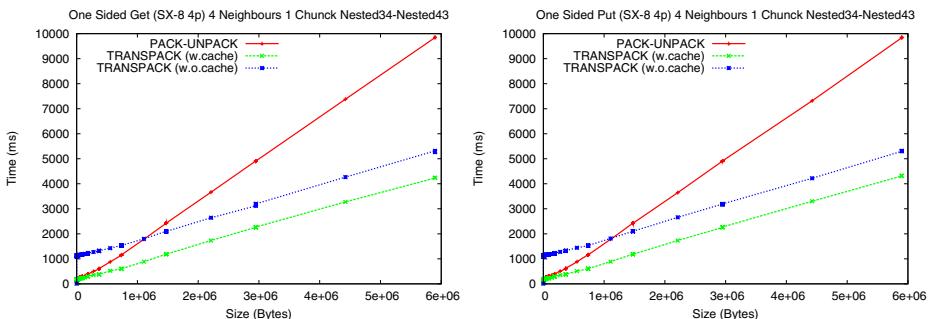


Fig. 6. SX-8 one-sided communication performance, time in microseconds as a function of number of bytes transferred. Nested structure, get and put, fence synchronization, transpacking with and without type caching.

The transpacking in all cases easily achieves the expected factor two performance increase, in many cases better. This is due to the change in memory access strides between the two implementations, which on a vector machine can effect performance significantly. Without caching, there is for the nested case a significant type construction overhead, such that break even between the two methods is reached at the order of several hundred KBytes. Thus, caching is definitely needed for the transpacking functionality to amortize the type construction overheads, possibly even coupled with performance prediction functions to decide if and from which data sizes type construction and transpacking become worthwhile. The latter prediction functions will be highly architecture dependent, and are subject to research.

3.2 Non-collective MPI-IO

We used the non-contig benchmark described in [12], and compare an implementation using transpacking as outlined in Figure 2 to a similar implementation [12] in which the typed memory copy from the non-contiguous intermediate buffer to the correct segment of the memory buffer is done by packing and unpacking via yet another, intermediate, but consecutive buffer. For this implementation additional typed buffer navigation functions as described in [12] are necessary. For I/O it is obvious that the actual file transfers will (on most systems) dominate performance, so the advantages of internal, memory copy optimizations like transpacking will have a more modest effect. We therefore concentrated on a case where the I/O is fast, and used a fast Network Attached Storage (NAS) RAID file system. The intermediate buffer size buffer size was chosen to 4MBytes (this is a user tunable parameter), and vectors with blocklength 8 were used.

The comparison between the two implementations is shown in Figure 7. As can be seen, improvements in both read and write time with single `MPI_File_read` and `MPI_File_write` calls of about 10% were achieved. For the write time a

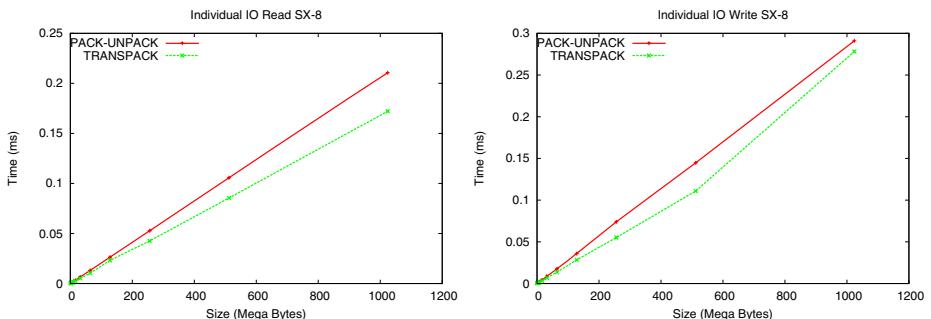


Fig. 7. Transpacking in MPI-IO, non-collective, individual read and write on NAS RAID filesystem

decrease can be seen with increasing write buffer size as actual file transfers become dominant.

In the absence of performance prediction functions for transpacking, the user can give a file hint (through the MPI info object in the file open or file info set calls) on whether transpacking shall be used in data sieving or not.

4 Conclusion

We analyzed requirements to a direct, typed copy functionality for structured, non-consecutive MPI data buffers, and showed how to integrate this in implementations of one-sided communication and non-collective MPI-IO. This is a non-trivial data type optimization with potential for significant improvements (theoretically up to a factor of two) in scenarios where the functionality is called for. For one-sided communication we could on an NEC SX-8 vector system show the expected performance gains, and we also showed the possible, non-trivial gains in MPI-IO for fast I/O systems. An evaluation with real applications is pending. We also showed the necessity of caching of constructed input-output types to amortize the construction overheads. This probably needs to be coupled with (architecture dependent) performance prediction functionality to decide when to employ this functionality. The latter is future work. We finally note that the transpack facility and extended, typed copy function can replace and simplify traditional MPI internal machinery for copying structured data by eliminating the need for dedicated pack and unpack functions.

References

1. Augustin, W., Straub, M.-O., Worsch, T.: Benchmarking one-sided communication with SKaMPI 5. In: Di Martino, B., Kranzlmüller, D., Dongarra, J. (eds.) EuroPVM/MPI 2005. LNCS, vol. 3666, pp. 301–308. Springer, Heidelberg (2005)
2. Byna, S., Gropp, W.D., Sun, X.-H., Thakur, R.: Improving the performance of MPI derived datatypes by optimizing memory-access cost. In: IEEE International Conference on Cluster Computing (CLUSTER 2003), pp. 412–419 (2003)
3. Byna, S., Sun, X.-H., Thakur, R., Gropp, W.D.: Automatic memory optimizations for improving MPI derived datatype performance. In: Mohr, B., Träff, J.L., Wörtingen, J., Dongarra, J. (eds.) PVM/MPI 2006. LNCS, vol. 4192, pp. 238–246. Springer, Heidelberg (2006)
4. Isaila, F., Tichy, W.F.: Mapping functions and data redistribution for parallel files. *The Journal of Supercomputing* 46(3), 213–236 (2008)
5. Mir, F.G., Träff, J.L.: Constructing MPI input-output datatypes for efficient transpacking. In: Lastovetsky, A., Kechadi, T., Dongarra, J. (eds.) EuroPVM/MPI 2008. LNCS, vol. 5205, pp. 141–150. Springer, Heidelberg (2008)
6. MPI Forum.MPI: A Message-Passing Interface Standard. Version 2.1, September 4 (2008), <http://www.mpi-forum.org>
7. Ross, R., Miller, N., Gropp, W.D.: Implementing fast and reusable datatype processing. In: Dongarra, J., Laforenza, D., Orlando, S. (eds.) EuroPVM/MPI 2003. LNCS, vol. 2840, pp. 404–413. Springer, Heidelberg (2003)

8. Santhanaraman, G., Wu, D., Panda, D.K.: Zero-copy MPI derived datatype communication over InfiniBand. In: Kranzlmüller, D., Kacsuk, P., Dongarra, J. (eds.) EuroPVM/MPI 2004. LNCS, vol. 3241, pp. 47–56. Springer, Heidelberg (2004)
9. Thakur, R., Gropp, W., Lusk, E.: Optimizing noncontiguous accesses in MPI-IO. Parallel Computing 28, 83–105 (2002)
10. Träff, J.L., Hempel, R., Ritzdorf, H., Zimmermann, F.: Flattening on the fly: efficient handling of MPI derived datatypes. In: Margalef, T., Dongarra, J., Luque, E. (eds.) PVM/MPI 1999. LNCS, vol. 1697, pp. 109–116. Springer, Heidelberg (1999)
11. Träff, J.L., Ritzdorf, H., Hempel, R.: The implementation of MPI-2 one-sided communication for the NEC SX-5. In: Supercomputing (2000),
<http://www.sc2000.org/proceedings/techpapr/index.htm#01>
12. Worringen, J., Träff, J.L., Ritzdorf, H.: Fast parallel non-contiguous file access. In: Supercomputing (2003),
http://www.sc-conference.org/sc2003/tech_papers.php
13. Wu, J., Wyckoff, P., Panda, D.K.: High performance implementation of MPI derived datatype communication over InfiniBand. In: 18th International Parallel and Distributed Processing Symposium (IPDPS 2004), p. 14 (2004)

Multiple-Level MPI File Write-Back and Prefetching for Blue Gene Systems

Javier García Blas¹, Florin Isailă¹, J. Carretero¹,
Robert Latham², and Robert Ross²

¹ University Carlos III, Spain

{fjblas,florin,jcarrete}@arcos.inf.uc3m.es

² Argonne National Laboratory

{robl,rross}@mcs.anl.gov

Abstract. This paper presents the design and implementation of an asynchronous data-staging strategy for file accesses based on ROMIO, the most popular MPI-IO distribution, and ZeptoOS, an open source operating system solution for Blue Gene systems. We describe and evaluate a two-level file write-back implementation and a one-level prefetching solution. The experimental results demonstrate that both solutions achieve high performance through a high degree of overlap between computation, communication, and file I/O.

Keywords: MPI-IO, Parallel I/O, Parallel File Systems, Supercomputers.

1 Introduction

The past few years have shown a continuous increase in the performance of supercomputers and clusters, as demonstrated by the evolution of Top 500 [1]. IBM's Blue Gene supercomputers have a significant share in the Top 500 lists and bring additionally the advantage of a highly energy-efficient solution. Blue Gene systems scale up to hundreds of thousands of processors. In order to allow data-intensive applications to make efficient use of the system, compute scale needs to be matched by a corresponding performance of file I/O.

In earlier work [2] we presented the design and implementation of an MPI-based hierarchical I/O cache architecture for Blue Gene/L systems based on open source software. Our solution was based on an asynchronous data-staging strategy that hides the latency of file system access of collective file operations from compute nodes. Blue Gene/L presents two limitations, which affected the efficiency of our solution. First, the BG/L compute nodes do not support multithreading. Therefore, computation can not be overlapped with the communication with the I/O node. Second, because L1 caches of the CPUs are not coherent, even though multithreading is supported on the I/O nodes, the threads could use only one processor. Consequently, the communication with the compute node could not be overlapped with file system activity. The limitations discussed above

have been removed from the Blue Gene/P architecture: Blue Gene/P has limited multithreading support, but adequate for our caching strategies and the L1 caches of the cores are coherent.

In this paper we discuss the evolution of our hierarchical I/O cache architecture and its deployment on Blue Gene/P systems. This paper makes the following contributions:

- We discuss the extensions of the initial Blue Gene/L solution necessary for the efficient porting to Blue Gene/P systems.
- The novel solution includes a compute-node write-back policy, which asynchronously transfers data **from the compute nodes to the I/O nodes**. This policy complements the initial data-staging strategy, which performed asynchronous transfers **between I/O nodes and the final storage system**.
- We describe and evaluate an I/O node prefetching strategy that asynchronously transfers file data from the storage system to the I/O node.
- We present and analyze a novel evaluation on a Blue Gene/P system.

The remainder of the paper is structured as follows. Section 2 reviews related work. The hardware and operating system architectures of Blue Gene/P are presented in Section 3. We discuss our novel solution for Blue Gene/P systems in Section 4. The experimental results are presented in Section 5. We summarize and discuss future work in Section 6.

2 Related Work

Several researchers have contributed techniques for hiding the latency of file system accesses. Active buffering is an optimization for MPI collective write operations [3] based on an I/O thread performing write-back in background. Write-behind strategies [4] accumulate multiple small writes into large, contiguous I/O requests in order to better utilize the network bandwidth. In this paper we present a two level write back strategy: aggregators merge small requests into file blocks written back in background to I/O nodes, while I/O nodes cache file blocks and write them asynchronously to the storage nodes. A number of works have proposed I/O prefetching techniques based on application disclosed access patterns [5], signatures derived from access pattern classifications [6], and speculative execution [7,8]. The I/O node prefetching strategy from this paper leverages the file block assignment to the aggregators and it is the basis of a two-level prefetching strategy on which we are currently working.

A limited number of recent studies have proposed and evaluated parallel I/O solutions for supercomputers. An implementation of MPI-IO for Cray architecture and the Lustre file system is described in [9]. In [10] the authors propose a collective I/O technique, in which processes are grouped together for collective I/O according to the Cray XT architecture. Yu et al. [11] present a GPFS-based three-tiered architecture for Blue Gene/L. The tiers are represented by I/O nodes (GPFS clients), network-shared disks, and a storage area network. Our solution

focuses on the Blue Gene/P, extends this hierarchy to include the memory of the compute nodes and proposes an asynchronous data-staging strategy that hides the latency of file accesses from the compute nodes.

3 Blue Gene/P

This section presents the hardware and operating system architectures of Blue Gene/P.

3.1 Blue Gene/P Architecture

Figure 1 shows a high-level view of a Blue Gene/P system. Compute nodes are grouped into processing sets, or “psets.” Applications run in exclusivity on partitions, consisting of multiples of psets. Each pset has an associated I/O node, which performs I/O operations on behalf of the compute nodes from the pset. The compute and I/O nodes are controlled by service nodes. The file system components run on dedicated file servers connected to storage nodes through a 10Gbit Ethernet switch. Compute and I/O nodes use the same ASIC with four PowerPC 450 cores, with core-private hardware-coherent L1 caches, core-private stream prefetching L2 caches, and a 8 MB shared DRAM L3 cache.

Blue Gene/P nodes are interconnected by five networks: 3D torus, collective, global barrier, 10 Gbit Ethernet, and control. The 3D torus (5.1 GBytes/s) is typically used for point-to-point communication between compute nodes. The collective network (1700 MBytes/s) has a tree topology and serves collective communication operations and I/O traffic. The global barrier network offers an efficient barrier implementation. The 10 Gbit Ethernet network interconnects I/O nodes and file servers. The service nodes control the whole machine through the control network.

3.2 Operating System Architecture

In the IBM solution [12], the compute node kernel (CNK) is a light-weight operating system offering basic services such as setting an alarm or getting the

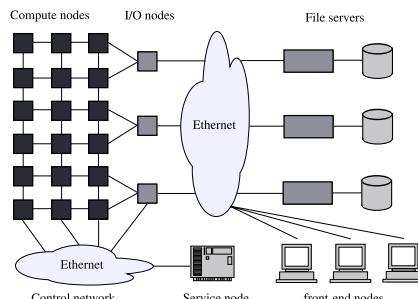


Fig. 1. Blue Gene/P architecture overview

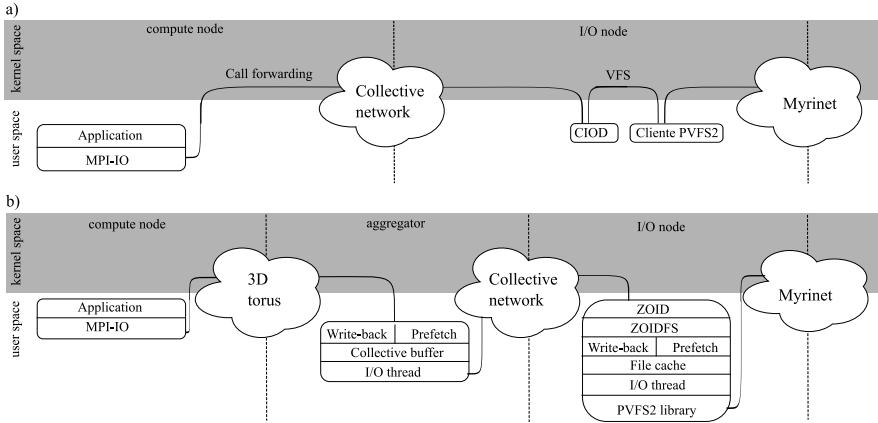


Fig. 2. I/O forwarding for IBM and ZeptoOS solutions

time. As shown in Figure 2(a), the I/O system calls are forwarded through the tree collective network to the I/O node. The I/O nodes run a simplified Linux OS kernel (IOK) with a small memory footprint, an in-memory root file system, TCP/IP and file system support, and no swapping and lacking the majority of classical daemons. The forwarded calls are served on the I/O node by the control and I/O daemon (CIOD). CIOD executes the requested system calls on locally mounted file systems and returns the results to the compute nodes.

An open-source alternative to the IBM's solution is developed in the ZeptoOS project [13]. Under ZeptoOS, Blue Gene compute nodes may run Linux, while the I/O forwarding is implemented in a component called ZOID. The I/O forwarding process shown in Figure 2(b) is similar to the one based on CIOD, in the sense that I/O related calls are forwarded to the I/O nodes, where a multithreaded daemon serves them. However, there are two notable differences in design and implementation between CIOD-based and ZOID-based solutions. First, ZOID comes with its own network protocol, which can be conveniently extended with the help of a plug-in tool, which automatically generates the communication code for new forwarded calls. Second, the file system calls are forwarded through ZOIDFS [14], an abstract interface for forwarding file system calls. ZOIDFS abstracts away the details of a file system API under a stateless interface consisting of generic functions for file create, open, write, read, close, and so forth.

4 Write-Back and Prefetching Policies

In this section we describe how we extended our initial parallel I/O design for Blue Gene systems. The solution is based on the ROMIO implementation of the MPI-IO standard [15] and ZOIDFS and is shown in Figure 2(b).

On the *compute node side*, the file system calls performed through MPI-IO are mapped to ADIO. ADIO [16] is an abstract device interface, which consists

of general-purpose components and a file-system specific implementation. In this case ADIO maps the calls to the ZOIDFS file system. The general-purpose components are the collective cache, the write-back module, and the prefetching module. In this paper we describe the design, implementation, and evaluation of the write-back module. The compute node prefetching module is part of our current efforts.

On the *I/O node side*, ZOIDFS maps the forwarded calls to specific file system calls (in our case PVFS). Between ZOIDFS and the file system there is a file cache accessed by the I/O node write-back and prefetching modules. The write-back module was implemented in our previous work and its performance evaluated for BG/L. The novelties presented in this paper are the prefetching module and an evaluation of the whole solution on BG/P systems. The write-back mechanism is implemented at each I/O node in one I/O thread. The I/O thread asynchronously writes file blocks to the file system following a high-low watermark policy, where the watermark is the number of dirty pages. When the high watermark is reached, the I/O thread is activated. The I/O thread flushes to the file system the last modified pages until the low watermark is reached.

In our solution, file consistency is ensured by assigning a file block exclusively to one aggregator and caching the respective file block only on the I/O node responsible for the pset of the aggregator. Therefore, all the processes send their accesses to the aggregators over the torus network, and in its turn each aggregator exclusively accesses a non-replicated cache block at I/O node over the tree network.

4.1 Write-Back Collective I/O

Our initial Blue Gene/L collective I/O solution was based on view-based collective I/O [17]. In view-based I/O each file block is uniquely mapped to one process called aggregator, which is responsible to perform the file access on behalf of all processes of the application. A view is an MPI mechanism that allows application to see noncontiguous regions of a file as contiguous. View-based I/O leverages this mechanism for implementing an efficient collective I/O strategy. When defined, the views are sent to aggregators, where they can be used by all the accesses. View-based I/O writes and reads can take advantage of collective buffers managed by aggregators, which are responsible for performing the file access on behalf of all processes of the application. At access time, contiguous view data can be transferred between compute nodes and aggregators: using the view parameters, the aggregator can perform locally the scatter/gather operations between view data and file blocks.

In our initial Blue Gene/L solution, the modified collective buffers were transferred to the I/O node either when the memory was full or when the file was closed. This approach was taken because the Blue Gene/L did not offer support for threads on the compute nodes.

For Blue Gene/P, however, the collective buffer cache is asynchronously written back to the file system by an I/O thread. The write-back allows for overlapping the computation and I/O, by gradually transferring the data from the

collective buffer cache to the I/O nodes. This approach distributes the cost of the file access over the computation phase and is especially efficient for scientific applications, which alternate computation and I/O phases.

The new implementation of our hierarchical I/O cache includes a two-level asynchronous file write strategy. The compute nodes asynchronously write back data to the I/O nodes, while the I/O nodes write asynchronously back data to the storage system.

4.2 Asynchronous I/O Node Prefetching

Another contribution of this paper is the design and implementation of a prefetching module at I/O node. The objective of I/O node prefetching is to hide the latency of read operations by predicting the future accesses of compute nodes.

The prefetching mechanism is leveraged by each I/O thread running at the I/O node. The prediction is based on the round-robin collective buffer distribution over the aggregators. Given that all collective buffers of an aggregator are mapped on the same I/O node, we implemented a simple strided prefetching policy. When an on-demand or a prefetch read for a collective block corresponding to an aggregator returns, a new prefetch is issued for the next file block of the same aggregator. The prefetching is performed by the I/O thread, while on-demand reads are issued by the thread serving the requesting compute node. A cache block, for which a read request has been issued, is blocked in memory, and all threads requesting data from the same page block at a condition variable. A further relaxation of this approach by allowing reads from page hits to bypass reads from page misses is possible, but it has not been implemented in the current prototype.

In the current implementation, only the I/O nodes perform prefetching into their local caches. However, we are developing an additional layer of prefetching between compute nodes and I/O nodes and its integration with the I/O prefetching module at I/O node presented in this paper.

5 Experimental Results

The experiments presented in this paper have been performed on the Blue Gene/P system from Argonne National Laboratory. The system has 1024 quad-core 850 MHz PowerPC 450 processors with 2 GB of RAM. All the experiments were run in Symmetric Multiprocessor mode (SMP), in which a compute node executes one MPI process per node with up to four threads per process. The PVFS file system at Argonne consists of four servers. The PVFS files were striped over all four servers with a stripe size of 4 MB.

5.1 SimParIO

We have implemented an MPI benchmark called SimParIO that simulates the behavior of data-intensive parallel applications. The benchmark consists of alternating compute and I/O phases. The compute phases are simulated by idle

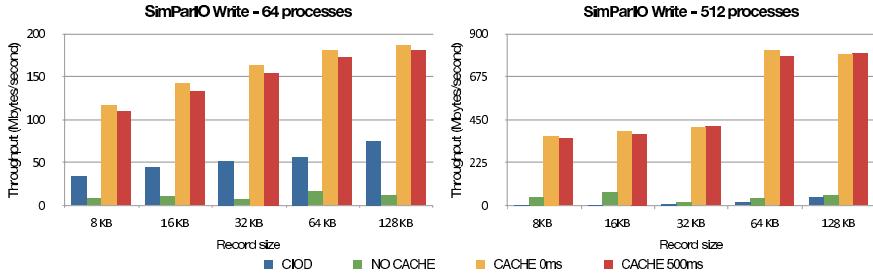


Fig. 3. SimParIO write performance for different record sizes for 64 and 512 processors

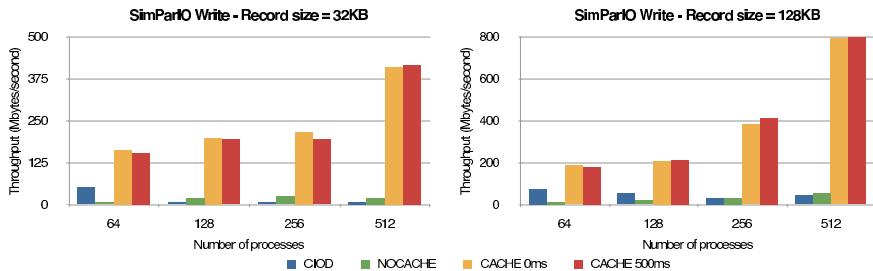


Fig. 4. SimParIO write performance for a fixed record size of 32KB and 128KB

spinning. In the I/O phase all the MPI processes write non-overlapping records to a file. The number of alternating phases was 20. The maximum file size produced by 512 processes and record size of 128 KB was 1.25 GB. The compute nodes do not use any caching.

We compare four cases: IBM's CIOD-based solution, ZOID without cache, ZOID with cache and zero-time compute phase, and ZOID with cache with a 0.5 seconds compute phase. In one setup we have fixed the record size (we report two cases: 128 KB and 1 MB) and varied the number of processors from 64 (one pset) to 512 (8 psets). In the second setup we use a fixed number of processors (we report two cases: 64 and 512 processors) and vary the record size from 1 KB to 128 KB. Figures 3 and 4 show the aggregate file write throughput results. Figures 5 and 6 plot the aggregate file read throughput.

As expected, the file writes based on the write-back cache significantly outperform the CIOD-based and the no-cached solution in all cases. The duration of the compute phase does not seem to influence the performance of the write-back strategy. This indicates a good overlap of communication from the compute nodes to I/O nodes and the write-back from compute nodes to the file system.

For file reads, the prefetching policy at I/O nodes works especially well for compute phases of 0.5 seconds. This performance benefit comes from the overlap between computation and file reads at I/O nodes. For 0 seconds compute phases, all 64 processes of a pnode request file reads on-demand roughly at the same time and reissue a new on-demand read request as soon as the previous one

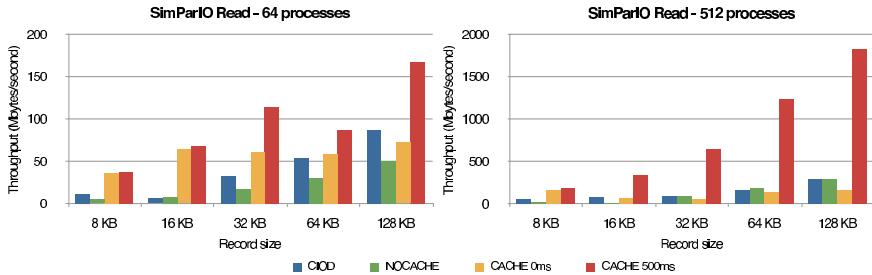


Fig. 5. SimParIO read performance for different record sizes for 64 and 512 processors

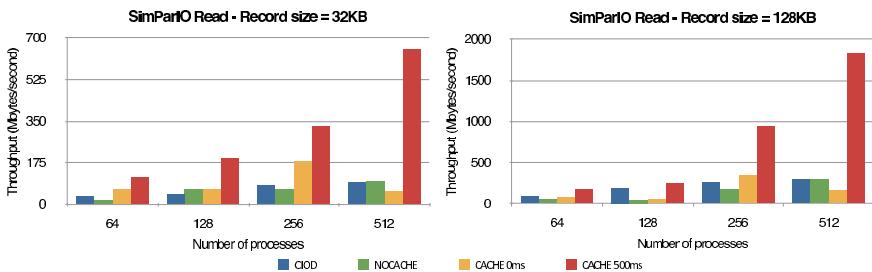


Fig. 6. SimParIO read performance for a fixed record size of 32KB and 128KB

has been served. In this case the current prefetching implementation has little margin for improvement, and its costs may even outweigh the benefits.

5.2 BTIO Benchmark

NASA's BTIO benchmark [18] solves the block-tridiagonal (BT) problem, which employs a complex domain decomposition across a square number of compute nodes. The execution alternates computation and I/O phases. Initially, all compute nodes collectively open a file and declare views on the relevant file regions. After each five computing steps the compute nodes write the solution to a file through a collective operation. At the end, the resulting file is collectively read and the solution verified for correctness. The aggregator cache on each compute node has 128 MB, while the I/O node buffer cache had 512 MB. All nodes acted as aggregators. We report the results for class B producing a file of 1.6 GB.

We have run the BTIO benchmark with four variants of collective I/O file writes: two-phase collective I/O from ROMIO (TP), view-based I/O with no write-back (VB-original), view-based I/O with one-level compute-node write-back (VB-CNWB), and view-based I/O with two-level write back (VB-2WB). Figure 7 shows the breakdown of time into compute time, file write time, and close time. The close time is relevant because all the data is flushed to the file system when the file is closed. We notice that in all solutions the compute time is roughly the same. VB-original reduces the file write time without any

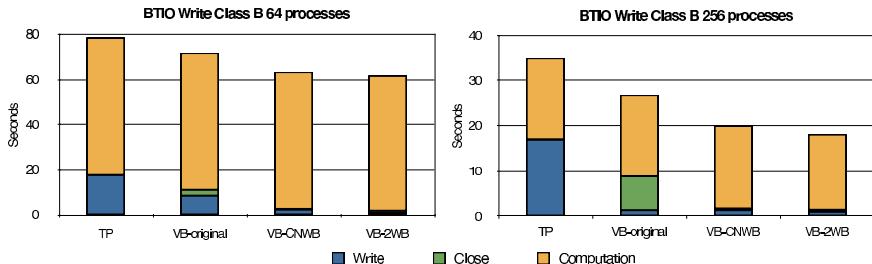


Fig. 7. BTIO class B times for 64 and 256 processors

asynchronous access. VB-CNWB reduces both the write time and close time, as data is asynchronously written from compute node to I/O node. For VB-2WB, the network and I/O activity are almost entirely overlapped with computation. We conclude that the performance of the file writes gradually improved with the increasing degree of asynchrony in the system.

6 Conclusions and Future Work

In this paper we describe the novel implementation of a multiple-level cache architecture for Blue Gene systems. The novel solution includes a compute-node write-back policy, which asynchronously transfers data from the compute nodes to the I/O nodes. This policy complements the initial data-staging strategy, which was performing asynchronous transfers between I/O nodes and the file system. Additionally, we implemented and evaluated an I/O node prefetching strategy, which asynchronously transfers file data from the file system to the I/O node. Both the data-staging and prefetching strategies provide a significant performance benefit, whose main source comes from the efficient utilization of the Blue Gene parallelism and asynchronous transfers across file cache hierarchy.

There are several directions we are currently working on. The solution presented in this paper is specific for Blue Gene architectures. However, the compute node part runs unmodified on any cluster, by leveraging the ADIO interface. In our cluster solution the I/O node cache level is managed by AHPIOS middleware [19]. Additionally, we are implementing and evaluating the prefetching module on the compute node. Finally, we plan to perform a cost-benefit analysis of various coordination policies among the several levels of caches.

Acknowledgments

This work was supported in part by Spanish Ministry of Science and Innovation under the project TIN 2007/6309, by the U.S. Dept. of Energy under Contracts DE-FC02-07ER25808, DE-FC02-01ER25485, and DE-AC02-06CH11357, and NSF HECURA CCF-0621443, NSF SDCIOCI-0724599, and NSF ST-HEC CCF-0444405.

References

1. Top 500 list, <http://www.top500.org>
2. Isaila, F., Garcia Blas, J., Carretero, J., Latham, R., Lang, S., Ross, R.: Latency hiding file I/O for Blue Gene systems. In: CCGRID 2009 (2009)
3. Ma, X., Winslett, M., Lee, J., Yu, S.: Improving MPI-IO Output Performance with Active Buffering Plus Threads. In: IPDPS, pp. 22–26 (2003)
4. Liao, W.-k., Coloma, K., Choudhary, A.K., Ward, L.: Cooperative Write-Behind Data Buffering for MPI I/O. In: Di Martino, B., Kranzlmüller, D., Dongarra, J. (eds.) EuroPVM/MPI 2005. LNCS, vol. 3666, pp. 102–109. Springer, Heidelberg (2005)
5. Patterson, R.H., Gibson, G.A., Ginting, E., Stodolsky, D., Zelenka, J.: Informed prefetching and caching. SIGOPS Oper. Syst. Rev., 79–95 (1995)
6. Byna, S., Chen, Y., Sun, X.H., Thakur, R., Gropp, W.: Parallel I/O prefetching using MPI file caching and I/O signatures. In: SC 2008, pp. 1–12 (2008)
7. Chang, F., Gibson, G.: Automatic I/O Hint Generation Through Speculative Execution. In: Proceedings of OSDI (1999)
8. Chen, Y., Byna, S., Sun, X.H., Thakur, R., Gropp, W.: Hiding I/O latency with pre-execution prefetching for parallel applications. In: SC 2008, pp. 1–10 (2008)
9. Yu, W., Vetter, J.S., Canon, R.S.: OPAL: An Open-Source MPI-IO Library over Cray XT. In: SNAPI 2007, pp. 41–46 (2007)
10. Yu, W., Vetter, J.: ParColl: Partitioned Collective I/O on the Cray XT. In: ICPP, pp. 562–569 (2008)
11. Sahoo, Y.H., Howson, R., et al: High performance file I/O for the Blue Gene/L supercomputer. In: HPCA, pp. 187–196 (2006)
12. Moreira, J., et al.: Designing a highly-scalable operating system: the Blue Gene/L story. In: SC 2006, p. 118 (2006)
13. ZeptoOs Project (2008), <http://wwwunix.mcs.anl.gov/zeptoos/>
14. Iskra, K., Romein, J.W., Yoshii, K., Beckman, P.: ZOID: I/O-forwarding infrastructure for petascale architectures. In: PPoPP 2008, pp. 153–162 (2008)
15. Thakur, R., Gropp, W., Lusk, E.: On Implementing MPI-IO Portably and with High Performance. In: Proc. of IOPADS, May 1999, pp. 23–32 (1999)
16. Thakur, R., Lusk, E.: An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces. In: Proc. of The 6th Symposium on the Frontiers of Massively Parallel Computation, pp. 180–187 (1996)
17. Blas, J.G., Isaila, F., Singh, D.E., Carretero, J.: View-Based Collective I/O for MPI-IO. In: CCGRID, pp. 409–416 (2008)
18. Wong, P., der Wijngaart, R.: NAS Parallel Benchmarks I/O Version 2.4. Technical report, NASA Ames Research Center (2003)
19. Isaila, F., Blas, J.G., Carretero, J., Liao, W.K., Choudhary, A.: AHPIOS: An MPI-based ad-hoc parallel I/O system. In: Proceedings of IEEE ICPADS (2008)

Performance Evaluation of MPI, UPC and OpenMP on Multicore Architectures

Damián A. Mallón¹, Guillermo L. Taboada², Carlos Teijeiro², Juan Touriño², Basilio B. Fraguerau², Andrés Gómez¹, Ramón Doallo², and J. Carlos Mouriño¹

¹ Galicia Supercomputing Center (CESGA), Santiago de Compostela, Spain

² Computer Architecture Group, University of A Coruña, A Coruña, Spain

{dalvarez, agomez, jmourino}@cesga.es

{taboada, cteijeiro, juan, basilio, doallo}@udc.es

Abstract. The current trend to multicore architectures underscores the need of parallelism. While new languages and alternatives for supporting more efficiently these systems are proposed, MPI faces this new challenge. Therefore, up-to-date performance evaluations of current options for programming multicore systems are needed. This paper evaluates MPI performance against Unified Parallel C (UPC) and OpenMP on multicore architectures. From the analysis of the results, it can be concluded that MPI is generally the best choice on multicore systems with both shared and hybrid shared/distributed memory, as it takes the highest advantage of data locality, the key factor for performance in these systems. Regarding UPC, although it exploits efficiently the data layout in memory, it suffers from remote shared memory accesses, whereas OpenMP usually lacks efficient data locality support and is restricted to shared memory systems, which limits its scalability.

Keywords: MPI, UPC, OpenMP, Multicore Architectures, Performance Evaluation, NAS Parallel Benchmarks (NPB).

1 Introduction

Currently, multicore clusters are the most popular option for the deployment of High Performance Computing (HPC) infrastructures, due to their scalability and performance/cost ratio. These systems are usually programmed using MPI [1] on distributed memory, OpenMP [2] on shared memory, and MPI+OpenMP on hybrid shared/distributed memory architectures. Additionally, the Partitioned Global Address Space (PGAS) languages, such as Unified Parallel C (UPC) [3], are an emerging alternative that allows shared memory-like programming on distributed memory systems, taking advantage of the data locality, being of special interest for hybrid architectures.

In the past other alternatives, such as High Performance Fortran, aimed to replace MPI as the primary choice for HPC programming. These alternatives have been dismissed and MPI is still the preferred platform for HPC developers due to its higher performance and portability. In order to evaluate the current

validity of this assessment, this work presents an up-to-date comparative performance evaluation of MPI, UPC and OpenMP on two multicore scenarios: a cluster composed of 16-core nodes interconnected via InfiniBand, and a 128-core shared memory system. This evaluation uses the standard parallel benchmarking suite, the NAS Parallel Benchmarks (NPB) [4]. A matrix multiplication kernel and the Sobel edge detection kernel, have also been used to assess the scalability of the evaluated parallel programming solutions.

This paper is structured as follows: Section 2 presents current trends in multicore parallel programming (MPI, OpenMP and PGAS). Section 3 introduces the benchmarks used for the comparative evaluation and related work. Benchmarking results obtained from two multicore systems are shown and analyzed in Section 4. Finally, Section 5 presents the main conclusions of this evaluation.

2 Parallel Programming for Multicore Architectures

This section presents three of the main options for parallel programming multicore architectures. These approaches are the message-passing paradigm, shared memory programming, and the PGAS programming model.

The message-passing is the most widely used parallel programming model as it is portable, scalable and provides good performance for a wide variety of computing platforms and codes. It is the preferred choice for parallel programming distributed memory systems, such as multicore clusters. Therefore, as the programmer has to manage explicitly data placement through point-to-point or collective operations, the programming of these architectures is difficult. MPI is the standard interface for message-passing libraries and there is a wide range of MPI implementations, both from HPC vendors and the free software community, optimized for high-speed networks, such as InfiniBand or Myrinet. MPI, although is oriented towards distributed memory environments, faces the raise of the number of cores per system with the development of efficient shared memory transfers and providing thread safety support.

The shared memory programming model allows a simpler programming of parallel applications, as here the control of the data location is not required. OpenMP is the most widely used solution for shared memory programming, as it allows an easy development of parallel applications through compiler directives. Moreover, it is becoming more important as the number of cores per system increases. However, as this model is limited to shared memory architectures, the performance is bound to the computational power of a single system. To avoid this limitation, hybrid systems, with both shared/distributed memory, such as multicore clusters, can be programmed using MPI+OpenMP. However, this hybrid model can make the parallelization more difficult and the performance gains could not compensate for the effort [5,6].

The PGAS programming model combines the main features of the message-passing and the shared memory programming models. In PGAS languages, each thread has its own private memory space, as well as an associated shared memory region of the global address space that can be accessed by other threads,

although at a higher cost than a local access. Thus, PGAS languages allow shared memory-like programming on distributed memory systems. Moreover, as in message-passing, PGAS languages allow the exploitation of data locality as the shared memory is partitioned among the threads in regions, each one with affinity to the corresponding thread. This feature of the PGAS languages could make them important in the near future, as modern multicore architectures became NUMA architectures with the inclusion of the memory controller in the CPU. UPC is the PGAS extension to the ISO C language, and has been used in this evaluation due to its important support by academia and industry, with compilers available from Univ. of Berkeley, Michigan Tech. Univ., Intrepid/GCC, IBM, Cray, and HP. UPC provides PGAS features to C, allowing a more productive code development [7]. However, as an emerging programming model, performance analysis are needed [8,9,10].

3 Parallel Benchmarks on Multicore Architectures

A comparative performance evaluation needs standard, unbiased benchmarks with implementations for the evaluated options. As UPC is an emerging option, the number of available benchmarks for UPC benchmarking is limited. The main source of UPC benchmarks is the Berkeley UPC distribution [11], which includes the codes selected for our evaluation: the UPC version of the NPB and the Sobel edge detection kernel coded in MPI, UPC and OpenMP. Additionally, we have implemented a blocking algorithm for matrix multiplication.

3.1 NAS Parallel Benchmarks Description

The NPB consist of a set of kernels and pseudo-applications, taken primarily from Computational Fluid Dynamics (CFD) applications. These benchmarks reflect different kinds of computation and communication patterns that are important across a wide range of applications, which makes them the de facto standard in parallel performance benchmarking. There are NPB implementations available for a wide range of parallel programming languages and libraries, such as MPI (from now on NPB-MPI), UPC (from now on NPB-UPC), OpenMP (from now on NPB-OMP), a hybrid MPI+OpenMP implementation (not used in this comparative evaluation as it implements benchmarks not available in NPB-UPC), HPF and Message-Passing Java [12], among others.

The NPB evaluated are: CG, EP, FT, IS and MG. NPB-MPI and NPB-OMP are implemented using Fortran, except for IS which is programmed in C. The fact that the NPB are programmed in Fortran has been considered as cause of a poorer performance of NPB-UPC [9], due to better backend compiler optimizations for Fortran than for C. The CG kernel is an iterative solver that tests regular communications in sparse matrix-vector multiplications. The EP kernel is an embarrassingly parallel code that assesses the floating point performance, without significant communication. The FT kernel performs series of 1-D FFTs on a 3-D mesh that tests aggregated communication performance. The IS kernel

is a large integer sort that evaluates both integer computation performance and the aggregated communication throughput. MG is a simplified multigrid kernel that performs both short and long distance communications. Moreover, each kernel has several workloads to scale from small systems to supercomputers.

Most of the NPB-UPC kernels have been manually optimized through techniques that mature UPC compilers should handle in the future: privatization, which casts local shared accesses to private memory accesses, avoiding the translation from global shared address to actual address in local memory, and prefetching, which copies non-local shared memory blocks into private memory. Although OpenMP 3.0 adds data locality support, the NPB-OMP codes do not take advantage of these features.

3.2 Related Work

Up to now only three works have analyzed the performance of MPI against UPC in computational kernels [9,10,13]. The first work [9] has compared the performance of NPB-MPI with NPB-UPC on a 16 processor Compaq AlphaServer SC cluster, using the class B workload. The second work [10] has used two SGI Origin NUMA machines, each one with 32 processors, using the class A workload for 3 NPB. The Sobel edge detector kernel has also been used in both works. The third work [13] compares MPI with UPC very briefly using 2 NPB kernels and class B, in a Cray X1 machine, with up to 64 processors. Other works have tackled the kernel and micro-benchmarking of low-level parameters on multicore architectures, especially memory hierarchy performance [14,15], but they are limited to 8-core systems, and do not evaluate distributed memory programming solutions such as MPI or UPC.

This paper improves previous works by: (1) using a higher number of processor cores (128) with different memory configurations (shared and hybrid shared/distributed memory); (2) including OpenMP performance in the shared memory configuration; (3) using a larger workload, more representative of large scale applications (class C); and (4) providing an up-to-date performance snapshot of MPI performance versus UPC and OpenMP on multicore architectures.

Finally, this paper addresses performance issues that are present in a cluster with 16-core nodes, and on a 128-core system, but not in 4- and 8-core systems. Thus, it is possible to foresee the main drawbacks that may affect performance on the multicore architectures that will be commonly deployed in the next years.

4 Performance Evaluation

The testbed used in this work is the Finis Terrae supercomputer [16], composed of 142 HP Integrity rx7640 nodes, each one with 8 Montvale Itanium 2 dual-core processors (16 cores per node) at 1.6 GHz and 128 GB of memory, interconnected via InfiniBand. The InfiniBand HCA is a dual 4X IB port (16 Gbps of theoretical effective bandwidth). For the evaluation of the hybrid shared/distributed memory scenario, 8 nodes have been used (up to 128 cores). The number of cores

used per node in the performance evaluation is $\lceil n/8 \rceil$, being n the total number of cores used in the execution, with consecutive distribution. An HP Integrity Superdome system with 64 Montvale Itanium 2 dual-core processors (total 128 cores) at 1.6 GHz and 1 TB of memory has also been used for the shared memory evaluation. The nodes were used without other users processes running, and the process affinity was handled by the operating system scheduler.

The OS is SUSE Linux Enterprise Server 10, and the MPI library is the recommended by the hardware vendor, HP MPI 2.2.5.1 using InfiniBand Verbs (IBV) for internode communication, and shared memory transfers (HP MPI SHM driver) for intranode communication. The UPC compiler is Berkeley UPC 2.8, which uses the IBV driver for distributed memory communication, and pthreads within a node for shared memory transfers. The backend for both and OpenMP compiler is the Intel 11.0.069 (icc/ifort, with -O3 flag).

4.1 Matrix Multiplication and Sobel Kernels Performance

Figure 1 shows the results for the matrix multiplication and the Sobel kernel. The matrix multiplication uses matrices of 2400×2400 doubles, with a blocking factor of 100 elements, and the experimental results include the data distribution overhead. The Sobel kernel uses a 65536×65536 unsigned char matrix and does not take into account the data distribution overhead. The graphs show the speedups on the hybrid scenario (MPI and UPC) and in the shared memory system (UPC and OpenMP).

The three options obtain similar speedups on up to 8 cores. MPI can take advantage of the use of up to 128 cores, whereas UPC (hybrid memory) presents poor scalability, as it lacks efficient collectives [17]. In shared memory, UPC and OpenMP show similar speedups up to 32 cores. However, on 128 cores UPC achieves the best performance, whereas OpenMP suffers an important performance penalty due to the sharing of one of the matrices, whereas in UPC this matrix is copied to private space, thus avoiding shared memory access contention. MPI shows better performance than OpenMP for this reason.

In the Sobel kernel results, because the data distribution overhead is not considered, the speedups are almost linear, except for UPC on the hybrid scenario, where several remote shared memory accesses limit seriously its scalability.

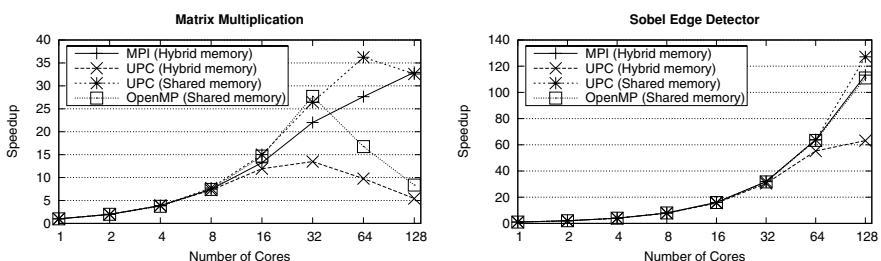


Fig. 1. Performance of matrix multiplication and Sobel kernels

Nevertheless, UPC on shared memory achieves the highest speedups as these remote accesses are intraprocess accesses (UPC uses pthreads in this scenario).

4.2 Performance of NPB Kernels on Hybrid Memory

Figure 2 shows NPB-MPI and NPB-UPC performance on the hybrid configuration (using both InfiniBand and shared memory communication). The left graphs show the kernels performance in MOPS (Million Operations Per Second), whereas the right graphs present their associated speedups.

Regarding the CG kernel, MPI performs slightly worse than UPC using up to 32 cores, due to the kernel implementation, whereas on 64, and especially on 128 cores MPI outperforms UPC. Although UPC uses pthreads within a node, its communication operations, most of them point-to-point transfers with a regular communication pattern, are less scalable than MPI primitives.

EP is an embarrassingly parallel kernel, and therefore shows almost linear scalability for both MPI and UPC. The results in MOPS are approximately 6 times lower for UPC than for MPI due to the poorer UPC compiler optimizations. EP is the only NPB-UPC kernel that has not been optimized through prefetching and/or privatization, and the workload distribution is done through a `upc forall` function, preventing more aggressive optimizations.

The performance of FT depends on the efficiency of the exchange collective operations. Although the UPC implementation is optimized through privatization, it presents significantly lower performance than MPI. The UPC results, although significantly lower than MPI in terms of MOPS, it shows higher speedups than MPI. This is a communication-intensive code that benefits from UPC intranode shared memory communication, which is maximized on 64 and 128 cores.

IS is a quite communication-intensive code. Thus, both MPI and UPC obtain low speedups for this kernel (less than 25 on 128 cores). Although UPC IS has been optimized using privatization, the lower performance of its communications limits its scalability, which is slightly lower than MPI speedups.

Regarding MG, MPI outperforms UPC in terms of MOPS, whereas UPC shows higher speedup. The reason is the poor performance of UPC MG on 1 core, which allows it to obtain almost linear speedups on up to 16 cores.

4.3 Performance of NPB Kernels on Shared Memory

Figure 3 shows NPB performance on the Superdome system. As in the hybrid memory figures, the left graphs show the kernels performance in MOPS and the right graphs show the speedups. MPI requires copying data on shared memory, and therefore could be considered less efficient than the direct access to shared memory of UPC and OpenMP. The following results do not support this hypothesis.

Regarding CG, all options show similar performance on up to 32 cores. However, for 64 and 128 cores UPC scalability is poor, whereas MPI achieves the best MOPS results. The poor performance of OpenMP on 1 core leads OpenMP to present the highest speedups on up to 64 cores, being outperformed by MPI on 128 cores, due to the lack of data locality support of NPB-OMP.

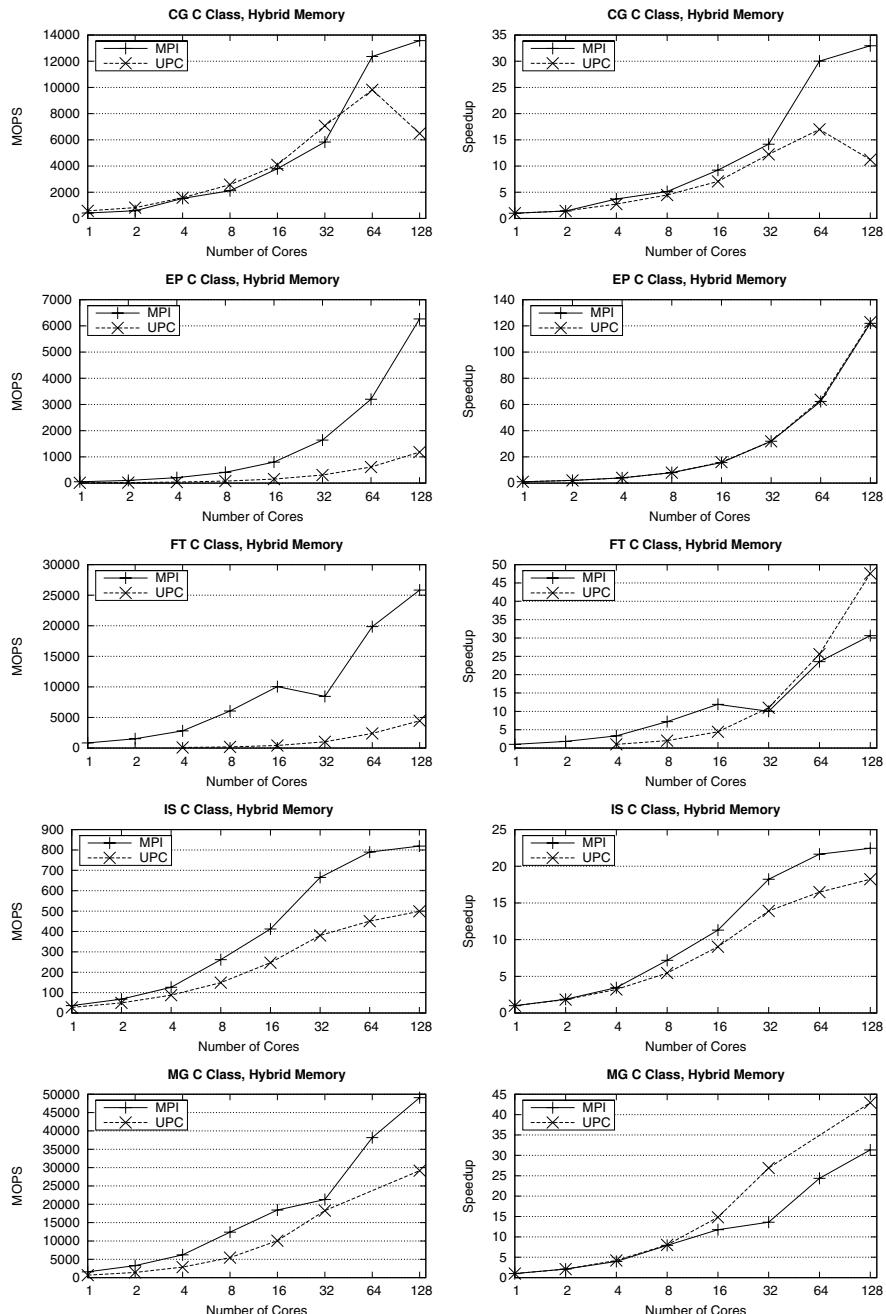


Fig. 2. Performance of NPB kernels on hybrid shared/distributed memory

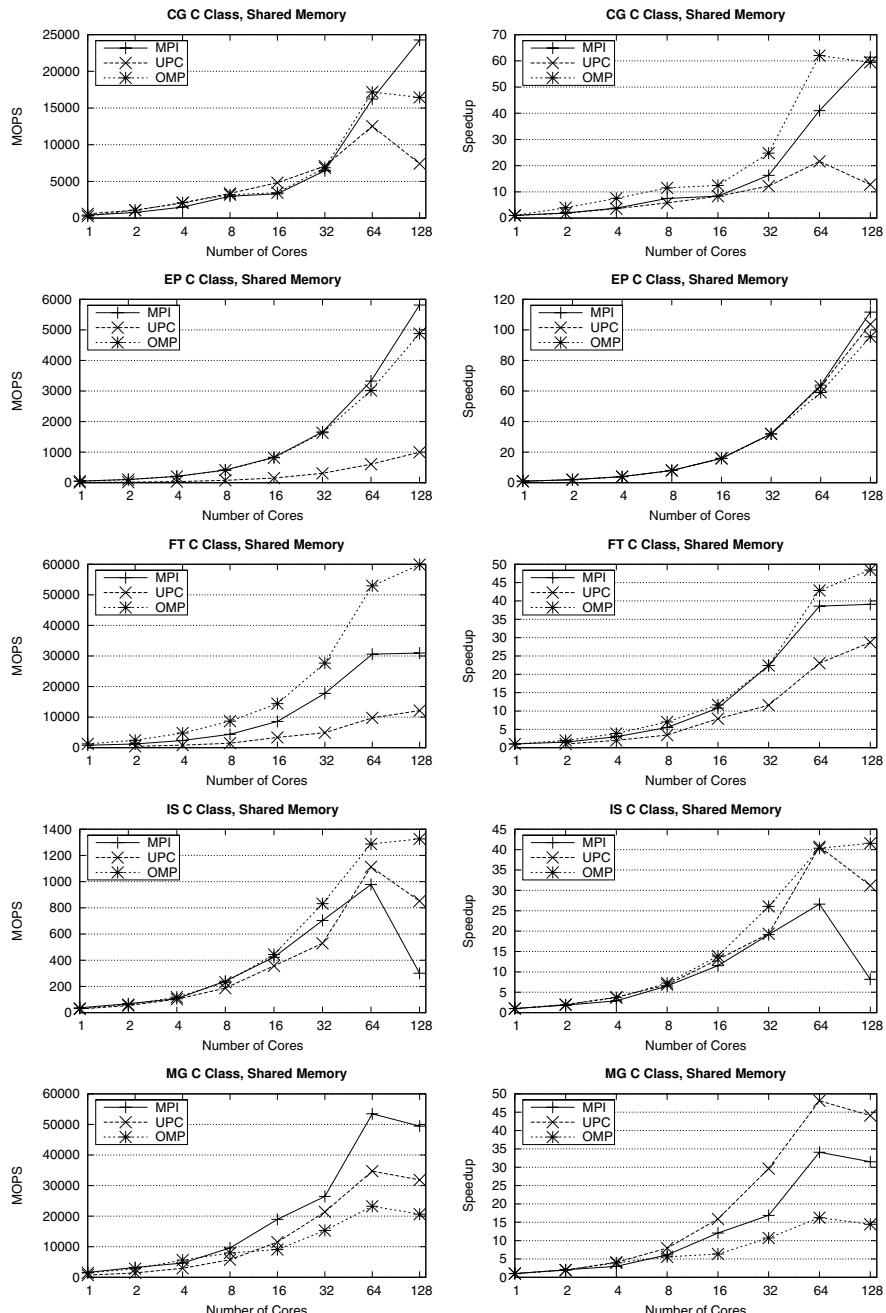


Fig. 3. Performance of NPB kernels on shared memory

As EP is an embarrassingly parallel code, the scalability shown is almost linear for MPI, UPC, and OpenMP, although MPI obtains slightly higher speedups, whereas OpenMP presents the lowest scalability. These results are explained by the efficiency in data locality exploitation of these three options. In terms of MOPS, UPC shows quite low performance, as discussed in subsection 4.2.

As FT is a communication-intensive code, its scalability depends on the performance of the communication operations. Therefore, OpenMP and MPI achieve high speedups, whereas UPC suffers from a less scalable exchange operation. The code structure of the OpenMP implementation allows more efficient optimizations and higher performance. Due to its good scalability OpenMP doubles MPI performance (in terms of MOPS) on 128 cores. UPC obtains the poorest performance.

IS is a communication-intensive code that shows similar performance for MPI, UPC and OpenMP on up to 32 cores, both in terms of MOPS and speedups, as the results on 1 core are quite similar among them. This fact can be partly explained by the fact that the IS kernels use the same backend compiler (icc). Regarding 64 and 128 cores results, OpenMP obtains the best performance and MPI the lowest, as the communications are the performance bottleneck of this kernel.

Regarding MG, MPI achieves better performance in terms of MOPS than UPC and OpenMP, whereas UPC obtains the highest speedups, due to the poor performance of this kernel on 1 core. OpenMP shows the lowest results, both in terms of MOPS and speedups.

5 Conclusions

This paper has presented an up-to-date performance evaluation of two well-established parallel programming models (MPI and OpenMP) and one emerging alternative (PGAS UPC) on two multicore scenarios. The analysis of the results obtained in the hybrid setup shows that MPI is the best performer thanks to its efficient handling of the data locality. However, UPC speedups are better for some benchmarks. Moreover, in some benchmarks the performance in the hybrid setup with 128 cores is not as high as expected, showing that for some workloads the network contention using a high number of cores may be a problem. This will be a bigger issue as the number of cores per node increases in the next years, where a higher network scalability will be required in order to confront this challenge.

Both MPI and UPC obtain better speedups in shared memory than in the hybrid setup up to 64 cores. However, for 128 cores all the options suffer from remote memory accesses and poor bidirectional traffic performance in the cell controller.

MPI usually achieves good performance on shared memory, although UPC and OpenMP outperform it in some cases. OpenMP speedups are generally higher than those of MPI due to its direct shared memory access, which avoids memory copies as in MPI. UPC, despite its direct shared memory access and data locality

support, suffers from its compiler technology and performs worse than the other two options. However, due to its expressiveness and ease of programming, is an alternative that has to be taken into account for productive development of parallel applications.

Acknowledgements. This work was funded by Hewlett-Packard and partially supported by the Spanish Government under Project TIN2007-67537-C03-02. We gratefully thank Jim Bovay and Brian Wibecan at HP for their valuable support, and CESGA for providing access to the Finis Terrae supercomputer.

References

1. MPI Forum, <http://www.mpi-forum.org> (last visited: June 2009)
2. OpenMP, <http://openmp.org> (last visited: June 2009)
3. Unified Parallel C, <http://upc.gwu.edu> (last visited: June 2009)
4. NAS Parallel Benchmarks,
<http://www.nas.nasa.gov/Resources/Software/npb.html>
(last visited: June 2009)
5. Rabenseifner, R., Hager, G., Jost, G.: Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes. In: Proc. of the 17th Euromicro Intl. Conf. on Parallel, Distributed, and Network-Based Processing (PDP 2009), Weimar (Germany), pp. 427–436 (2009)
6. Rabenseifner, R., Hager, G., Jost, G., Keller, R.: Hybrid MPI and OpenMP Parallel Programming. In: Mohr, B., Träff, J.L., Worringen, J., Dongarra, J. (eds.) PVM/MPI 2006. LNCS, vol. 4192, p. 11. Springer, Heidelberg (2006)
7. El-Ghazawi, T.A., Cantonnet, F., Yao, Y., Annareddy, S., Mohamed, A.S.: Productivity Analysis of the UPC Language. In: Proc. 3rd Workshop on Performance Modeling, Evaluation and Optimization of Parallel and Distributed Systems (PMEO 2004), Santa Fe (NM), pp. 1–7 (2004)
8. El-Ghazawi, T.A., Sébastien, C.: UPC Benchmarking Issues. In: Proc. 30th IEEE Intl. Conf. on Parallel Processing (ICPP 2001), Valencia (Spain), pp. 365–372 (2001)
9. El-Ghazawi, T.A., Cantonnet, F.: UPC Performance and Potential: a NPB Experimental Study. In: Proc. of the 15th ACM/IEEE Conf. on Supercomputing (SC 2002), Baltimore (MD), pp. 1–26 (2002)
10. Cantonnet, F., Yao, Y., Annareddy, S., Mohamed, A., El-Ghazawi, T.A.: Performance Monitoring and Evaluation of a UPC Implementation on a NUMA Architecture. In: Proc. of the 2nd Workshop on Performance Modeling, Evaluation and Optimization of Parallel and Distributed Systems (PMEO 2003), Nice (France), 274 (8 Pages) (2003)
11. Berkeley UPC, <http://upc.lbl.gov/> (last visited: June 2009)
12. Mallón, D.A., Taboada, G.L., Touriño, J., Doallo, R.: NPB-MPJ: NAS Parallel Benchmarks Implementation for Message Passing in Java. In: Proc. of the 17th Euromicro Intl. Conf. on Parallel, Distributed, and Network-Based Processing (PDP 2009), Weimar (Germany), pp. 181–190 (2009)
13. El-Ghazawi, T.A., Cantonnet, F., Yao, Y., Vetter, J.: Evaluation of UPC on the Cray X1. In: Proc. of the 47th Cray User Group meeting (CUG 2005), Albuquerque (NM), 10 Pages (2005)

14. Kayi, A., Yao, Y., El-Ghazawi, T.A., Newby, G.: Experimental Evaluation of Emerging Multi-core Architectures. In: Proc. of the 6th Workshop on Performance Modeling, Evaluation and Optimization of Parallel and Distributed Systems (PMEO 2007), Long Beach (CA), pp. 1–6 (2007)
15. Curtis-Maury, M., Ding, X., Antonopoulos, C.D., Nikolopoulos, D.S.: An Evaluation of OpenMP on Current and Emerging Multithreaded/Multicore Processors. In: Mueller, M.S., Chapman, B.M., de Supinski, B.R., Malony, A.D., Voss, M. (eds.) IWOMP 2005 and IWOMP 2006. LNCS, vol. 4315, pp. 133–144. Springer, Heidelberg (2008)
16. Finis Terrae Supercomputer, <http://www.top500.org/system/details/9500> (last visited: June 2009)
17. Taboada, G.L., Teijeiro, C., Touriño, J., Fraguela, B.B., Doallo, R., Mouríño, J.C., Mallón, D.A., Gómez, A.: Performance Evaluation of Unified Parallel C Collective Communications. In: Proc. of the 11th IEEE Intl. Conf. on High Performance Computing and Communications (HPCC 2009), Seoul (Korea), 10 Pages (2009)

Automatic Hybrid MPI+OpenMP Code Generation with `llc`^{*}

Ruymán Reyes, Antonio J. Dorta, Francisco Almeida,
and Francisco de Sande

Dept. de E.I.O. y Computación
Universidad de La Laguna, 38271-La Laguna, Spain
`{rreyes,ajdorta,falmeida,fsande}@ull.es`

Abstract. The evolution of the architecture of massively parallel computers is progressing toward systems with a hierarchical hardware design where each node is a shared memory system with several multi-core CPUs. There is consensus in the HPC community about the need to increment the efficiency of the programming paradigms used to exploit massive parallelism, if we are to make a full use of the advantages offered by these new architectures. `llc` is a language where parallelism is expressed using OpenMP compiler directives. In this paper we focus our attention on the automatic generation of hybrid MPI+OpenMP code from the `llc` annotated source code.

Keywords: MPI, OpenMP, `llc`, Hybrid parallel programming, multi-core.

1 Introduction

The eruption in the market of multi-core processors [1] is producing radical changes in the architecture of massively parallel computers. High-Performance Computing (HPC) platforms with many thousands of cores will be deployed in the coming years [2,3,4]. These new systems present a hierarchical hardware design: individual nodes in these systems will be heterogeneous multi-threading, multi-core systems with a deep memory hierarchy. The new HPC systems will deliver a sustained performance of one to two petaflops for many applications.

Although the computational power of new generation processors has been increasing continuously, the new systems still lack programmability. Application programmers will have to address several problems. Some of these problems are present now in the current architectures, but new ones are arising. The number of computing nodes is so huge that scalability is becoming a key issue. The number of control threads executing simultaneously will be much greater than it is nowadays. Resource sharing between threads on a much larger scale is another point that will require attention. Consequently, the code will need to exhibit a greater amount of parallelism.

* This work was partially funded by the EC (FEDER) and the Spanish MEC (Plan Nacional de I+D+I, TIN2008-06570-C04-03).

The intense changes in the newest architectures are arriving so fast that we are failing to take advantage of the increased power of the systems. The efficiency of the programming paradigms and tools we use to exploit massive parallelism must be increased [5]. Therefore, the efficient use of these new systems is an important challenge, and the scientists and engineers responsible for these applications, in general, are seldom experts in HPC, and usually do not want either to introduce new changes in their codes or to learn new programming paradigms. They need solutions in terms of effective automatic parallelization tools and libraries. When the target architecture is a heterogeneous cluster or a multi-core processor, current parallelization tools usually offer unsatisfactory results, mainly due to the continuous changes, complexity and diversity of these parallel platforms, which poses serious obstacles to the development of efficient and stable solutions.

In this paper, we aim to address the problem of automatic code generation for hybrid architectures from high level standard languages. Starting from an `11c` [6,7] source code, we can generate hybrid MPI/`OpenMP` code ready to be compiled in a hierarchical architecture using standard tools. As a noteworthy feature, we can mention that the generation process is quite simple and the code generated is as efficient as ad-hoc implementations. The rest of the paper is organized as follows: Section 2 presents the motivation of the work. In Sect. 3, through the use of an example, we illustrate the `11c` programming model and expose the simplicity of our method. Section 4 is dedicated to explaining the translation process that `11CoMP` performs. Several computational results obtained for `OpenMP`, MPI and `11c` codes are shown and discussed in Sect. 5. Finally, Sect. 6 offers conclusions and comments on future work.

2 Motivation

While MPI [8] and OpenMP [9] are nowadays the standard *de-facto* tools for programming distributed and shared memory architectures, respectively, both should evolve to adapt to the many-core era. According to [10], these new systems will support a variety of programming models, including the “MPI everywhere” model, which is the most common today. However, according to [11], whereas MPI has proven to be an excellent means of expressing program parallelism when nodes have a small number of cores, future architectures may make this a tough proposition. In particular, MPI does not provide the programmer with the means to conserve memory or to directly modify the code to benefit from resource sharing and to avoid its negative implications. Both authors agree, however, that in the current scenario with the presence of hybrid architectures, one possible step forward is to systematically combine MPI with OpenMP. Such a combined programming model is usually called a hybrid model [12,13,14], and is based on the use of multiple threads in each MPI process. Hybrid programming techniques have clear advantages over the classic approach to the new hardware. They allow for better control of application granularity, and they improve scalability and load balancing. The hybrid programming model has found mixed success to date, with many experiments showing little benefit, and others showing promise.

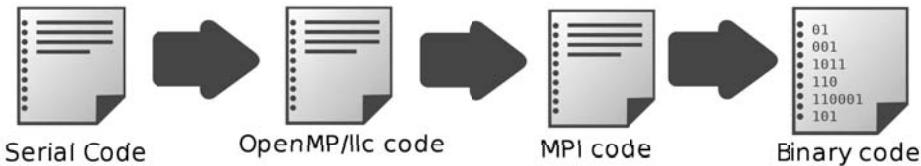


Fig. 1. 11CoMP code generation

The reason for the succeeding situations is related to the use of **OpenMP** within MPI programs where **OpenMP** is used to complement MPI, for example, by providing better support for load-balancing, adaptive computations or sharing large data tables.

11c [6,7] is a high-level parallel language that aims to exploit the best features of both MPI and **OpenMP**. **11c** follows the simplicity of **OpenMP** and avoids its well-known drawbacks. While in **OpenMP** we can start from a sequential code and parallelize it incrementally through the use of directives, the code cannot be ported to either distributed memory architectures or to the large new systems. In **11c** the code annotated with parallel directives is compiled with **11CoMP**, the **11c** compiler-translator. **11CoMP** produces an efficient and portable parallel source code, valid for both shared and distributed memory architectures. Figure 1 illustrates this process. Different directives have been designed and implemented in **11c** to support common *parallel constructs*: *forall*, *sections*, *pipelines* and *task queues*.

The main contribution of this work is the design and development of the new **11CoMP** backend that is able to produce hybrid code for multi-core architectures from the **11c** annotated source. Some consequences of this contribution are:

- Scientists and engineers can generate efficient code without learning new material, just by using the well-known **OpenMP** programming paradigm.
- Through the use of **11c**, many **OpenMP** codes can be ported to large multi-/many core systems without loss of efficiency.
- The new backend of the **11c** compiler represents an improvement with respect to its former version.
- **11c** is best suited for non-expert users as a semi-automatic tool for generating hybrid MPI+**OpenMP** parallel code.

In our approach, MPI handles inter-node communications, while **OpenMP** is used inside each SMP node. The preliminary results we present indicate a clear improvement over the previous implementation of the compiler when targeting the new architectures.

3 Example

The *OTOSP* (*One Thread is One Set of Processors*) model is the distributed memory computational model underlying the **11c** language. It is essential to know *OTOSP* in order to understand the **11c** implementation. The key reference on the model is [6].

Probably the first example of any parallel API is the “computing π ” program. The code in Listing 1 shows the calculation of π in 11c. This well-known code is based on the relationship: $\pi = \int_0^1 \frac{4}{1+t^2} dt$. Splitting the interval $[0 \dots 1]$ into N segments of size w , this integral can be approximated as the sum of the areas of the rectangles delimited by the central point of a segment t and the curve value at this point.

When compiled by 11CoMP, the loop (line 5) iterations are distributed among the processors. The clause **private** in line 3 is kept only for compatibility with OpenMP, since all the storages are private in the OTOSP computing model. The OpenMP clause **reduction** indicates that all the local values of variable **pi** have to be added at the end of the loop. This operation implies a collective communication among all processors in the OTOSP group and the updating of the variable with the result of the reduction operation. Since type analysis has not been included in 11CoMP, the type of reduction variable has to be specified in line 4.

Notice that an OpenMP code can be adapted to 11c with very few changes. In fact, all the OpenMP 2.5 directives and clauses are recognized by 11CoMP and therefore, we have three versions in the same code: sequential, OpenMP and 11c/MPI. We need only choose the proper compiler to obtain the corresponding binary.

```

1 h = 1.0 / N;
2 pi = 0.0;
3 #pragma omp parallel for private(t) reduction (+: pi)
4 #pragma 11c reduction-type (double)
5 for (i = 0; i < N; i++) {
6     x = (i + 0.5) * h;
7     pi = pi + 4.0 / (1.0 + t * t);
8 }
9 pi *= w;
```

Listing 1. Computing π in parallel with 11c

We have selected the “computing π ” code due to its simplicity to show the main features of 11c. Obviously, algorithms like this one, with a reduced amount of communications, are those best suited to attain optimal performance when parallelized with 11c. Apart from the *parallel for*, 11c also provides support for the most common *parallel constructs*: *sections*, *pipelines* and *farms* ([7,15]).

The code in Listing 2 shows part of the hybrid code automatically generated by 11CoMP for the “computing π ” code.

4 The Translation Process

Studying a wide set of parallel applications, we can conclude that most of them can be classified according to the parallelization paradigm used [16]. Moreover,

```

1 #pragma omp parallel for private (t) reduction (+ : pi)
2   for(i = (0) + llc_F[llc_grp_save]; i < (0) + llc_F[
3     llc_grp_save] + LLC_PROCSGRP(llc_grp_save); i++) {
4   {
5     x = (i + 0.5) * h;
6     pi = pi + 4.0 / (1.0 + t * t);
7   };
8 . . .
9 if (LLC_NAME == 0) {
10   for (llc_i = 1; llc_i < LLC_NUMPROCESSORS; llc_i++) {
11     MPI_Recv (llc_buf, llc_buf_size, MPI_BYTE, llc_i,
12       LLC_TAG_REDUCE_DATA,
13       *llc_CurrentGroup, &llc_status);
14     llc_buf_ptr = llc_buf;
15     pi += (*((double *) llc_buf_ptr));
16     llc_buf_ptr += sizeof(double);
17   }
18   llc_buf_ptr = llc_buf;
19   memcpy (llc_buf_ptr, &pi, sizeof(double));
20   llc_buf_ptr += sizeof(double);
21   MPI_Bcast (llc_buf, llc_buf_size, MPI_BYTE, 0, *
22     llc_CurrentGroup);
23 } else {
24   llc_buf_ptr = llc_buf;
25   memcpy (llc_buf_ptr, &pi, sizeof(double));
26   llc_buf_ptr += sizeof(double);
27   MPI_Send (llc_buf, llc_buf_size, MPI_BYTE, 0,
28     LLC_TAG_REDUCE_DATA, *llc_CurrentGroup);
29   MPI_Bcast (llc_buf, llc_buf_size, MPI_BYTE, 0, *
30     llc_CurrentGroup);
31   llc_buf_ptr = llc_buf;
32   memcpy (&pi, llc_buf_ptr, sizeof(double));
33   llc_buf_ptr += sizeof(double);
34 }
```

Listing 2. The llc translation of the “computing π ” code

the parallel code itself (data distribution, communications, etc.) is quite similar in applications following the same paradigm. llCoMP takes advantage of this feature to generate code. We have named these portions of reusable code *patterns*, and llCoMP uses two kinds of *patterns* in order to process the llc parallel constructs.

Static patterns are the *pure* parallel code and they do not depend directly on the application, but rather on the parallelization paradigm specified using the llc and/or OpenMP constructs. These codes are implemented in llCoMP using the target language (i.e. MPI) and they encode operations like initialization of the parallel environment, resources distribution, data communications, load

balancing, etc. The compiler adapts the *static patterns* to a specific translation using special tags in the pattern that the compiler fills with information coming from the source code directives.

To ease the compiler code maintenance, static patterns have been split into several files, each file implementing a specific stage of the entire pattern: initialization, execution, communication and/or finalization. In addition to the general case, 11CoMP implements specialized code for some common situations. When it detects any of these situations, it uses the optimized code. For this reason, each of the stages can be supported through different files. The good performance delivered for the general case can be improved if an optimization is detected. As a result, each paradigm and its static pattern is implemented by several text files.

The static patterns need some extra code to work. This additional code handles the operations on data structures needed to build the translation, i.e. management of buffers used during communications. This code is sequential and specific to each application and therefore cannot be embedded in the static patterns. 11CoMP uses *dynamic patterns* to produce this complementary code. The dynamic pattern code is generated by the compiler during the compilation process and stored in temporary files. The static patterns use special marks to indicate the compiler the right point where each temporary file has to be inserted to produce the target code. The dynamic patterns are carefully tuned to optimize parallel performance. For instance, data packing/unpacking greatly reduces communications.

It is through a combination of both static and dynamic patterns that 11CoMP produces the target code. The compilation process is carried out by 11CoMP without user intervention. All the information needed is gathered from the parallel constructs in the source code, and from it, the compiler selects the best combination of static and dynamics patterns, including any available optimization, to build the target code.

The OTOSP computational model guarantees that each processor owns at least a part of the main dataset. We are therefore able to add a new level of parallelism by simply re-parallelizing the annotated loop. The tags in the static patterns allow for the injection of this second parallel level to generate hybrid code. In this nested level, we use a shared memory approach to parallelization, using pure OpenMP code for the previously 11c-annotated loop. 11CoMP builds the OpenMP code to be injected from the source code.

For the distribution between MPI and OpenMP threads, we have chosen to use one MPI process per computation node, and shared memory threads between the cores of the node. According to the classification in [17], we use a *hybrid masteronly* scheme. This allows us to take advantage of the shared memory in the cores, in particular of their shared caches, thus increasing the performance of the loops.

Since our target systems in this work are multi-node clusters, the most efficient approach is to use one processor to communicate over the internode network, given a single processor's capability to use the network completely, as shown in [14].

llCoMP generated code handles the MPI communications and synchronization of data between nodes. This code would not be enhanced by using more than one CPU, as it consists mainly of synchronization routines.

5 Experimental Results

In this Sect. we present the preliminary results of our implementation by showing the performance obtained for 4 algorithms in two different multi-core systems. The algorithms we used are the “computing π ” code introduced in Sect. 3, a Molecular Dynamic (MD) code, the Mandelbrot Set computation and the Conjugate Gradient (CG) algorithm. The MD code is an **llc** implementation of the velocity Verlet algorithm [18] for Molecular Dynamics simulations. It employs an iterative numerical procedure to obtain an approximate solution whose accuracy is determined by the time step of the simulation. The Mandelbrot Set is the convergence domain of the complex series defined by $Z_n = Z_{n-1}^2 + C$. The area of the set is an open question in Mathematics. Using a Monte-Carlo method, the algorithm computes an estimation of the set area. The CG algorithm is one of the kernels in the NAS Parallel Benchmarks [19]. CG uses the inverse power method to find an estimate of the largest eigenvalue of a symmetric positive definite sparse matrix with a random pattern of nonzeros. The algorithm performs a certain number of iterations, and for each of them, the solution z to the linear system of equations $Az = x$ is approximated using the conjugate gradient (CG) method, where A denotes the sparse matrix of order N . The source code for all four algorithms is available at the **llc** project home page [15].

We evaluated the above algorithms using 2 different systems, named Tajinaste and Verode. Both systems were run on Linux CentOS 5.3 with kernel 2.6. The codes were compiled using `gcc 4.1` and the OpenMPI MPI implementation. Tajinaste is an IBM cluster with 14 dual core Opteron processors interconnected with a Gigabit Ethernet network. The results we present for this cluster are limited to 12 cores to guarantee exclusive use of the system. Verode is a two quad-core Opteron system. In this system, we used the same OpenMPI implementation with the shared memory intercommunication device.

Figure 2 shows the results for 4 different implementations of the Mandelbrot set computation algorithm in Tajinaste. The meaning of the labels in the Fig. corresponds with the following implementations:

- *MPI*: a pure MPI implementation.
- *LLC-MPI*: **llCoMP** generates pure MPI code.
- *LLC-HYB*: **llCoMP** generates hybrid OpenMP-MPI code.
- *HYB*: An *ad-hoc* hybrid implementation.

In every case the code iterates over 16368 points in the complex plane. This algorithm is usually taken as an academic example of a situation with an intense load imbalance: each complex point analyzed requires a large-varying number of iterations of the main loop to converge. In this situation, Fig. 2 reflects the

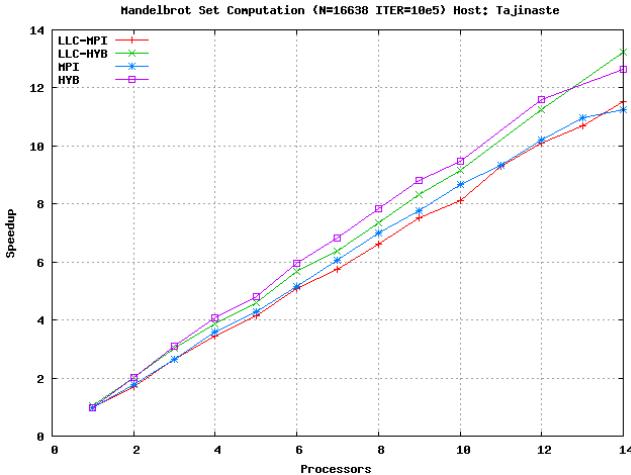


Fig. 2. Speed-up of the Mandelbrot set computation on Tajinaste

Table 1. Speedup for the MD and computing π algorithms in Tajinaste

#Proc.	MD		π	
	MPI	HYB	MPI	HYB
SEQ	38.590		9.184	
1	0.984	0.950	0.994	0.997
2	1.967	1.903	1.993	1.989
3	2.948	2.849	2.986	2.968
4	3.929	3.776	3.975	3.890
6	5.813	5.600	5.949	5.791
8	7.637	7.316	7.881	7.744
9	7.558	8.533	8.775	8.477
12	9.715	11.303	11.681	11.482

benefit of the *LLC-HYB* approach in relation to the *LLC-MPI* case. The parallel execution time improvement is 17% percent for 14 processors. The *llc* versions exhibit results comparable to their *ad-hoc* counterparts, with a much smaller coding effort.

Tables 1 and 2 show the results for the MD, computing π and CG algorithms in Tajinaste and Verode, respectively. In both tables, row *SEQ* shows the time (seconds) corresponding to the sequential version of the code, while the remaining data correspond to the speedups achieved by *llc* implementations. Column *MPI* corresponds to pure MPI code generation by *llCoMP* and column *HYB* reflects the results using the new *llCoMP* backend. In the case of the computing π and MD algorithms, no significant improvement is evident. Nevertheless, in the case of the CG code, again the results show the benefit achieved with the hybrid

Table 2. Speedup for the MD, computing π and CG algorithms in Verode

#Proc.	MD		π		CG	
	MPI	HYB	MPI	HYB	MPI	HYB
SEQ	38.590		9.184		198.941	
1	0.817	0.823	0.997	1.004	1.003	0.988
2	1.613	1.624	1.982	1.987	1.681	1.792
3	2.421	2.465	2.959	2.953	2.123	2.566
4	3.226	3.247	3.956	3.970	2.334	3.209
5	4.027	4.089	4.936	4.942	2.471	3.835
6	4.857	4.929	5.944	5.944	2.448	4.372
7	5.670	5.754	6.935	6.831	2.579	4.772
8	6.451	6.586	6.545	6.862	2.423	5.247

code. The reason behind this improvement is the granularity of the algorithm: the CG code parallelizes several loops, some of them used for initializations. This fine-grain parallelism penalizes the pure MPI code version, while the hybrid translation takes advantage of it.

6 Conclusions and Future Work

We have explored the potential of using **11c** and its compiler as a simple method for producing hybrid MPI/OpenMP code to target new hierarchical multi/many-core parallel architectures. We are curious as to whether our methodology could also be applied to other compilers or languages. In our approach, the use of direct generation of hybrid code for the translation of the OpenMP directives conjugates simplicity with reasonable performance. Our code generation scheme, while simple, is also reasonably efficient.

Through the use of **11c**, most OpenMP codes can be directly ported to the new hierarchical architectures without loss of efficiency. We believe that **11c** is a good choice for non-expert users to exploit massive parallelism. The results we have shown in this work represent a clear improvement over the previous implementation of **11CoMP**. Combining the above facts leads us to believe that the hybrid model of programming is deserving of more thorough study.

Work in progress in our project includes the following issues:

- Enlarge the computational experiments with additional algorithms that benefit more fully from our approach.
- Test our implementations on machines with a larger number of computational nodes.
- Study the applicability of our method to mixed multi-core CPU/GPU architectures.

Acknowledgments

We wish to thank the anonymous reviewers for their suggestions on how to improve the paper.

References

1. Olukotun, K., Nayfeh, B.A., Hammond, L., Wilson, K., Chang, K.: The case for a single-chip multiprocessor. *SIGPLAN Not.* 31(9), 2–11 (1996)
2. Asanovic, K., Bodik, R., Catanzaro, B.C., Gebis, J.J., Husbands, P., Keutzer, K., Patterson, D.A., Plishker, W.L., Shalf, J., Williams, S.W., Yelick, K.A.: The landscape of parallel computing research: A view from Berkeley, Tech. Rep. UCB/EECS-2006-183, EECS Department, University of California, Berkeley (December 2006), <http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html>
3. Dongarra, J., Gannon, D., Fox, G., Kennedy, K.: The impact of multicore on computational science software. *CTWatch Quarterly* 3(1) (2007)
4. Borkar, S., Dubey, P., Kahn, K., Kuck, D., Mulder, H., Pawłowski, S., Rattner, J.: Platform 2015: Intel(r) processor and platform evolution for the next decade, *Technology@Intel Magazine*, http://tbp.berkeley.edu/research/cmp/borkar_015.pdf
5. Bosilca, G.: The next frontier. In: Lastovetsky, A., Kechadi, T., Dongarra, J. (eds.) EuroPVM/MPI 2008. LNCS, vol. 5205, p. 1. Springer, Heidelberg (2008)
6. Dorta, A.J., González, J.A., Rodríguez, C., de Sande, F.: 11c: A parallel skeletal language. *Parallel Processing Letters* 13(3), 437–448 (2003)
7. Dorta, A.J., Lopez, P., de Sande, F.: Basic skeletons in 11c. *Parallel Computing* 32(7–8), 491–506 (2006)
8. Message Passing Interface Forum, MPI: A Message-Passing Interface Standard, University of Tennessee, Knoxville, TN (1995), <http://www.mpi-forum.org/>
9. OpenMP Architecture Review Board, OpenMP C/C++ Application Program Interface (March 2002), <http://www.openmp.org/specs/mp-documents/cspec20.pdf>
10. Gropp, W.D.: MPI and hybrid programming models for petascale computing. In: Lastovetsky, A., Kechadi, T., Dongarra, J. (eds.) EuroPVM/MPI 2008. LNCS, vol. 5205, pp. 6–7. Springer, Heidelberg (2008)
11. Chapman, B.: Managing multicore with openMP (Extended abstract). In: Lastovetsky, A., Kechadi, T., Dongarra, J. (eds.) EuroPVM/MPI 2008. LNCS, vol. 5205, pp. 3–4. Springer, Heidelberg (2008)
12. Smith, L., Bull, M.: Development of mixed mode MPI OpenMP applications. *Scientific Programming* 9(2–3), 83–98 (2001)
13. Cappello, F., Etiemble, D.: MPI versus MPI+openMP on IBM SP for the NAS benchmarks. In: Proceedings of Supercomputing 2000 (CD-ROM). IEEE and ACM SIGARCH, Dallas (2000)
14. Rabenseifner, R.: Hybrid parallel programming on hpc platforms. In: Proc. of the Fifth European Workshop on OpenMP (EWOMP 2003), Aachen, Germany, pp. 185–194 (2003)
15. 11c Home Page, <http://11c.pcg.ul1.es>
16. Cole, M.: Algorithmic Skeletons: structured management of parallel computation, Monograms. Pitman/MIT Press, Cambridge (1989)

17. Rabenseifner, R., Hager, G., Jost, G.: Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. In: Proc. of the 17th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP 2009), Weimar, Germany (2009)
18. Swope, W.C., Andersen, H.C., Berens, P.H., Wilson, K.R.: A computer simulation method for the calculation of equilibrium constants for the formation of physical clusters of molecules: Application to small water clusters. *Journal of Chemical Physics* 76, 637–649 (1982)
19. Bailey, D.H., Barszcz, E., Barton, J.T., Browning, D.S., Carter, R.L., Dagum, D., Fatoohi, R.A., Frederickson, P.O., Lasinski, T.A., Schreiber, R.S., Simon, H.D., Venkatakrishnan, V., Weeratunga, S.K.: The NAS parallel benchmarks. *The International Journal of Supercomputer Applications* 5(3), 63–73 (1991),
<http://www.nas.nasa.gov/NAS/NPB/>

Optimizing MPI Runtime Parameter Settings by Using Machine Learning*

Simone Pellegrini, Jie Wang, Thomas Fahringer, and Hans Moritsch

University of Innsbruck – Distributed and Parallel Systems Group

Technikerstr. 21A, 6020 Innsbruck, Austria

{spellegrini,wangjie,tf,hans}@dps.uibk.ac.at

Abstract. Manually tuning MPI runtime parameters is a practice commonly employed to optimise MPI application performance on a specific architecture. However, the best setting for these parameters not only depends on the underlying system but also on the application itself and its input data. This paper introduces a novel approach based on machine learning techniques to estimate the values of MPI runtime parameters that tries to achieve optimal speedup for a target architecture and any unseen input program. The effectiveness of our optimization tool is evaluated against two benchmarks executed on a multi-core SMP machine.

Keywords: MPI, optimization, runtime parameter tuning, multi-core.

1 Introduction

Existing MPI implementations allow the tuning of runtime parameters providing system administrators, end-users and developers the possibility to customise the MPI environment to suit the specific needs of applications, hardware or operating environments. An example is the opportunity to change the semantics of point-to-point communications in relation to the size of the message being transmitted. According to a *threshold* (runtime parameter) value the library can use an *eager* protocol when small messages are exchanged or the more expensive *rendezvous* method for larger messages. A *default* setting, designed to be a good compromise between functionalities and performance, is nevertheless provided by the MPI library to allow easy deployment of MPI applications. Open MPI provides an entire layer – called Modular Component Architecture (MCA) [1] – with the purpose of providing a simple interface to tune the runtime environment. The current development version of Open MPI has several hundred MCA runtime parameters. This large number of configurable parameters makes the manual tuning of the Open MPI environment particularly difficult and challenging.

Considering a subset of tunable runtime parameters, determining the values that optimize the performance of an MPI program on a *target* system is not trivial and can depend on several factors (e.g. number of nodes in a cluster, type

* This work is funded by the Tiroler Zukunftsstiftung under contract nr. P7030-015-024.

of network interconnection and layout and amount of shared cache in a single node). This optimization problem is comparable with the selection of the *optimization sequence* (usually expressed as a vector of flags) performed by modern compilers. As the best sequence that reduces the execution time of a program strongly depends on the underlying architecture, and because the definition of effective cost models is not always feasible, *feedback-driven iterative* techniques (also known as *iterative compilation*) have been employed [2]. The number of possible *settings* (or *configurations*) of runtime parameters, and thus the corresponding *optimization space*, is too large to be manually explored. For these reasons, tools have been developed in order to *automate* the process of evaluating the execution of an MPI program considering numerous combinations/settings of these parameters in order to find the one with the best execution time [3]. Even if the process is automated, the exhaustive exploration of the optimization space could be very expensive in terms of time; and it makes sense only if the cost of the optimization phase can be amortised over many runs of the program.

In this paper we introduce a mechanism to estimate *optimal* runtime parameter settings for MPI programs running on any hardware architecture. The approach is based on *machine learning* (ML) techniques which use specific knowledge of the underlying system (acquired during an *off-line* training phase) to build a *predictor* capable of estimating the best setting for a subset of runtime parameters for any unseen MPI input program. A program is described by a set of *features* extracted both *statically* by analysing the source code and *dynamically* by profiling *one* run of the program. Although our approach is general, we focus on the optimization of MPI applications running on a multi-core SMP node as the increased deployment of multi-core systems in clusters makes intra-node optimizations fundamental in the overall application performance [4]. As a result of our work we show that despite the information used to describe a program being rather simple, the estimated optimal settings of the runtime parameters always outperform the *default* one and achieve, on average, around 90% of the available performance improvement. Two ML algorithms (i.e. decision trees and artificial neural networks) are used, and the accuracy of their prediction is evaluated against two different benchmarks: the Jacobi relaxation method [5] and the Integer Sort (IS) benchmark of the Nas Parallel Benchmarks suite [6].

The rest of the paper is organised as follows. Section 2 depicts the impact of runtime parameters. In Section 3, we introduce our machine learning framework, decision trees and artificial neural networks. Section 4 presents the results and accuracy of the optimal runtime parameter values prediction for the two benchmarks. Section 5 discusses related work, and Section 6 concludes the paper with some brief considerations and an outlook to future work.

2 Impact of Runtime Parameter Settings

The motivation behind our work is that the correct setting of runtime parameters within an application and an architecture can result in a valuable performance gain of the program itself. Exploring the entire optimization space for MPI applications

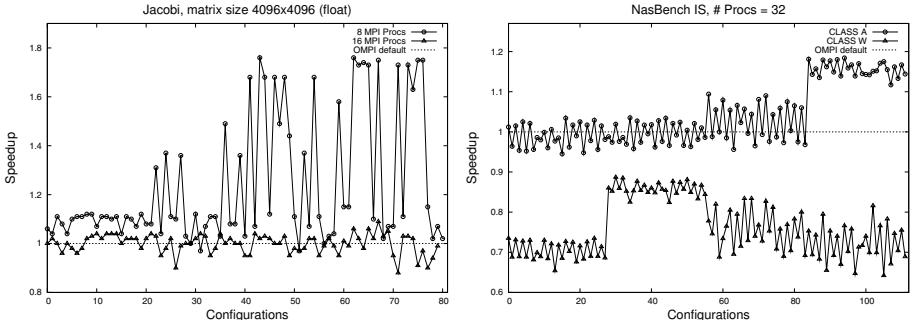


Fig. 1. Variation of the speedup of the Jacobi method (on the left) and IS benchmark (on the right) against combinations of the runtime parameters using different communicator (for Jacobi) and problem (for IS) sizes

requires the execution of the same program using a large number of different configurations which exponentially increases with the amount of runtime parameters considered. Formally, we define a *configuration* (C_j) of a set of n runtime parameters (p_1, p_2, \dots, p_n) as a vector $C_j = (c_{1,j}, c_{2,j}, \dots, c_{n,j})$ where $c_{i,j}$ is the value (a buffer size or a threshold/flag value) associated with the parameter p_i .

According to the particular program to optimize and the underlying architecture a *pre-selection* of *interesting* tunable runtime parameters can be done. For example, Open MPI's MCA provides an entire framework for the optimization of collective operations (i.e. `col11`). Although this module offers several parameters to select which kind of algorithm to use or the maximum depth of the tree employed to perform reduction operations, it is not useful if the target application does not contain collective operations.

Fig. 1 depicts the impact of the variation of 4 runtime parameters on the performance of the Jacobi method and IS benchmark. The graph on the left compares the speedup achieved by the Jacobi method (executed with 8 and 16 MPI processes) considering several configurations. As the values of 4 different runtime parameters are varied across configurations, vectors of parameter values have been numbered in order to make a 2-dimensional representation possible. For the Jacobi, configurations that achieve a high speedup using 8 processors turn to be very inefficient when 16 processes are used (and vice-versa). In addition to the number of processes being used, the problem size plays an important role. The graph on the right in Fig. 1 shows the speedup achieved by the IS benchmark using two different problem sizes (i.e. classes A and W) while varying the configuration of the runtime parameters. None of the tested configurations outperforms the Open MPI default for the small problem size (class W). However, for the larger problem size (class A) some configurations achieve a performance gain of up to 18%.

2.1 Selection of Runtime Parameters

As the focus of our work is the performance optimization of MPI programs running on a multi-core SMP machine, the optimization space can be reduced

by selecting those parameters whose effects are relevant in a shared memory system. Preliminary experiments have been conducted in order to determine the set of parameters with high impact on the performance of MPI application. Four parameters have been selected known to have a significant impact on point-to-point and collective operations:

- sm_eager_limit:** threshold (in byte) which decides when the MPI library switches from *eager* to *rendezvous* mode (for **Send** and **Receive** operations)
- mpi_paffinity_alone:** flag used to enable *processor affinity* (if enabled (1), each MPI process is exclusively bound to a specific processor core)
- coll_sm_control_size:** buffer size (in byte) chosen to optimize collective operations (usually sized according to the size of the shared cache)
- coll_tuned_use_dynamic_rules:** flag which enables *dynamic* optimization of collective operations.

Considering a threshold value and a buffer size in the range of $[2^9, \dots, 2^{28}]$ (a total of 20 values if power of 2 values are considered) and 2 possible values for the flags, the number of possible configurations generated by the variation of these 4 parameters is $20^2 * 2^2 = 1600$. The use of automatic iterative techniques for finding the best configuration in this four-dimensional space for an application with an execution time of 5 minutes would take around 5.5 days.

3 Using Machine Learning to Estimate Optimal Configurations

The challenge is to develop a predictive model that analyses any MPI *input program*, and according to the gained knowledge of how the *target architecture* behaves, determines the *values* – for a set of pre-defined runtime parameters – which achieve *optimal* speedup. As the prediction depends on the characteristics of the input program and its input data, we define five *program features* (see Table 1), extracted both *statically* and *dynamically*, which allow MPI programs to be *classified*. Furthermore, the target system has to be described in order to obtain a configuration of runtime parameters which is optimal for the underlying architecture. The behaviour of a system is nevertheless non-trivial to model as several variables (e.g. number of cores, amount of private/shared cache, memory and network latency/bandwidth) must be considered; additionally, even with the complete knowledge of the system, interactions between components can not be described by simple analytical models.

To solve this problem, we use machine learning (ML) techniques; the main idea is illustrated in Fig. 2. A set of training programs (described by *program features*) is executed on the *target architecture* using several configurations of the selected runtime parameters (see Section 2.1). Each program of the training set is executed against a representative set of parameter configurations; the execution time is measured and *compared* with the one obtained using the Open MPI's *default* parameter values (*baseline*). The resulting *speedup*, together with the program features and the current configuration, is stored and used to train a

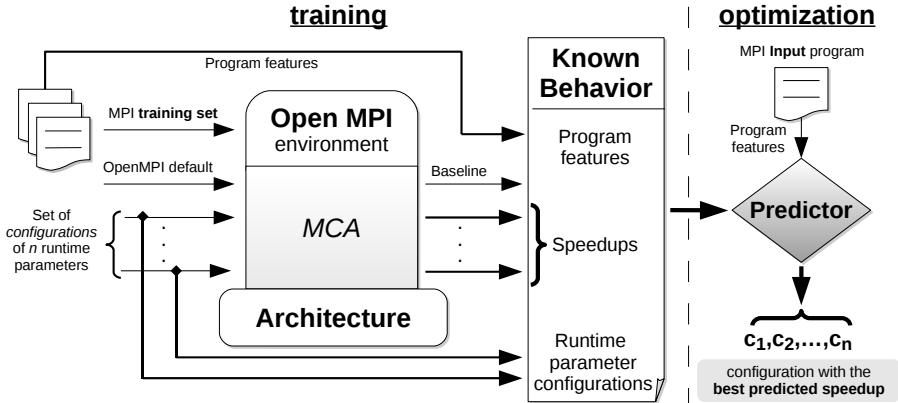


Fig. 2. Training and use of the predictor

predictor. This phase, also known as *training* or *learning* phase, is executed *offline* and it must be repeated every time the target architecture changes (or when a prediction for a different system is needed). During the optimization phase, when an *unseen* MPI *input program* is presented to the predictor, its program features are extracted and used to determine the configuration which optimizes the execution time of the *input program*.

We use two standard machine learning techniques, *Decision Trees* [7] and *Artificial Neural Networks* [8]. The construction of the two ML-based predictors is compared and their accuracy is evaluated using two different benchmarks (i.e. Jacobi method and IS). This section continues by describing the three key phases of our approach: (*i*) MPI program characterisation, (*ii*) generation of training data and (*iii*) prediction model construction.

3.1 Characterising MPI Programs

A program is described by a set of *features* which is extracted both *statically* and *dynamically*. As the runtime parameters only affect the communication between processes, a program is described by *analysing* its MPI statements. The *communication pattern*, amount of *exchanged data* and the *communicator size* are extracted from any MPI program by instrumenting and tracing its execution *once*. Table 1 displays the features used to characterise the behaviour of MPI programs.

The set of program features considered in this paper is relatively small as our goal is to evaluate the suitability of machine learning techniques for the estimation of MPI runtime parameter values. Furthermore, as the training set is designed in accordance with program features (see Section 3.2), this characterisation enables a fast training phase. In Section 4 we show that even if the program characterisation is rather simple, the predicted configurations of the runtime parameters achieve, on average, 90% of the available performance improvement.

Table 1. MPI program features

<code>coll_ratio</code>	ratio between the amount of collective operations and the total number of communication operations
<code>coll_data</code>	average amount of data exchanged in collective operations
<code>p2p_ratio</code>	ratio between the amount of point-to-point operations and the total number of communication operations
<code>p2p_data</code>	average amount of data exchanged in point-to-point operations
<code>comm_size</code>	number of processes involved in the communication

3.2 Generating Training Data

According to the defined program features, we designed a training set of micro-benchmarks with the purpose of generating training data. The training-set is composed of three micro-benchmarks: `p2p`, `coll` and `mixed`.

- `p2p`: measures the execution time of a *synchronous send/recv* operation between two processes using different message sizes
- `coll`: measures the execution time of a *collective* operation, `alltoall`, varying the communicator and message sizes
- `mixed`: mixes point-to-point and collective operations varying the message and communicator sizes

The three training programs have been executed on our target architecture using several configurations of the 4 selected runtime parameters. Each configuration was executed 10 times, and the mean value of the measured execution time has been used to calculate the speedup. The resulting set of training data contains around 3000 instances.

3.3 Predictor Model Construction

The training data is stored in a repository (also called *known behaviour* in Fig. 2), and each entry is encoded as a vector of 10 values:

$$\{(f_{1,j}, \dots, f_{5,j}), (c_{1,k}, \dots, c_{4,k}), \text{speedup}\}$$

Each vector represents the *speedup* achieved by one of the training programs whose i -th feature values $f_{i,j}$ and was executed using $c_{i,k}$ as the value of the i -th runtime parameter. The known behaviour is used to build two predictor models based on different machine learning algorithms.

Decision Trees. We use `REPTree`, a fast *tree learner* which uses reduced-error pruning and can build a regression tree [7], to train a predictor from the repository data. The speedups, together with program features ($f_{i,j}$) and runtime parameter ($c_{i,k}$) values are used to model the input/output relationship in the form of *if-then* rules. The constructed decision tree is able to *estimate* the speedup of

a program (described by its feature F_i) running with a specific configuration C_j of the runtime parameters. A formal representation can be given as follows: let dt be the decision tree model, the predicted speedup $S = dt(F_i, C_j)$. For a specific input program F_x , the best configuration of the runtime parameters C_{best} would be the one with the highest predicted speedup $S_{max} = dt(F_x, C_{best})$.

Artificial Neural networks. *Artificial Neural Networks* (ANNs) [8] are a class of machine learning models that can map a set of input values to a set of target values [9]. We use ANNs because are robust to noise and they have been successfully used in modeling both linear and non-linear *regression problems* [9]. As opposed to decision tree model, ANNs are trained only using the configurations of runtime parameters with the best speedup for each distinct vector of program features. Its input/output relationship can be described as follows: let ann be the ANN model, $C_{best} = ann(F_x)$. A three-layer feed-forward back-propagation network [8] is used and the ANN structure with the best performance for our problem is chosen as below. The hidden layer of the network contains 10 neurons and the the transfer function is *hyperbolic tangent sigmoid*, for the output layer the *logarithmic sigmoid* function has been used.

4 Experimental Results

In this section we evaluate the accuracy of the two predictors based respectively on decision trees and neural networks for the Jacobi method and IS benchmark. To run the experiments we used a 32-cores Sun X4600 M2 server with AMD Opteron 8356 (Barcelona) processors. In this system there are 8 sockets with one processor each. Each chip contains, there are four cores – with private L1 and L2 cache (512 KB) – which share an L3 cache of 2 MB. The system runs CentOS version 5 (kernel 2.6.18) 64 bits and the Open MPI version used is 1.2.6 (compiled with gcc-4.1.2). We use Weka’s implementation of **REPTree** [7] and Matlab’s Neural Network Toolbox [10] for training the ANN-based model.

4.1 Jacobi Relaxation Method

The considered Jacobi implementation uses a 2-dimensional matrix split where the rows are exchanged synchronously between neighbour processes. At the end of each iteration, a *reduce* operation is performed to calculate the residual error. The values of the program features associated to the Jacobi implementation are the following. As in each iteration a process performs 4 point-to-point operations (i.e. 2 sends and 2 receives) and only one collective operation (i.e. reduce) **coll_ratio** and **p2p_ratio** can be statically evaluated respectively as $4/5 = 0.8$ and $1/5 = 0.2$ (which means that 80% of the communication time is spent in point-to-point operations and the remaining 20% in collective ones). The value **p2p_data** is the size of the exchanged row and thus depends on the problem size. The Jacobi method has been executed considering different matrix sizes and varying the number of MPI processes (8, 16 and 32).

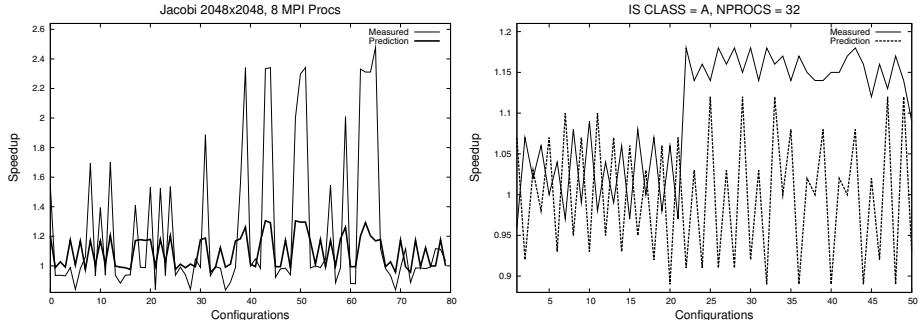


Fig. 3. Predicted and measured speedup using REPTree

The result of the predictor based on decision trees is depicted in Fig. 3. The graph on the left compares the predicted and measured speedup for the Jacobi method using 8 processes (because of space limits predictions for 16 and 32 processes are not shown). As stated in the previous sections, the decision tree-based predictor is capable of estimating the speedup of a program running with a specific configuration of the runtime parameters. In order to obtain the *best setting*, the predictor is queried several times using different values of the runtime parameters. The configuration with the highest predicted speedup contains the best runtime parameter settings for the Jacobi method on the target architecture. For example, in Fig. 3 the highest predicted speedup is 1.305 which corresponds to configuration number 46 (the respective values of the runtime parameters are $\{262144, 1, 262144, 0\}$); its measured speedup is 2.3. The best possible speedup achieved by Jacobi is for configuration number 68 (i.e. $\{32768, 1, 2097152, 1\}$), with a value of 2.48. Therefore, in this case, the trained decision tree predicts a configuration that gives 95% of the maximum possible performance improvement. The structure of the decision tree built from training data is depicted in Fig. 4. Because of space limit, the model has been simplified reducing the depth from 11 to 3. The impact of a specific feature/parameter on the value of speedup (represented in leafs) depends from its depth in the tree.

The results of the neural network-based predictor are depicted in Table 2. The table shows Jacobi's features vector followed by the configuration of the runtime parameters that gives the best speedup (for the particular feature vector) followed by the predicted best configuration with the respective *measured* speedup. The speedup achieved by the predicted configurations achieve on average 94% of the available speedup.

4.2 Integer Sort (IS)

The characterisation of the IS benchmark has been performed by instrumenting the source code and tracing one run for each problem size (i.e. class A and B). The core routine of the benchmark contains calls to `MPI_Alltoallv` and `MPI_Allreduce` operations. As our classification makes no distinction between

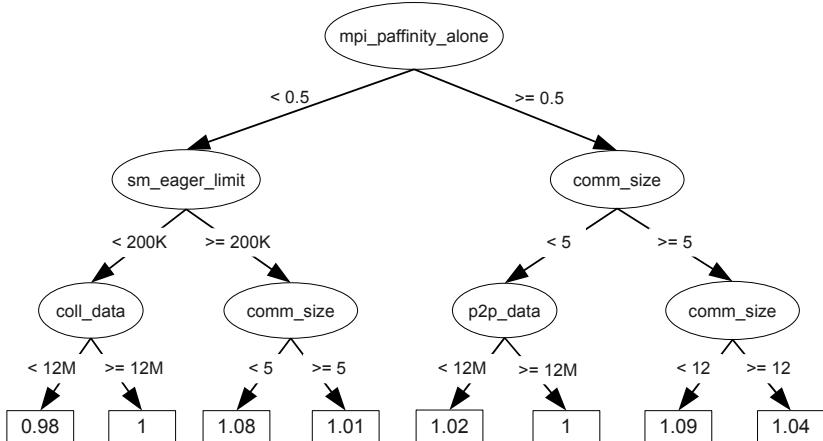


Fig. 4. Decision tree derived from the training data (`depth = 3`)

the two, the program has been characterised with `coll_ratio` = 1 (i.e. 100% of collective operations). The amount of data exchanged by the `MPI_alltoallv` operation depends on both the problem and the communicator size.

The result of the prediction based on `REPTree` for problem size A using 32 processes is depicted on the right of Fig. 3. For this benchmark, the predicted best configuration achieves around 84% of the maximum speedup. The `mpi_paffinity_alone` parameter value plays a key role in this benchmark, configurations with the flag value set to 0 always have better performance compared to the ones where the flag is activated (this behaviour can be observed in the graph comparing the speedup of two consecutive configurations). However, the predictor fails to correctly estimate the effect of the parameter on the speedup as the configurations where the affinity flag is equal to 1 are predicted to be faster. The other runtime parameter that clearly has effects on the speedup is the `sm_eager_limit` threshold. As the considered problem size requires to exchange around 32 KB of data among the processes, setting a value of 32 KB for the threshold (from configuration number 22) forces the library use the eager protocol instead of the more expensive rendezvous. This effect is correctly estimated by the predictor as the returned best configuration is {32768, 1, 2048, 1}.

In table 2, the accuracy of the ANN-based predictor is depicted. As for decision trees, the prediction of the affinity flag is wrong and the accuracy of the two predictors is comparable. The wrong prediction of the affinity bit can be explained considering the workload of the IS benchmark. As intra-chip communication is faster than inter-chip communication in our test system and because of the nature of the training set (based on point-to-point and collective operations), the trained predictor always turns on the affinity (see predicted configurations in Table 2). However, in computation-bound scenarios, affinity scheduling can cause performance degradation due to the limited amount of shared cache and available memory bandwidth per chip.

Table 2. Best measured and predicted configurations using ANN

Jacobi Method				
Feature Vector	Best conf.	speedup	Predicted conf.	speedup
0,2,1,0,8,1024, 8	32768,1, 512,1	3.83	65536,1, 32768,1	3.74
0,2,1,0,8,1024,16	262144,1, 512,0	1.09	2048,1, 512,0	1.02
0,2,1,0,8,2048, 8	32768,1,2097152,1	2.48	65536,1, 32768,0	2.36
0,2,1,0,8,2048,16	262144,0, 262144,0	1.09	262144,1, 512,0	1.03
0,2,1,0,8,4096, 8	32768,1, 512,0	1.76	32768,1, 32768,0	1.57
0,2,1,0,8,4096,16	262144,1, 262144,0	1.09	512,1,2097152,0	1.03
Integer Sort Benchmark				
Feature Vector	Best conf.	speedup	Predicted conf.	speedup
1, 262400,0,0,32	32768,0,512,1	1.18	32768,1,32768,0	1.14
1,1024000,0,0,32	32768,0,512,1	1.07	65536,0, 8192,0	1.03

5 Related Work

Most of the work about optimization of MPI applications focuses on *collective operations*. STAR-MPI (Self Tuned Adaptive Routines for MPI collective operations) [11], provides several implementations of the collective operation routines and it *dynamically* selects the best performing algorithm (using an empirical technique) for the application and the specific platform. Jelena et al. [12] also address the same problem by using machine learning techniques (decision trees). Compared to our work, these approaches are less general as the optimization is only limited to collective operations. Furthermore, the significant overhead introduced by the dynamic optimization environment makes these tools not suitable for short-running MPI programs.

The only work in literature that is comparable with our approach is OPTO [3]. OPTO systematically tests large numbers of combinations of Open MPI's runtime parameters for *common communication patterns* and *performance metrics* to determine the *best* set for a specific benchmark under a given platform. Differently from our approach, OPTO needs several runs of the application in order to find the optimal set of the runtime parameters and the search process is helped by configuration files provided by the user which specify the combinations and the parameter values that should be tested by the tool.

6 Conclusions and Future Work

This paper presents a novel approach to optimize MPI runtime environments for any application running on a target architecture. We build two predictors (based on machine learning techniques) capable of estimating the values, of a subset of Open MPI runtime parameters, which optimize the performance of unseen input programs for the underlying architecture. In contrast to existing optimization tools, our predictor needs only one run of the input program and no additional knowledge has to be provided by the user. No runtime overhead is

introduced as the model is built and trained *off-line*. Experiments demonstrate that the predicted optimal settings of runtime parameter for different programs achieve on average 90% of the maximum performance gain.

Future work will consider a better characterisation of MPI programs to improve the quality of the prediction (e.g. by introducing the computation-to-communication ratio to predict the affinity flag). Furthermore, we will apply our predictor to other hardware configurations (e.g. clusters of SMPs).

References

1. Open MPI: Modular Component Architecture, <http://www.open-mpi.org>
2. Bodin, F., Kisuki, T., Knijnenburg, P., O'Boyle, M., Rohou, E.: Iterative Compilation in a Non-Linear Optimisation Space. In: Proceedings of the Workshop on Profile Directed Feedback-Compilation (October 1998)
3. Chaarawi, M., Squyres, J.M., Gabriel, E., Feki, S.: A tool for optimizing runtime parameters of open mpi. In: Lastovetsky, A., Kechadi, T., Dongarra, J. (eds.) EuroPVM/MPI 2008. LNCS, vol. 5205, pp. 210–217. Springer, Heidelberg (2008)
4. Chai, L., Lai, P., Jin, H.W., Panda, D.K.: Designing an efficient kernel-level and user-level hybrid approach for mpi intra-node communication on multi-core systems. In: ICPP 2008: Proceedings of the 2008 37th International Conference on Parallel Processing, Washington, DC, USA. IEEE Computer Society Press, Los Alamitos (2008)
5. Meijerink, J.A., Vorst, H.A.v.d.: An iterative solution method for linear systems of which the coefficient matrix is a symmetric m -matrix. Mathematics of Computation 31(137), 148–162 (1977)
6. der Wijngaart, R.F.V.: Nas Parallel Benchmarks Version 2.4. Technical Report NAS-02-007, Computer Science Corporation NASA Advanced Supercomputing (NAS) Division (October 2002)
7. Ian, H., Witten, E.F.: Data Mining: Practical Machine Learning Tools and Techniques. Elsevier, Amsterdam (2005)
8. Bishop, C.M.: Neural Networks for Pattern Recognition. Oxford University Press, Oxford (1996)
9. Ipek, E., Supinski, B.R., Schulz, M., McKee, S.A.: An Approach to Performance Prediction for Parallel Applications. In: Euro-Par Parallel Processing (2005)
10. Matlab: Neural Network Toolbox,
<http://www.mathworks.com/products/neuralnet/>
11. Faraj, A., Yuan, X., Lowenthal, D.: Star-mpi: self tuned adaptive routines for mpi collective operations. In: ICS 2006: Proceedings of the 20th annual international conference on Supercomputing, pp. 199–208. ACM, New York (2006)
12. Fagg, G.E., Angskun, T., Bosilca, G., Dongarra, J.J.: Decision trees and mpi collective algorithm selection problem (2006)

CoMPI: Enhancing MPI Based Applications Performance and Scalability Using Run-Time Compression

Rosa Filgueira, David E. Singh, Alejandro Calderón, and Jesús Carretero

University Carlos III of Madrid

Department of Computer Science

{rosaf, desingh, acaldero, jcarrete}@arcos.inf.uc3m.es

Abstract. This paper presents an optimization of MPI communications, called *CoMPI*, based on run-time compression of MPI messages exchanged by applications. A broad number of compression algorithms have been fully implemented and tested for both MPI collective and point to point primitives. In addition, this paper presents a study of several compression algorithms that can be used for run-time compression, based on the datatype used by applications. This study has been validated by using several MPI benchmarks and real HPC applications. Show that, in most of the cases, using compression reduces the application communication time enhancing application performance and scalability. In this way, *CoMPI* obtains important improvements in the overall execution time for many of the considered scenarios.

1 Introduction

Cluster architectures have become the most common solution for implementing scientific applications. A cluster is a collection of computers (nodes), working together and interconnected by a network. The network used is commonly, but not always, a fast local area network.

At present time, many scientific applications require a large number of computing resources and operate on a large data volume that has to be transferred among the processes. Because of that, communications are a major limiting factor of the application performance in cluster architectures. The programming model used in most clusters is the MPI (Message-Passing Interface) standard. Therefore, this paper presents an extended MPI run-time, called *CoMPI*, which uses data compression to reduce the volume of communications, thus reducing execution time and enhancing scalability. The technique developed can fit on any particular application, due to its implementation is transparent for the user, and integrates different compression algorithms that can be used depending the datatype and characteristics of the application.

We have implemented the proposed technique by using *MPICH* [1] implementation, developed jointly by Argonne National Laboratory and Mississippi State University.

We also have evaluated our implementation with the original *MPICH*, and the results show that our approach obtains, for many of the considered scenarios, important reductions in the execution time by reducing the size of the messages. It reduce the

total communication time and the network contention, thus enhancing, not only performance, but also scalability.

The rest of the paper is structured as follows: Section 2 contains related work. Section 3 the architecture of *CoMPI*, showing the MPI modules where the compression techniques are implemented. Section 4 contains a study of the performance of each compression technique for different synthetic traces. Section 5 is dedicated to performance evaluation of the complete implementation for different benchmarks and applications. Finally, section 6 shows our conclusions and future work.

2 Related Work

In this section two main aspects are reviewed: existing techniques to add compression into MPI, and the most interesting compression algorithm to be used for a high performance environment.

2.1 Adding Compression to MPI

The use of compression within MPI is not new, although it has been particularized for very few special cases. Major examples of compression added into MPI tools are cMPI, PACX-MPI, COMPASSION and MiMPI.

PACX-MPI (PArallel Computer eXtension to MPI) [2,3] is an on-going project of the HLRS, Stuttgart. It enables an MPI application to run on a meta-computer consisting of several, possibly heterogeneous machines, each of which may itself be massively parallel. Compression is used for TCP message exchange among different systems in order to increase the bandwidth, but a fixed compression algorithm is used and compression is not used for messages within the same system.

cMPI [4,5] has some similar goals as PACX-MPI: to enhance the performance of inter-cluster communication with a software-based data compression layer. Compression is added to all communication, so it has not flexibility in order to configure when and how to use compression.

COMPASSION [6] is a parallel I/O runtime system including chunking and compression for irregular applications. The LZO algorithm is used for fast compression and decompression, but again it is only used for (and focused on) the I/O part of irregular application.

MiMPI [7,8] was a prototype of a multithread implementation of MPI with thread-safe semantics that adds run-time compression of messages sent among nodes. Although the compression algorithm can be changed (providing more flexibility), the use of compression is global for all process of a MPI application. That is to say: all the MPI processes have to use the same compression technique at once. The compression is used for messages larger than a given size, and can not be tailored per message (based on datatype, source rank, destination rank, etc.).

As far as the authors know, there is no MPI implementation with compression support that allows to dynamically selecting the estimated as the most appropriated compression algorithm to be used per message exchange inside the MPI primitives.

Table 1. Compressors considered for inclusion within a MPI implementation

Name	Characteristics	Recommended data type
LZO	LZO is a block compression algorithm with very low compression and decompression time	Binary (image, sound, etc.)
RLE	Based on sequences of the same byte	Text
Huffman	Assign short codes to frequent sequence of bytes. Works at byte level	Text and binary
Rice	Rice coding could be used as the entropy encoding method	Binary (image, sound, etc.)
FPC	Based on the prediction of the next value	Doubles

2.2 Compression Algorithms

Our target is to compress the exchanged messages of parallel scientific applications in a transparent way. This implies some important requirements:

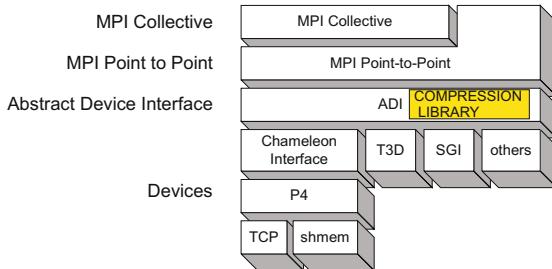
- The compression algorithm must be a lossless algorithm in order to preserve the information.
- The compressor must produce the smallest overhead possible in terms of execution time and memory requirements. Because of that, is more important for us to get speed than a compression ratio.

In this work we have preselected a group of compressors in order to evaluate their suitability for their inclusion in our MPI platform. The compressors selected have been used in some high performance environments (like PACX-MPI, MiMPI, etc.) or in real-time systems where the time needed for compression/decompression is critical. These compressors are: RLE [9], Huffman [10], Rice [11], FPC [12] and LZO [13]. Table 1 shows the main characteristics of each one of them.

There are many compressors that have been evaluated with different dataset. Nowadays, the best well known benchmark to evaluate lossless compression methods is the canterbury corpus. In the associated web site visitors can find the results for compression ratio [14], compression time, and decompression time of several different compressors. Unfortunately, the canterbury corpus is a general purpose benchmark, and many of our compressors are not present in its results. For these reasons this paper also introduces a novelty benchmarking of the selected compressors for messages of MPI applications.

3 Compression of MPI Messages

There are several factors that limit the performance of parallel MPI applications executed on clusters. One of the major bottlenecks is the communication overhead. This overhead is especially important for communication networks with low bandwidths and high latency. Our main objective is to reduce the impact of the communications using different compression algorithms.

**Fig. 1.** Layers of *MPICH*

The basic communication mechanism of MPI is the transfer of data between a pair of processes, one sender and one receiver. This kind of communication is point to point communication. MPI also defines in collective communication routines, a pattern of communication is established amongst a group of processes.

MPI is only a standard upon which many implementations exist. In this work we have used *MPICH*[1]. The architecture of *MPICH* consists of 3 layers: Application Programmer Interface (API), Abstract Device Interface (ADI) and Channel Interface (CI). API is the interface between the programmer and ADI. The API uses an ADI to send and receive information. The ADI controls the data flow between API and hardware. It specifies whether the message is sent or received, handles the pending message queues, and contains the message passing protocols. In other words, the point-to-point communications are provided by the ADI. Our objective is to provide of compression facilities to all MPI communications (point-to-point and collective). Collective communication routines are implemented using point-to-point routines. Therefore, applying compression on point-to-point routines, not only compress these communications but also the collective communications. Therefore, we have modified the ADI layer in order to include facilities for compressing and decompressing the communicated data, as shows Figure 1. This is done by using a external compression library.

We have used different compression algorithms depending on the type of data used in each communication. Also, it allows using (or not) compression per message, depending on the length of data is larger than a pre-set size. We include a header in the exchanged messages in order to inform the receiver process, whether it has to decompress or not the message, and which is the algorithm that must be use. This header contains the following information:

- Compression used or not: a byte that indicates if the message is compressed or not.
- Compression algorithm: a byte that indicates which compression algorithm has been selected.
- Original length: an integer that represents the original length of data. This is information is necessary to decompress the data.

All compression algorithms have been included in a single library called *Compression-Library*. To add more compression algorithms, we only have to replace this library with a new version. Therefore, *CoMPI* can be updated easily to include new compression algorithms.

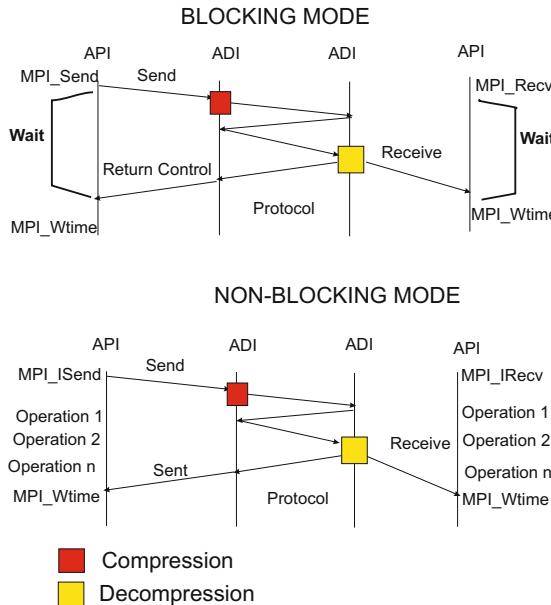


Fig. 2. Compression and decompression for Blocking and Non-Blocking modes

The communication time for messages is determined by communication network bandwidth. To achieve a benefit from transferring compressed data instead of uncompressed data, the additional computational time of the compression algorithm has to be lower than the time saved during the communication. We have performed several studies, and we note that if a message is smaller than 2 KB more time is spent performing the compression/decompression than send the uncompressed data. For this reason, when the message is smaller than 2 KB, we do not apply compression.

As we can see in Figure 2, before ADI sends data, a compress operation must be performed. In the same way, when ADI receives data, must decompress them. ADI can send messages in blocking and non-blocking form. Also, it can send and receive messages with datatypes contiguous or non-contiguous. So, when the message to send is non-contiguous, the ADI packs the data in a contiguous buffer, and then compression is performed. On the other hand, when the message to receive is non-contiguous type, the decompression is performed and, after that, data is unpacked. The stages to compress and decompress a message are detailed as follows:

- Compression stages:
 1. Message size evaluation: If the size of message is smaller than 2 KB, data is sent uncompressed.
 2. Compression algorithm selection: Select the compression algorithm depending on the type of data.
 3. Data compression: Compress data, with the most appropriated algorithm and check the size of the compressed data. If the size of compressed data is larger than original data, the original message is sent.

4. Header inclusion: Header is added to the message in order to notify to the receiver whether the message has to be decompressed and which decompression algorithm has to be used.
- Decompression stages:
 1. Header checking: The header is checked in order to know whether the message has to decompress or not and which is the algorithm that must be used.
 2. Data decompression: The data is decompressed in case it is compressed.

4 Compression Evaluation

This section contains the study that we have developed in order to select the most appropriate compression algorithm for each type of data our study takes into account two factors: buffer size to compress and redundancy level in the buffer.

When data compression is used in data transmission, the goal is to increase the transmission speed which depends on the number of bits sent, the time required for the encoder to generate the coded message, and the time required for the decoder to recover the original ensemble. Therefore, it is necessary to study the algorithm complexity and the amount of compression for each compressor with different types of data, buffer sizes and redundancy level. A redundancy level of a given compression buffer (measured in percentage) is defined as the percentage of entries with zero values. These entries are randomly distributed among the buffer memory map¹.

We have developed synthetic datasets with three datatype (integer, floating-point and double precession). Each dataset contains buffers with different:

- Buffer size: The buffer sizes considered are 100 KB, 500 KB, 900 KB and 1500 KB for integer and floating-point datasets. For double precision datasets, the buffer sizes are 200 KB, 1000 KB, 1800 KB and 3000 KB .
- Redundancy levels: 0%, 25%, 50%, 75% and 100% .

Our target is to evaluate the speedup, complexity and compression ratio of the algorithms described in Table 1 when the synthetic datasets are used. The speedup is calculated as shown in equation 1:

$$\frac{time_Sent_Original}{(time_Sent_Compressed + time_compression + time_decompression)} \quad (1)$$

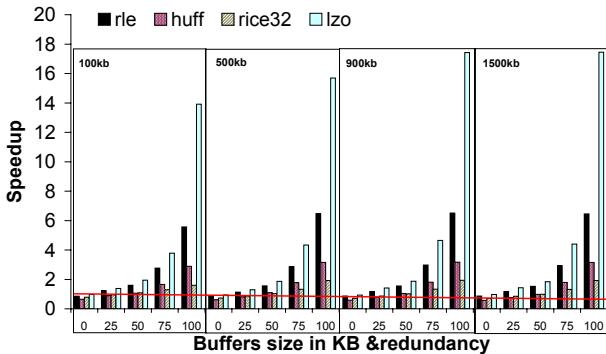
We measured message communication latency and bandwidth with the MPPTEST tool, available in the *MPICH* distribution. MPPTEST performs point to point communications that are basically the classic ping-pong of messages with different size, repeated several times. The results have been obtained in the cluster described in the *Evaluation using applications and benchmarks* section.

Figure 3 illustrates the results of this evaluation. We can see that the LZO compression algorithm achieves the highest speedup. Looking at the compression ratio shown in Table 2, we can see that the Rice compression algorithm achieves the highest ratio when the redundancy level is repeated between 0% and 75%. In contrast, when the redundancy

¹ Note that the redundancy level and the entropy of the message contents are related.

Table 2. Compression and decompression time for integer data

Size (KB)	Redundancy %	Compression ratio				Time compr. + decompr. (ms)			
		RLE	HUFF	RICE	LZO	RLE	HUFF	RICE	LZO
100	0%	0	38.8	53.1	19.3	1.3	8.2	7.2	1.8
100	25%	33.4	55.5	68.3	43.6	1.3	6.2	6.5	1.4
100	50%	53.7	65.5	77.6	61.9	1.3	5	6.2	1
100	75%	78.8	77.5	88.1	81.3	1.4	3.4	5.7	0.8
100	100%	99.9	87.5	96.9	99.6	1.3	1.7	5.2	0.4
500	0%	0	31.7	46.6	16.8	6.8	43.2	36.8	10.5
500	25%	27.7	48.7	61	40.7	6.8	33	34.1	7.8
500	50%	51.3	63.1	73.4	59.6	6.9	24.8	31.7	5.7
500	75%	80.5	78.1	88.1	83.4	6.8	15.4	28.5	2.9
500	100%	99.9	87.5	96.8	99.1	6.8	8.5	22.1	2.2
900	0%	0	28.3	43.8	16.5	11.6	81.3	66.9	19.2
900	25%	29.7	47.1	60.4	44.5	12.1	59.8	61.4	12
900	50%	51	60.7	72.1	59.6	12.4	46	57.6	10.4
900	75%	82.1	78.6	88.4	84.8	12.5	26.9	51.2	5
900	100%	99.9	87.5	96.8	99.6	12.4	15.3	39.2	4.1
1500	0%	0	26.4	41.6	19.1	19.2	135.4	117.8	28.9
1500	25%	29.3	44.3	58.7	44.7	19.8	102	102.6	20.1
1500	50%	49.8	58.1	70.3	59.1	20.5	80.4	95.8	17.8
1500	75%	81.5	78.1	87.8	84.1	21	46.6	85.4	9.4
1500	100%	99.9	87.5	96.9	99.6	20	26	65.3	6.9

**Fig. 3.** Speedup for integer data with different buffer sizes and redundancy levels

level is 100%, the RLE and LZO compression algorithms achieve the highest compression ratio. However, Table 2 also shows that Rice algorithm is slower than the LZO and RLE algorithms for all the considered scenarios. In terms of performance, we found that the LZO compression algorithm is the most appropriate to compress integer data.

We have performed the same study for floating-point and double-precision datasets. Figure 4 shows the results in terms of the speedup for floating-point and Figure 5 for double-precision data.

Figure 4 illustrates, for the redundancy level of 0%, compression is not useful in order to improve the performance. For redundancy level equal or higher than 25%, LZO and RLE are the algorithms that obtain the best results.

We have evaluated an additional algorithm (FPC), for double-precision data, whose main characteristics are detailed in Section 2.2. FPC is based on data prediction. Thus

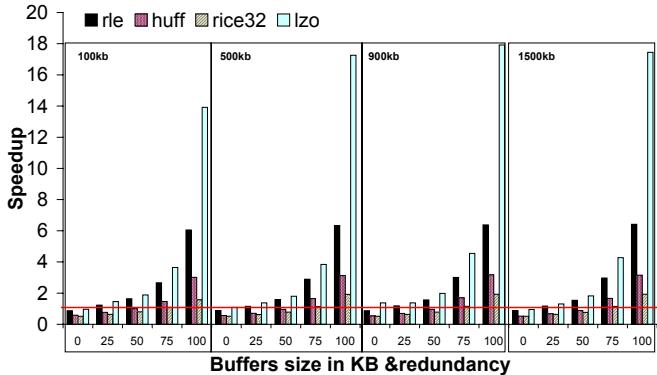


Fig. 4. Speedup for floating-point data with different buffer sizes and redundancy levels

two different traces have been used: without pattern and with a pattern². For traces without pattern, LZO achieves again the better speedup. However, for traces with a behaviour pattern and redundancy level between 0% and 50%, the best results are obtained using FPC. In the others cases, the best option is the LZO algorithm.

In this study we can conclude that the LZO algorithm achieves the highest speedup almost all the considered scenarios.

5 Evaluation Using Applications and Benchmarks

We have evaluated the performance of our *MPICH* implementation using the following applications and benchmarks:

- *BISP3D*: Is a 3-dimensional simulator of BJT and HBT bipolar devices. Communicated datatype are floating-point.
- *PSRG*: Is a parallel segmentation tool that process grey-scale images [15], exchanging integers datatype messages.
- *STEM-II*: Is an Eulerian numerical model to simulate the behavior of pollutant factors in the air [16]. Communicated datatype are floating-point.
- The *NAS Parallel Benchmarks*: A set of benchmarks targeting performance evaluation of massively parallel computers [17]. In this work, we have used two of them:
 - *IS*: Sort small integers using the bucket sort. Communicated datatype are integers.
 - *LU*: Solves a synthetic system of nonlinear PDEs using symmetric successive over-relaxation (SSOR) solver kernels. Communicated datatype are double-precision.

This section compares the speedup of all applications when use the original *MPICH* and *CoMPI* with different numbers of processes and different compression algorithms.

² We understand as pattern a given data sequence that can be easily predicted. An example of this sequence is [50001.0 50003.0 50005.0 50007.0 ...].

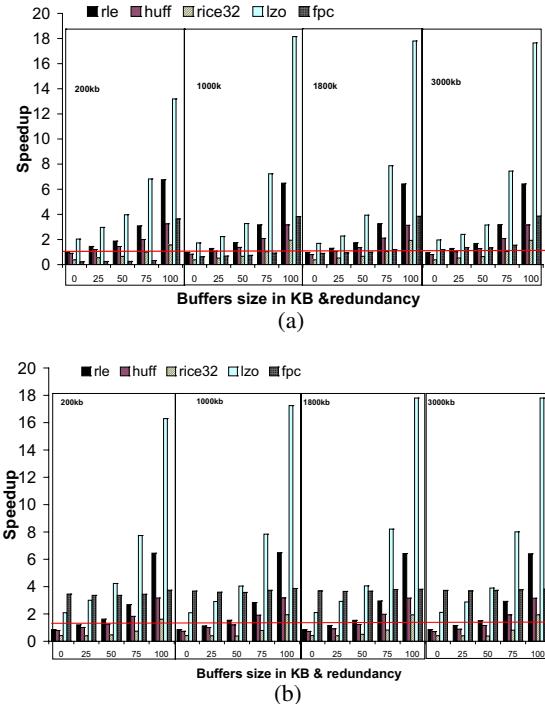


Fig. 5. Speedup for double data: (a) without pattern and (b) with pattern.

We have used the MPICHGM-1.2.7.15NOGM-COMP distribution and developed our technique by modifying the ADIO layer.

A cluster with 64 nodes has been used for the tests. Each node is Dual-Core AMD with 512 MB of RAM. The interconnection network is FastEthernet.

Figure 6 shows the speedup achieved using *CoMPI*. The speedup is defined as the ratio between the original *MPICH* and the *CoMPI* execution times. A value higher than 1 indicates that compression reduces the application execution time. In contrast, a value smaller than 1 implies an increment of the execution time when compression is used.

The figures show that many applications improve the performance when compression is applied, especially when the problem size grows (scalability). *BISP3D* improves between 1.2 and 1.4 when we use the LZO compression algorithm. This is due to the fact that LZO compressed more efficiently than the others compression algorithms, as we showed in Section 4.1. *PSRG* improvement is up to 2 when LZO is used because integers compressed very efficiently. *STEM-II* performance improves to 1.4 using LZO. *IS* benchmark also uses integer data, and using LZO algorithm, the performance is improved between 1.2 and 1.4. In contrast, for *LU* benchmark, there is no performance improvement but notice that performance degradation is small. However, using 32 processes and FPC algorithm, a speedup equal than 1 is reached. This is due to fact that the doubles compressed worse than the others datatype and the redundancy level of these benchmarks are close to 0%.

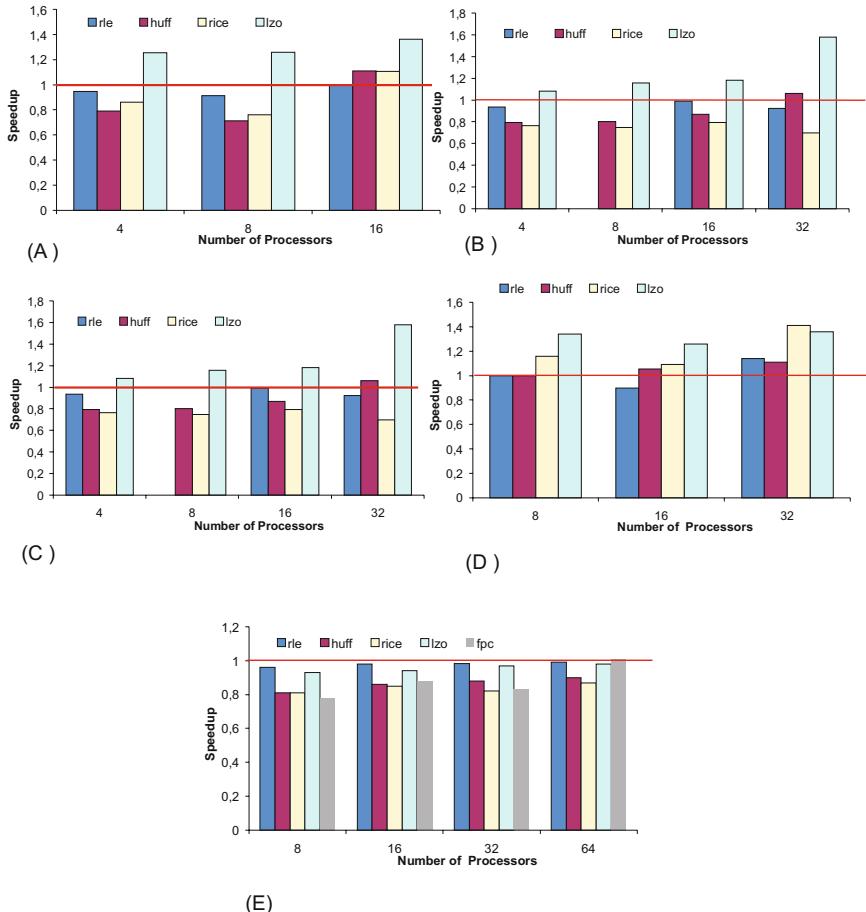


Fig. 6. Execution time improvement of applications: (a) BISP3D (b) PSRG (c) STEM-II (d) IS (e) LU

We emphasize that, the greater number of processes, the bigger application speedups. This behaviour is due to the increasing number of communications, and therefore, the improvement of the communication performance produces a bigger impact on the overall application performance. Scalability is also enhanced with compression. We can see that the performance improvement depends of type of data, redundancy level, and specific characteristics of the application. Usually, the LZO efficiency is better for integers and floats than doubles. For this reason, *CoMPI* will use LZO algorithm to compress integers and floating-point.

6 Conclusions

In this paper we present a new compression library fully integrated into MPI that transparently allows compressing the exchanged messages. Our proposal addresses both

point to point and collective communications. We have integrated this library in MPICH and we call it *CoMPI*. Currently, *CoMPI* includes five different compression techniques that can be used transparently to users (by means of a MPI hints). The selection of the specific compressor is codified in the message header which allows using different compressors in each process and communication stage.

In this work we have evaluated *CoMPI* in different scenarios: Firstly, using synthetic traces, we analyzed the performance (both in terms of compression ratio and execution time) of each compression technique. Different factors (type of data, buffer sizes, redundancy levels and data patterns) were considered, concluding that the LZO is the most efficient approach for most of the cases (but not for all cases). Secondly, we evaluated CoMPI using different benchmarks and real applications. The results demonstrated that, in most of the cases, it is feasible the use of compression in all communication primitives. In addition, the performance gain is bigger when more processes are employed. In contrast, when compression is not appropriated (LU and MG benchmarks) a little bit degradation is observed on the performance evaluation.

As future work, we want to develop an adaptive compression framework that allows selecting in runtime the most appropriate compression technique taking into account the specific communication characteristics (type of data, size of the message, etc.), the platform specs (network latencies, computing power, etc.) and the compression technique performance.

Acknowledgements

This work has been partially funded by project TIN2007-63092 of Spain Ministry of Education.

References

1. Gropp, W., Lusk, E.: Sowing MPICH: A case study in the dissemination of a portable environment for parallel scientific computing. *The International Journal of Supercomputer Applications and High Performance Computing* 11(2), 103–114 (1997)
2. Balkanski, D., Trams, M., Rehm, W.: Heterogeneous computing with mpich/madeleine and pacx mpi: a critical comparison (2003)
3. Keller, R., Liebing, M.: Using pacx-mpi in metacomputing applications. In: 18th Symposium Simulationstechnique, Erlangen, September 12-15 (2005)
4. Ratanaworabhan, P., Ke, J., Burtscher, M.: Fast lossless compression of scientific floating-point data. In: DCC 2006: Proceedings of the Data Compression Conference, Washington, DC, USA, pp. 133–142. IEEE Computer Society Press, Los Alamitos (2006)
5. Ke, J., Burtscher, M., Speight, E.: Runtime compression of mpi messanes to improve the performance and scalability of parallel applications. In: SC 2004: Proceedings of the 2004 ACM/IEEE conference on Supercomputing, Washington, DC, USA, p. 59. IEEE Computer Society Press, Los Alamitos (2004)
6. Carretero, J., No, J., Park, S.-S., Choudhary, A., Chen, P.: Compassion: a parallel I/O runtime system including chunking and compression for irregular applications. In: Proceedings of the International Conference on High-Performance Computing and Networking, April 1998, pp. 668–677 (1998)

7. García Carballeira, F., Galderón, A., Carretero, J.: MiMPI: A multithread-safe implementation of MPI. In: Margalef, T., Dongarra, J., Luque, E. (eds.) PVM/MPI 1999, vol. 1697, pp. 207–214. Springer, Heidelberg (1999)
8. Thakur, R.: Issues in developing a thread-safe MPI implementation. In: Mohr, B., Träff, J.L., Worringen, J., Dongarra, J. (eds.) PVM/MPI 2006. LNCS, vol. 4192, pp. 12–21. Springer, Heidelberg (2006)
9. Zigon, R.: Run length encoding. *j-DDJ* 14(2) (February 1989)
10. Knuth, D.E.: Dynamic huffman coding. *J. Algorithms* 6(2), 163–180 (1985)
11. Coco, S., D'Arrigo, V., Giunta, D.: A rice-based lossless data compression system for space. In: Proceedings of the 2000 IEEE Nordic Signal Processing Symposium, pp. 133–142 (2000)
12. Burtscher, M., Ratanaworabhan, P.: Fpc: A high-speed compressor for double-precision floating-point data. *IEEE Transactions on Computers* 58(1), 18–31 (2009)
13. Oberhummer, M.F.X.J.: Lzo real-time data compression library (2005)
14. Bell, T., Powell, M., Horlor, J., Arnold, R.: The canterbury corpus (2009)
15. Pichel, J.C., Singh, D.E., Rivera, F.F.: Image segmentation based on merging of sub-optimal segmentations. *Pattern Recogn. Lett.* 27(10), 1105–1116 (2006)
16. Mourino, J., Martin, M., Doallo, R., Singh, D., Rivera, F., Bruguera, J.: The stem-ii air quality model on a distributed memory system (2004)
17. Carter, R., Ciotti, B., Fineberg, S., Nitzberg, B.: NHT-1 I/O benchmarks. Technical Report RND-92-016, NASA Systems Division, NASA Ames (1992)

A Memory-Efficient Data Redistribution Algorithm^{*}

Stephen F. Siegel¹ and Andrew R. Siegel²

¹ Verified Software Laboratory, Department of Computer and Information Sciences,
University of Delaware, Newark, DE 19716, USA
siegel@cis.udel.edu

² Mathematics and Computer Science Division, Argonne National Laboratory,
9700 South Cass Avenue, Argonne, IL 60439, USA
siegeala@mcs.anl.gov

Abstract. Many memory-bound distributed applications require frequent redistribution of data. Pinar and Hendrickson investigated two families of memory-limited redistribution algorithms. The first family has many advantages, but fails on certain inputs, and, if not implemented carefully, may lead to an explosion in the number of local data copies. The second family eliminates the possibility of failure at the expense of considerable additional overhead. We carefully analyze these algorithms and develop a modified method that potentially combines advantages of each. The resulting algorithm has been implemented in MADRE and experiments reveal its performance to be superior to that of other MADRE algorithms in most cases.

Keywords: MADRE, redistribution, memory-limited, distributed, MPI.

1 Introduction

Many distributed-memory, message-passing, parallel programs periodically redistribute data among the program's processes. Solutions to such problems have been studied extensively in a variety of contexts. For example, the well known $M \times N$ parallel data migration problem (e.g. [2, 1]) involves efficiently moving data between typically disjoint processor sets with different decomposition schemes. A related problem that has received considerable attention is the so-called block-cyclic array redistribution (e.g. [7, 3]), which seeks efficient algorithms to map data from a *cyclic(x)* to a *cyclic(Kx)* decomposition.

The class of algorithms discussed in this paper address a related problem but where the driving requirement is the transparent and efficient use of memory at the application level. Furthermore, unlike much of the work on block-cyclic algorithms, aspects of the physical network (processor topology, routing algorithms, etc.) are intentionally abstracted away. A large class of scientific applications

* This research was supported by the National Science Foundation under Grants CCF-0733035 and CCF-0540948.

are memory-bound, and cannot guarantee the availability of sufficient memory for a more straightforward approach to data migration (e.g. of invoking MPI_ALLTOALL). For these applications, a limited-memory redistribution algorithm is required.

Pinar and Hendrickson [4] introduced a formal “phase”-based description of the limited-memory redistribution problem, showed the problem of finding a minimal-phase solution to be NP-hard, and analyzed a class of “basic” algorithms approximating optimal solutions. They also pointed out that the basic algorithms would not complete on some inputs. For this reason, and to further reduce the number of phases, a second class of “parking” algorithms was introduced. These complete on any valid input, but add significant overhead and inter-process synchronization, and in many of the experiments of [4] performed worse than the basic ones.

The Memory-Aware Data Redistribution Engine (MADRE) has been used to study these and other solutions to the limited-memory redistribution problem [5, 6]. Experiments with MADRE revealed that in some cases, the time spent on local data copies could dominate instances of the Pinar-Hendrickson algorithms, an issue that was abstracted away in [4].

In this paper, we show that the basic algorithms can be modified to overcome both issues described above: the modified algorithm can complete on any input and the number of local data copies can be significantly reduced. We give a precise description of the original algorithm in Section 2, and prove that the algorithm is always deadlock-free, though it can “livelock” for certain inputs. In Section 3, we find a condition on the input that suffices for termination. Using this, we show how the basic algorithm can be modified to terminate successfully on any input. The problem of local data copies is dealt with in Section 4. The resulting algorithm has been implemented in MADRE, and experiments comparing its performance are given in Section 5. The results show the new algorithm to be superior to all other MADRE algorithms in most cases.

2 The Basic Pinar-Hendrickson Redistribution Algorithm

We are given a distributed memory parallel program of n processes. Each process maintains in its local memory a contiguous, fixed-sized array of data blocks. At any time, some number of these blocks are *free*, i.e., do not contain useful data. The free blocks may be used as temporary space in the redistribution.

The input to a redistribution algorithm consists of the following:

1. free_p ($0 \leq p < n$): the initial number of free spaces on each process, and
2. $\text{out}_p[i]$ ($0 \leq p, i < n$): the number of blocks process p is to send to process i .

We assume all of these values are non-negative and $\text{out}_p[p] = 0$ for all p . (In practice, the input must specify the destination rank and index for each non-free block, but in this presentation we elide such detail.) The goal is to transfer the blocks to their new locations in a sequence of phases without ever exceeding the free space available on any process.

symbol	meaning
free	number of free spaces on this proc (given)
out[i]	number of blocks remaining to send to proc i (given)
in[i]	number of blocks remaining to receive from proc i (initially 0)
send[i]	number of blocks to send to i in current phase (initially 0)
recv[i]	number of blocks to receive from i in current phase (initially 0)

```

1 procedure main is
2   Alltoall(out, 1, INT, in, 1, INT);
3   nPhase  $\leftarrow$  0;
4   while  $\sum_i$ (out[i] + in[i])  $>$  0 do
5     | computeMoves();
6     | executePhase();
7     | nPhase  $\leftarrow$  nPhase + 1;
8     | move blocks to final positions;
9   procedure computeMoves is
10    r  $\leftarrow$  free;
11    for i  $\leftarrow$  0 to n - 1 do
12      | recv[i]  $\leftarrow$  min{r, in[i]};
13      | r  $\leftarrow$  r - recv[i];
14    foreach {i | in[i] > 0} do
15      | post send of recv[i] to proc i;
16    foreach {i | out[i] > 0} do
17      | post recv into send[i] for proc i;
18    wait for all requests to complete;
19   procedure executePhase is
20     | sort blocks to create contiguous
21     | buffers;
22     | foreach {i | recv[i] > 0} do
23       | post recv for recv[i] blocks
24       | from proc i;
25       | in[i]  $\leftarrow$  in[i] - recv[i];
26     | foreach {i | send[i] > 0} do
27       | post send of send[i] blocks to
28       | proc i;
29       | out[i]  $\leftarrow$  out[i] - send[i];
30     free  $\leftarrow$  free +  $\sum_i$ (send[i] - recv[i]);
31   wait for all requests to complete;

```

Fig. 1. Basic Pinar-Hendrickson Algorithm using “first-fit” heuristic

Let $\text{in}_p[i] = \text{out}_i[p]$ and

$$\text{free}'_p = \text{free}_p - \sum_{i=0}^{n-1} \text{in}_p[i] + \sum_{i=0}^{n-1} \text{out}_p[i]. \quad (1)$$

Then free'_p will be the amount of free space on process p after redistribution. The input is *feasible* if $\text{free}'_p \geq 0$ for all p . Ideally, a redistribution algorithm should work correctly for any feasible input. We will see, however, that the Basic Algorithm described below fails for certain feasible inputs.

The *Basic Algorithm* appears in Figure 1. The input provides the initial values for the variables free and $\text{out}[i]$ on each process p . Process p proceeds in a sequence of phases until it has no more blocks to send or receive. In each phase, p receives into its free space while sending from some of its non-free spaces. In the first part of a phase, p allocates its current free spaces to the processes that have data to send to it. Different heuristics can be used to determine this allocation; for simplicity we use the *first-fit* heuristic of [4], though the results we describe here are valid for any heuristic. After deciding on the allocation, p sends a message consisting of a single integer to all of its source processes informing them of how many blocks they should send to p in the current phase. At the same

time, p receives similar messages from its targets. Once this communication has completed, the phase is *executed* by sending and receiving the agreed quantities of data to and from each process, and updating local data structures.

We first establish the following:

Theorem 1. *The Basic Algorithm is deadlock-free on any input.*

Proof. Let E be a maximal execution. This means that either (1) E is infinite, or (2) E is finite and every process has either terminated or become permanently blocked waiting for some request to complete.

Say $0 \leq p < n$. If process p becomes permanently blocked in E , let M_p be the final value of `nPhase`. Otherwise let $M_p = \infty$. For $0 \leq j \leq M_p$ and $0 \leq i < n$, let $\text{out}_{p,j}[i]$ denote the value of `out`[i] on process p when the expression on line 4 is evaluated and `nPhase` = j , or 0 if p terminates before `nPhase` = j . Define $\text{in}_{p,j}[i]$, $\text{send}_{p,j}[i]$, and $\text{recv}_{p,j}[i]$ similarly. We show for all $j \geq 0$, the following all hold:

1. $M_p \geq j$ ($0 \leq p < n$),
2. $\text{in}_{p,j}[i] = \text{out}_{i,j}[p]$ ($0 \leq i, p < n$),
3. $\text{send}_{p,j}[i] = \text{recv}_{i,j}[p]$ ($0 \leq i, p < n$), and
4. if $j > 0$, any send request generated in phase $j - 1$ is matched with a receive request generated in phase $j - 1$.

This implies, in particular, $M_p = \infty$ for all p , i.e., E does not deadlock.

The proof is by induction on j . The case $j = 0$ is clear: the all-to-all operation guarantees the `in` and `out` variables correspond in the desired way, and the `send` and `recv` arrays are uniformly 0. Suppose $j > 0$ and the hypothesis holds for values less than j ; we will show it holds for j .

Suppose process p enters phase $j - 1$ and posts a send to process i in `computeMoves`. Then $\text{out}_{p,j-1}[i] = \text{in}_{i,j-1}[p] > 0$, by the induction hypothesis. So process i must enter phase $j - 1$ and post a receive to p in `computeMoves`. Similarly, if p enters and posts a receive for process i then i must enter and post a send to p .

Since all requests from previous phases have been matched, each send in phase $j - 1$ must be paired with the corresponding receive. All of these paired requests must eventually complete, so all processes which have not terminated will return from `computeMoves`. Moreover, since the message sent by process p to process i is exactly $\text{recv}_{p,j}[i]$, and this message is received into $\text{send}_{i,j}[p]$, we have $\text{recv}_{p,j}[i] = \text{send}_{i,j}[p]$.

In `executePhase`, the same argument shows that the sends and receives match up in the expected way. Moreover, the assignments on lines 23 and 26 guarantee

$$\text{in}_{p,j}[i] = \text{in}_{p,j-1}[i] - \text{recv}_{p,j}[i] = \text{out}_{i,j-1}[p] - \text{send}_{i,j}[p] = \text{out}_{i,j}[p],$$

which completes the inductive step. \square

3 Termination and Modified Basic Algorithm

Though the Basic Algorithm cannot deadlock, it is possible for it to “livelock”: some processes may loop forever without making progress. This may happen

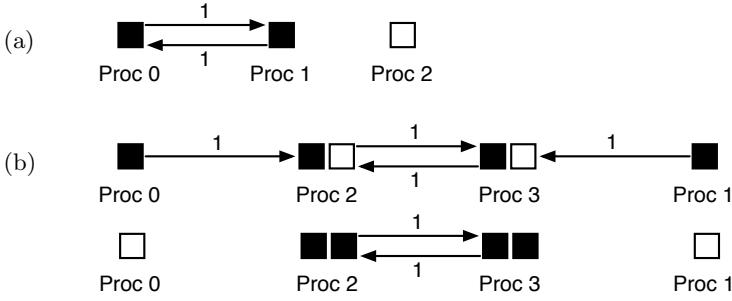


Fig. 2. Scenarios in which Basic Algorithm fails

even if the input is feasible. An obvious case is where no process has any free space. Another example, considered in [4], is where two processes with no free space attempt to exchange data while there is free space on a third (Figure 2(a)). In these cases the algorithm will always fail, regardless of the heuristic used to assign free spaces to incoming blocks.

What is less obvious is that there are situations where the algorithm can succeed if certain choices are made but will fail for other choices. Consider the example of Figure 2(b). If, in the first phase, process 2 chooses to allocate its one free space to process 0, and process 3 chooses to allocate its one free space to process 1, then the algorithm will not progress after the phase completes. (These are in fact the choices made by the first-fit heuristic.) If, on the other hand, process 2 first selects 3, and 3 selects 2, the algorithm completes normally.

To deal with these problems, and to reduce the total number of phases, Pinar and Hendrickson introduce the concept of *parking*. The idea is to move data blocks to temporary locations on processes that have extra free space, and later move these blocks to their final destinations. They describe a family of such algorithms and show that these will solve any feasible redistribution problem as long as there is at least one process with at least one free space. But as mentioned above, the parking algorithms also have many disadvantages.

The following establishes a sufficient condition for the successful completion of the Basic Algorithm:

Theorem 2. Suppose the input to the Basic Algorithm satisfies $\text{free}'_p \geq 1$ for all p . Then all processes will terminate normally.

Proof. For any process p , whenever control enters the **while** loop,

$$\text{free} - \sum_{i=0}^{n-1} \text{in}[i] + \sum_{i=0}^{n-1} \text{out}[i] = \text{free}'_p \geq 1. \quad (2)$$

This is true when control first enters the loop by the definition of free'_p . It remains true on subsequent iterations because the value of the expression on the left-hand side of (2) is preserved by *computeMoves* and *executePhase*.

By Theorem 1, the algorithm does not deadlock. Suppose it livelocks. Then eventually a point is reached after which the values of *in*, *out*, and *free* never

change. For $0 \leq p < n$, let in_p denote the final value of $\sum_i in[i]$ on process p . Define out_p similarly. Let $free_p$ be the final value of free on process p . Let $S = \{p \mid in_p > 0\}$. After the stable point has been reached, any process with incoming data must have no free space, i.e., $p \in S \Rightarrow free_p = 0$. By (2),

$$-in_p + out_p \geq 1 \quad \text{for all } p \in S. \quad (3)$$

Summing (3) over all $p \in S$ yields

$$-\sum_{p \in S} in_p + \sum_{p \in S} out_p \geq |S|. \quad (4)$$

On the other hand, the total amount of incoming data must equal the total amount of outgoing data:

$$\sum_{p=0}^{n-1} in_p = \sum_{p=0}^{n-1} out_p$$

Hence

$$\sum_{p \in S} in_p - \sum_{p \in S} out_p = \sum_{p=0}^{n-1} in_p - \sum_{p=0}^{n-1} out_p + \sum_{p \notin S} out_p = \sum_{p \notin S} out_p \geq 0 \quad (5)$$

Adding (4) and (5) yields $0 \geq |S|$, i.e., $S = \emptyset$. This means $in_p = 0$ for all p , and hence $out_p = 0$ for all p , so all processes terminate normally, a contradiction. \square

Theorem 2 suggests a way to modify the Basic Algorithm so that it will complete normally on any feasible input: simply allocate one extra block on every process. The existence of this block is not revealed to the user, so that a feasible map from the user's point of view will always result in at least one free block on every process post-redistribution. The extra blocks will initially be free. Theorem 2 guarantees the algorithm will complete and there will be at least one free block on each process. After all blocks are moved to their final positions, the extra block will again be free. We call this algorithm the Modified Basic Algorithm (MBA). For brevity, we will elide the details concerning the extra block, and just assume the input satisfies the hypothesis of Theorem 2.

4 Local Copy Efficient Algorithm

To this point we have ignored the question of local data movement. In [5], it was seen that in many examples, the time to perform local data copies could dominate the inter-process communication time. We now show how the MBA can be altered to deal efficiently with local data copying.

In each phase, the MBA sorts the blocks so that the data to be sent to a given process will be contiguous and can therefore be sent using a single MPI operation. The sort also makes the free space contiguous to facilitate the receives. Our implementation of MBA accomplishes this by sorting the blocks by increasing

```

1 procedure main is
2   Alltoall(out, 1, INT, in, 1, INT); nPhase  $\leftarrow$  0;
3   while  $\sum_i (\text{out}[i] + \text{in}[i]) > 0$  do computePhase();
4   sort blocks in phase order;
5   for  $i \leftarrow 0$  to nPhase  $- 1$  do executePhase(sched[i]);
6   move blocks to final positions;

7 procedure computePhase is
8    $R \leftarrow \emptyset$ ;  $S \leftarrow \emptyset$ ;  $r \leftarrow \text{free}$ ;
9   for  $i \leftarrow 0$  to  $n - 1$  do
10     $\text{recv}[i] \leftarrow \min\{r, \text{in}[i]\}$ ;  $r \leftarrow r - \text{recv}[i]$ ;
11    if  $\text{recv}[i] > 0$  then  $R \leftarrow R \cup \{(i, \text{recv}[i])\}$ ;
12    foreach  $\{i \mid \text{in}[i] > 0\}$  do
13       $\lfloor \text{post send of } \text{recv}[i] \text{ to proc } i; \text{in}[i] \leftarrow \text{in}[i] - \text{recv}[i]; \text{free} \leftarrow \text{free} - \text{recv}[i]$ ;
14    foreach  $\{i \mid \text{out}[i] > 0\}$  do post recv into send[i] for proc i;
15    wait for all requests to complete;
16    foreach  $\{i \mid \text{out}[i] > 0 \wedge \text{send}[i] > 0\}$  do
17       $\lfloor S \leftarrow S \cup \{(i, \text{send}[i])\}; \text{free} \leftarrow \text{free} + \text{send}[i]; \text{out}[i] \leftarrow \text{out}[i] - \text{send}[i]$ ;
18    if  $R \neq \emptyset \vee S \neq \emptyset$  then { sched[nPhase]  $\leftarrow (R, S)$ ; nPhase  $\leftarrow nPhase + 1$ ; }

19 procedure executePhase( $R, S$ ) is
20   foreach  $(i, q) \in R$  do post recv for  $q$  blocks from proc  $i$ ;
21   foreach  $(i, q) \in S$  do post send of  $q$  blocks to proc  $i$ ;
22   wait for all requests to complete;

```

Fig. 3. Local Copy Efficient Algorithm. A schedule *sched* is computed completely before it is executed. Each entry in *sched* is an ordered pair (R, S) . *R* encodes the number of blocks to receive from each process in the phase; *S* the number to send.

destination rank and placing all free blocks at the end. While this is a simple solution, it can lead to an explosion in local data copies. An example, in one of the maps considered in [5], all processes have the same number of blocks m , there are no free spaces, and the solution requires m phases. Each phase involves sending and receiving one block, and the local sort turns out to be the worst possible case, involving approximately m calls to `memcpy`. Hence the total number of copies performed by a process is approximately m^2 .

We now describe the Local Copy Efficient (LCE) Algorithm (Figure 3). This algorithm performs at most $O(m)$ local data copies on each process.

The first change to the earlier algorithm involves separating the process of schedule *computation* from that of schedule *execution*. (The same idea was employed in [5] to a very different algorithm, the “cyclic scheduler.”) In the new algorithm, *computePhase* is called repeatedly and the result accumulated in a *schedule*. After the entire schedule has been computed, *executePhase* is called repeatedly to carry out the actual sending and receiving of blocks.

A schedule is an array of *phase structures*. Each phase structure specifies the number of blocks to send to each process and the number to receive from each process, in that phase. Careful consideration must be given to the data structures used in a schedule. We have already seen it is possible for the number of phases to

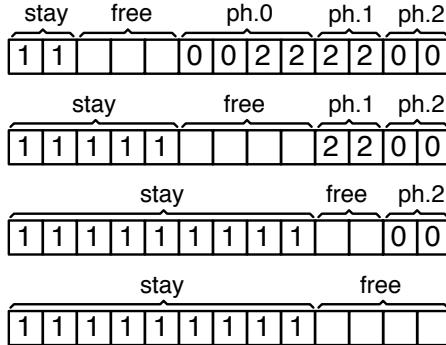


Fig. 4. Data layout to avoid local movement during schedule execution. Blocks are initially sorted in *phase order*. During phase i , blocks are received into the free region and sent from the region marked $\text{ph.}i$. No local data movement occurs between phases.

approach m , the number of blocks on one process. If each phase requires storing an array of length n then the memory consumed by a schedule can approach nm , an unacceptable level. In contrast, the worst-case memory requirements of the Basic Algorithm, and all algorithms discussed in [5, 6], are linear combinations of n , m , and the size of one block.

We therefore use a sparse format to represent a phase structure. It consists of a set R of pairs (i, in_i) for which $in_i > 0$, for the incoming data, and a set S of pairs (j, out_j) for which $out_j > 0$, for the outgoing data. In the worst case, the total number of blocks sent and the total number received are both m . Since each pair in the schedule represents the send or receive of at least one block, the total number of pairs is at most $2m$. Furthermore, a phase in which there is no incoming or outgoing data is simply not recorded, so the number of phase structures on a process is at most $2m$. Hence the memory consumed by the schedule is $O(m)$.

Once the schedule has been computed, the blocks are sorted in *phase order*: all blocks staying on the process appear first, followed by all free blocks, then all blocks departing in phase 0, followed by all blocks departing in phase 1, and so on. For each i , the outgoing blocks for phase i are further arranged by increasing destination rank, to create contiguous send buffers. (See Figure 4.)

Once the blocks are arranged in phase order, no local data movement is needed until the entire schedule has been executed. This is because incoming blocks are received into the free region while outgoing blocks are sent from the region immediately following the free region. Once a phase completes, the new free region is contiguous, consisting of perhaps part of the old free region (if not all free spaces were used to receive incoming data) and all of the old send region.

After schedule execution, a final local redistribution is required to move all blocks to their final positions. Hence the entire algorithm requires two local redistributions: one at the beginning to sort the blocks in phase order, and one at the end. An efficient local redistribution algorithm is described in [6]. That algorithm performs precisely the minimum number of copies possible; in

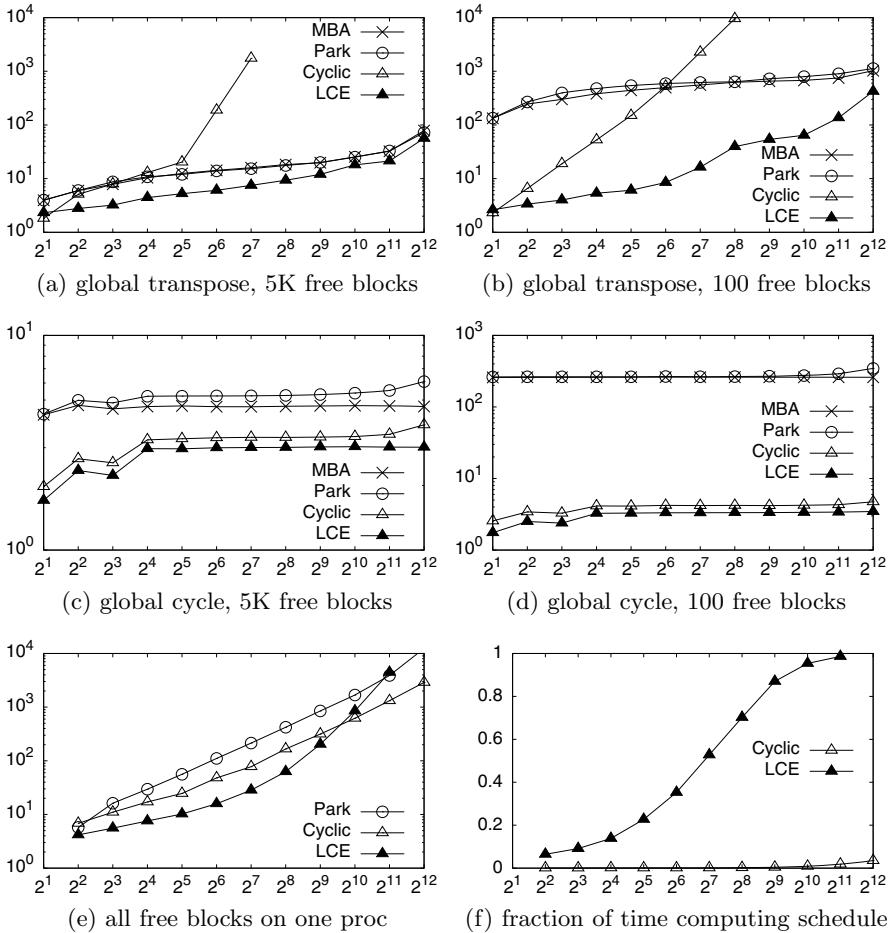


Fig. 5. Time to redistribute data. Each process maintains 25,000 blocks of 16,000 bytes each. In (a)–(e), x -axis is number of processes, y -axis is the time in seconds to complete a data redistribution.

particular, each block is moved at most 2 times. So the LCE algorithm performs at most $O(m)$ local copies.

5 Experiments and Conclusions

Experiments were run on Argonne’s *Full Disclosure*, a SiCortex 5832 with 972 nodes, each with 6 cores and 4 GB RAM (512 MB of which is reserved for fabric use). Each MPI process was mapped to one core. In all experiments, each process managed 25K blocks of 16 KB each, for a total of 400 MB per process. Each experiment was executed using $n = 2^k$ processes, where k was scaled to 12 or until execution time exceeded 10^4 seconds. This was repeated for

each of four redistribution algorithms: (1) MBA, (2) Park: a Pinar-Hendrickson “parking” algorithm, (3) Cyclic: the “cyclic scheduler” algorithm, and (4) LCE. (See [5, 6] for a detailed description of (2) and (3).) Timing results appear in Figure 5.

In experiment (a), each process has 5K free blocks and the map sends block j of process i to position $(mi + j)/n$ of process $(mi + j)\%n$, where $m = 20K$. (The map is essentially a global transpose of the data matrix.) Experiment (b) is similar but with only 100 free blocks per process. As explained in [5], the cyclic algorithm blows up on this map. In both cases, LCE performs better than the other algorithms for all $n > 2$, though it is not clear if this trend would continue for $n > 2^{12}$. The decreasing difference between the performance of LCE and MBA/Park in (b) appears to be due to the fact that the proportion of time devoted to schedule computation increases with n ; e.g, schedule computation consumes 202 of the 422 seconds for $n = 2^{12}$. We have not shown data on the memory consumed by the algorithms, but the numbers are generally small and the differences not great, e.g., 1060 KB per process for LCE vs. 845 KB for MBA at $n = 2^{12}$ in (b).

In experiments (c) and (d) all data from process i is sent to process $(i+1)\%n$. For (c), each process has 5K free blocks; in (d), 100 free blocks. Again LCE proves faster than the other algorithms, with Cyclic not far behind. The inefficiency of MBA/Park in these experiments is due primarily to the large number of local data copies performed between phases, which both Cyclic and LCE avoid.

In experiment (e), $n - 1$ processes have no free space; the last process has all 25K blocks free. Each of the $n - 1$ “full” processes divides its data into $n - 2$ approximately equally sized slices and sends one slice to each of the other full processes. MBA cannot complete within 10^4 seconds (even for $n = 2$), due to the enormous number of local data copies. Park, in contrast, takes advantage of the free space on the last process to greatly reduce the number of phases. Cyclic makes no use at all of the free process, but has the advantage that its schedule is computed in a single phase (see [5]). The experiment reveals that LCE performs better through $n = 2^9$, but beyond this point LCE is surpassed by Cyclic, and at $n = 2^{11}$ by Park. The blowup of LCE time again appears to be due to the time required to compute the schedule (f). For any full process, each phase involves sending and receiving one block, so there are a total of 25K phases. Scheduling a phase requires essentially an all-to-all communication among the $n - 1$ full processes. In contrast, to execute a phase each process communicates with only two other processes.

Conclusion. The LCE algorithm performs better than the other MADRE algorithms in most, but not all, cases. Our future work will look for ways to reduce the communication cost of schedule computation when free space is very limited. Other planned improvements include developing a local copy efficient version of the parking algorithm and a multi-threaded version of LCE for use in hybrid MPI/threaded programs.

References

1. Bertrand, F., Bramley, R., Sussman, A., Bernholdt, D.E., Kohl, J.A., Larson, J.W., Damevski, K.B.: Data redistribution and remote method invocation in parallel component architectures. In: IPDPS 2005: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2005) - Papers, Washington, DC, USA, p. 40.2. IEEE Computer Society Press, Los Alamitos (2005)
2. Bertrand, F., Yuan, Y., Chiu, K., Bramley, R.: An approach to parallel MxN communication. In: Proceedings of the Los Alamos Computer Science Institute (LACSI) Symposium, Santa Fe, NM (October 2003)
3. Park, N., Prasanna, V.K., Raghavendra, C.: Efficient algorithms for block-cyclic array redistribution between processor sets. In: Supercomputing 1998: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing (CDROM), Washington, DC, USA, 1998, pp. 1–13. IEEE Computer Society Press, Los Alamitos (1998)
4. Pinar, A., Hendrickson, B.: Interprocessor communication with limited memory. *IEEE Transactions on Parallel and Distributed Systems* 15(7), 606–616 (2004)
5. Siegel, S.F., Siegel, A.R.: MADRE: The Memory-Aware Data Redistribution Engine. In: Lastovetsky, A., Kechadi, T., Dongarra, J. (eds.) EuroPVM/MPI 2008. LNCS, vol. 5205, pp. 218–226. Springer, Heidelberg (2008)
6. Siegel, S.F., Siegel, A.R.: MADRE: The Memory-Aware Data Redistribution Engine. Technical Report UDEL-CIS 2009/335, Department of Computer and Information Sciences, University of Delaware (2009); *The International Journal of High Performance Computing Applications* (to appear)
7. Thakur, R., Choudhary, A., Fox, G.: Runtime array redistribution in HPF programs. In: Proc. of Scalable High Performance Computing Conference, pp. 309–316. IEEE, Los Alamitos (1994)

Impact of Node Level Caching in MPI Job Launch Mechanisms*

Jaidev K. Sridhar and Dhabaleswar K. Panda

Network-Based Computing Laboratory

The Ohio State University

2015 Neil Ave., Columbus, OH 43210 USA

{sridharj,panda}@cse.ohio-state.edu

Abstract. The quest for petascale computing systems has seen cluster sizes expressed in terms of number of processor cores increase rapidly. The Message Passing Interface (MPI) has emerged as the defacto standard on these modern, large scale clusters. This has resulted in an increased focus on research into the scalability of MPI libraries. However, as clusters grow in size, the scalability and performance of job launch mechanisms need to be re-visited.

In this work, we study the information exchange involved in the job launch phase of MPI applications. With the emergence of multi-core processing nodes, we examine the benefits of caching information at the node level during the job launch phase. We propose four design alternatives for such node level caches and evaluate their performance benefits. We propose enhancements to make these caches memory efficient while retaining the performance benefits by taking advantage of communication patterns during the job startup phase. One of our cache design – Hierarchical Cache with Message Aggregation, Broadcast and LRU (HCMAB-LRU) reduces the time involved in typical communication stages to one tenth while capping the memory used to a fixed upper bound based on the number of processes. This enables scalable MPI job launching for next generation clusters with hundreds of thousands of processor cores.

1 Introduction

The last few years has seen cluster sizes increase rapidly fueled by the increasing computing demands of parallel applications. The Top500 list [1], a biannual list of the top 500 supercomputers in the World, shows that the top-ranked cluster, the LANL RoadRunner [2] has 129,600 processor cores. In comparison, the largest cluster in the year 2000, the ASCI White had only 8,192 cores. As

* This research is supported in part by U.S. Department of Energy grants #DE-FC02-06ER25749 and #DE-FC02-06ER25755; National Science Foundation grants #CNS-0403342, #CCF-0702675 and #CCF-0833169; grant from Wright Center for Innovation #WCI04-010-OSU-0; grants from Intel, Mellanox, Cisco, and Sun Microsystems; Equipment donations from Intel, Mellanox, AMD, Advanced Clustering, Appro, QLogic, and Sun Microsystems.

clusters increase in node counts, an emerging trend is the increase in the number of processing cores per node due to the emergence of multi-core processors. For instance, the Sandia Thunderbird [3] cluster introduced in 2006 has $4K$ nodes – each with dual CPUs for a total of $8K$ processors while the TACC Ranger [4] cluster introduced in 2008 has $4K$ nodes – each with four quad-core CPUs for a total of $64K$ processors. The trend is likely to continue towards more cores per node, a fact underlined by the recent demonstration of an 80 core processor.

The Message Passing Interface (MPI) [5] has emerged as the defacto standard on these clusters. As clusters increase in size, there has been increased research into MPI libraries and their scalability. Most of these studies focus on reducing the memory footprint of the MPI libraries by reducing the memory used for interconnect connections [6] [7], by sharing resources between connections [8] [9] [10], or by establishing connections only when needed [11]. A more basic concern, the scalability of job launch mechanisms must also be addressed. Previously [12] we have proposed a Scalable and Extensible Job Launching Architecture for Clusters (ScELA) where we demonstrate an order of magnitude increase in startup performance on modern large scale clusters such as the TACC Ranger.

In this paper, we continue the study to analyze the communication between processes during job initialization with the goal to improve the startup performance. In particular we aim to study the performance benefits of caching information at the node level on multi-core processors. We propose four design alternatives for caching mechanisms that improve the performance of the startup phase in MPI applications. The first three designs include: Hierarchical Cache Simple (HCS), Hierarchical Cache with Message Aggregation (HCMA) and Hierarchical Cache with Message Aggregation and Broadcast (HCMAB). We study the communication pattern during a typical MPI job startup phase and enhance our designs to introduce the Hierarchical Cache with Message Aggregation, Broadcast and LRU (HCMAB-LRU) which is memory efficient while retaining the performance benefits of earlier methods. We evaluate our designs on a 512 core InfiniBand cluster to demonstrate the performance benefits of pre-populating node level caches. This reduces the time taken for a typical information stage during startup to less than one tenth while adhering to an upper bound on the memory used for the cache.

The rest of this paper is organized as follows. In Section 2 we describe the job launch phase and the ScELA framework briefly. We study the impact of node level caching and design alternatives in Section 3. We evaluate our designs in Section 4. We conclude and give future directions in Section 5.

2 Job Launch Phase

Figure 1 shows the phases involved during the launch of an MPI application. The job launcher is responsible for setting up the environment for MPI processes and starting the executables on target processors. Once MPI processes start execution, they invoke `MPI_Init` to initialize the MPI environment. This phase involves discovering peer processes and initializing communication channels and

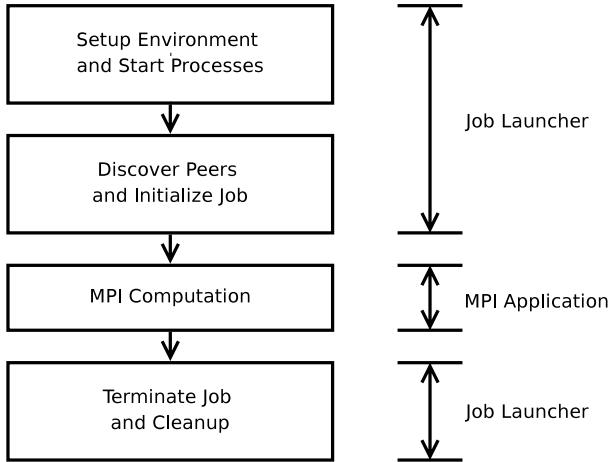


Fig. 1. Phases in the Launching of a Typical MPI Application with the Entities Responsible for Them

other features of the MPI library. For instance, in InfiniBand MPI libraries such as MVAPICH / MVAPICH2 [13], each MPI process needs to discover the InfiniBand Local IDentifiers (LIDs) and Queue Pair (QP) numbers. The job launcher has to facilitate this communication between MPI processes since the primary interconnect network is not initialized at this point. Additionally, the job launcher may also play a role during job termination at `MPI_Finalize`.

2.1 The ScELA Framework

Figure 2 shows the hierarchical architecture introduced in [12]. To minimize the number of connections created over the management network, we use Node Launch Agents (NLA) on each node. The NLAs are responsible for launching all MPI processes on a particular compute node. In addition to these, the NLAs facilitate communication between MPI processes during `MPI_Init`. MPI libraries such as MVAPICH [13] use a protocol called PMGR [14] which defines a set of MPI style collective operations for job initialization. Other MPI libraries such as MVAPICH2 [15], MPICH2 [16] and IntelMPI [17] use the Process Management Interface (PMI) [18] – a bulletin board like messaging protocol where MPI processes publish information as a `(key, value)` pair through the `PMI_Put` operation and other processes access this information through the `PMI_Get` operation.

In the ScELA framework, the main job launcher launches NLAs on all target nodes. The NLAs then connect to each other and form a hierarchical k -nomial tree to facilitate communication among MPI processes. The performance of this communication phase can be enhanced by using node level caches for protocols such as PMI. The use of caches enables us to serve `PMI_Get` requests faster from a local cache as well as reduce communication over the management network [12]. However, job launchers need to be minimalistic in their resource usage so that the

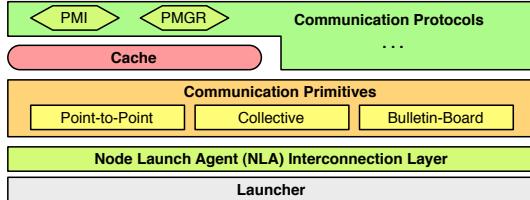


Fig. 2. The ScELA Framework [12]

performance of the MPI application is not adversely affected. In particular, the NLAs caches need to be memory efficient. In the following section, we propose several design alternatives for node level caches in NLAs and examine their memory efficiency and performance impact.

3 Proposed Designs

In this Section we describe alternative design mechanisms for caches over the hierarchical NLA network.

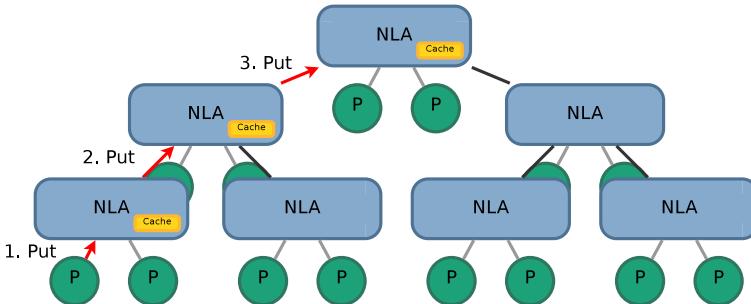
3.1 Hierarchical Cache Simple (HCS)

Figure 3 shows the basic caching mechanism in ScELA – the Hierarchical Cache Simple (HCS). When an MPI application publishes information with the `PMI_Put` operation, the data is sent to the local NLA. The NLA adds it to its local cache and forwards the data to the upper level NLA. The upper level NLAs add the information to their respective caches and forward it up the NLA tree.

When an MPI application requests for information via the `PMI_Get` operation, the local NLA checks if the data is available in its cache. If so, it responds with the data. If not, the request is forwarded up the NLA tree until an NLA finds the information in its cache. The response is cached in all intermediate NLAs as it travels down the tree. Since all `(key, value)` pairs get cached in the root of the NLA tree, worst case memory usage is $O(p \times n)$ where p is the number of processes and n is the number of `(key, value)` pairs published by each process.

3.2 Hierarchical Cache with Message Aggregation (HCMA)

In HCS, the number of messages sent over the NLA tree is large. The root of the NLA tree receives $O(p \times n)$ individual messages each containing a `(key, value)` pair. Sending a large number of small messages over the NLA tree is a costly operation. We observe that all MPI libraries ensure that MPI processes are synchronous when they publish and retrieve information. The typical information exchange phase can be summarized by the following pseudo-code:

**Fig. 3.** Hierarchical Cache Simple

```

PMI_Put (mykey, myvalue);
PMI_Barrier ();
...
val1 = PMI_Get (key1);
val2 = PMI_Get (key2);

```

Since we know that all MPI processes on a node publish information around the same time, we can aggregate this information at the local NLA before forwarding it up the tree. All processes must publish information for such a scheme to be used. The higher level NLAs wait for all local processes to publish information as well as aggregate information from all lower level NLAs and send a single aggregate message up the NLA tree after populating the local cache. Figure 4 shows such a mechanism. We call this mechanism as Hierarchical Cache with Message Aggregation (HCMA). With the message aggregation, we reduce the number of messages sent over the NLA tree. The root of the NLA tree receives $O(\log_k(p) \times n)$ messages (where k is the degree of the NLA tree) since it only receives one message from each NLA and smaller messages from the local MPI processes. The worst case memory usage is the same as HCS at $O(p \times n)$ which is related to the amount of data cached at the root NLA.

The handling of information retrieval in HCMA is identical to HCS.

3.3 Hierarchical Cache with Message Aggregation and Broadcast (HCMAB)

In both HCS and HCMA, when an MPI process requests for information that is not available in the local NLA cache, the request is forwarded up the NLA tree. The root NLA has data from all MPI processes. Thus the number of PMI_Get messages reaching the root is high and the responses need to travel multiple hops before the MPI application receives the data.

On multi-core clusters, we observe that due to the presence of multiple MPI processes on each compute node, most of the data would eventually be requested by some MPI process on the node. Thus, pre-populating all NLA caches would be beneficial in terms of reducing the number of messages sent over the network. This method – Hierarchical Cache with Message Aggregation and Broadcast

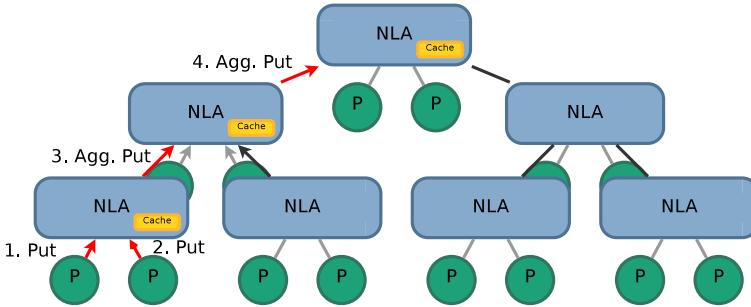


Fig. 4. Hierarchical Cache with Message Aggregation

(HCMAB) is similar to HCMA with one additional step. After the root NLA has accumulated data from all processes, it broadcasts an aggregate message to all NLAs. This ensures that all `PMI_Get` requests are served from local caches.

In this method, each NLA in the tree has memory requirement of the order of $O(p \times n)$ since all information is cached in all NLAs.

3.4 Hierarchical Cache with Message Aggregation, Broadcast with LRU (HCMAB-LRU)

Since HCMAB has the least number of messages traveling over the broadcast network and has all data requests served from a local cache, it offers the best performance of the three prior schemes. However populating every NLA cache involves memory usage of $O(p \times n)$ on every NLA where p is the number of MPI processes and n is the number of $(key, value)$ pairs published by every MPI process. Job launchers however need to keep their memory usage low. We observe that the information exchange during `MPI_Init` happens in stages where ranks publish information and retrieve information published by other ranks. The exchange is repeated in subsequent stages with new information. Thus we can limit memory used for the cache to $O(p)$ such that the cache can hold information required for one particular stage of information exchange. We use the Least Recently Used (LRU) cache replacement algorithm when storing data in the cache so that information from prior stages are discarded. We replace data from previous information exchange phases with data from newer exchange phases. We call this mechanism the Hierarchical Cache with Message Aggregation, Broadcast and LRU (HCMAB-LRU).

3.5 Comparison of Memory Usage

Table 1 shows a comparison of the memory usage between the four caching mechanisms for n rounds of information exchange. We observe that HCMAB-LRU is able to maintain a strict upper bound on its memory usage while the other schemes use increasing amount of memory as the amount of information exchanged increases.

Table 1. Node Level Memory Usage of Proposed Caching Mechanisms on Various Cluster Sizes for n published (*key, value*) pairs

MPI Job Size (p)	Caching Mechanism			
	HCS	HCMA	HCMAB	HCMAB-LRU
64	$O(64 \times n)$	$O(64 \times n)$	$O(64 \times n)$	$O(64)$
256	$O(256 \times n)$	$O(256 \times n)$	$O(256 \times n)$	$O(256)$
1024	$O(1024 \times n)$	$O(1024 \times n)$	$O(1024 \times n)$	$O(1024)$
4096	$O(4096 \times n)$	$O(4096 \times n)$	$O(4096 \times n)$	$O(4096)$
16384	$O(16384 \times n)$	$O(16384 \times n)$	$O(16384 \times n)$	$O(16384)$
65536	$O(65536 \times n)$	$O(65536 \times n)$	$O(65536 \times n)$	$O(65536)$

4 Performance Evaluation

In this Section we evaluate the four design alternatives presented in Section 3. Our testbed is a 64 node InfiniBand Linux cluster. Each node has dual 2.33 GHz Intel Xeon “Clovertown” quad-core processor for a total of 8 cores per node for a total of 512 processor cores. The nodes have a Gigabit Ethernet adapter for management traffic such as job launching. We represent cluster size as $n \times c$, where n is the number of nodes and c is the number of cores per node used.

We implement our design alternatives over the ScELA-PMI startup mechanism available in MVAPICH2 which uses PMI during the job launch phase.

4.1 Simple PMI (1:2) Exchange

We profile a part of the code where MVAPICH2 processes exchange information with two peers to form a ring connection over InfiniBand. In this phase each MPI process publishes one (*key, value*) pair using `PMI_Put` and retrieves `values` published by two other MPI processes. For instance, this kind of information exchange is used in MVAPICH2 to establish a ring network over InfiniBand. Figure 5 shows the performance of the caching mechanisms.

We see that HCS performs the worst since the number of messages traveling the NLA tree during both the publishing phase and the retrieval phase is highest. The HCMA scheme performs better since it reduces the number of messages traveling over the network during the publishing phase through message aggregation. The HCMAB and HCMAB-LRU schemes perform virtually identical since the communication patterns are identical. These schemes reduce the time taken for such a communication pattern to less than one third of HCS. Note that HCMAB-LRU uses much less memory since it does not retain information from previous stages of communication in the NLA caches.

4.2 Heavy PMI (1:p) Exchange

In another common form of information exchange, each MPI process publishes one piece of information. All p MPI processes then read information published

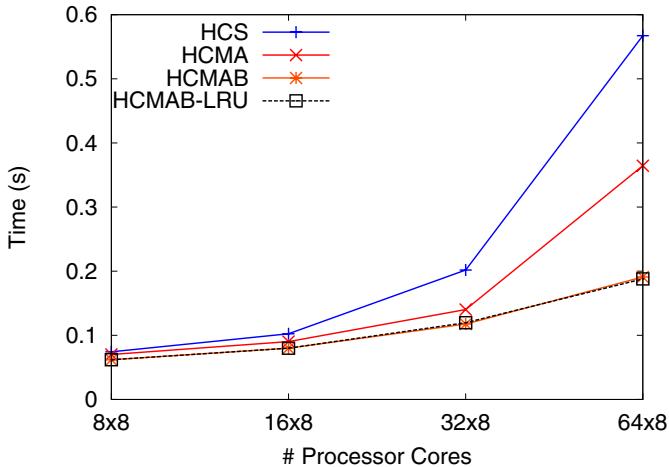


Fig. 5. Time Taken for a Simple PMI Operation with One Put and Two Gets

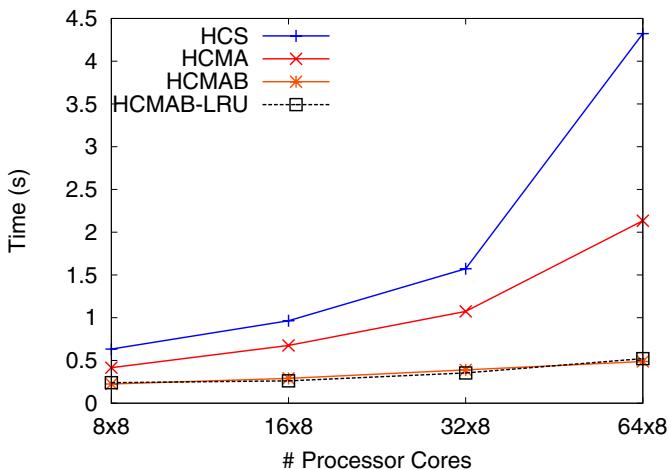


Fig. 6. Time Taken for a PMI Operation with One Put and p Gets

by every other MPI process. For instance, this method is used to learn the InfiniBand LIDs of all MPI processes in MVAPICH2. Figure 6 shows the results from profiling such an exchange. We observe the performance benefits of caching mechanisms such as HCMAB and HCMAB-LRU that populate all the caches prior to any PMI_Get requests from MPI processes. We observe that these mechanisms reduce the time taken for such exchanges to one tenth that of HCS. In these methods all data is served from the local NLA cache and the number of messages traveling through the NLA tree is minimal. These two factors improve the performance of the information exchange greatly. As the number of

messages exchanged is low, these mechanisms can scale to clusters of larger node counts.

5 Conclusion

Clusters continue to scale in processor counts. Node counts are increasing significantly but much of the growth is coming from multi-core clusters. In this light, improving the startup performance of MPI jobs has become all the more important. and has traditionally been a neglected area.

In this paper we have proposed four alternatives for node level caches in MPI job launchers. The simplest method – HCS improves the scalability and performance of the startup while keeping the average memory usage per node low in the job launcher. We improve the performance using message aggregation in HCMA. We propose an enhancement over this method that takes advantage of communication patterns used by typical MPI libraries that use PMI with HCMAB. We propose an enhancement over HCMAB to cap memory used to a fixed value. We reduce the performance of communication phases to around a tenth with our optimizations. Though we discuss our designs in terms of the k -nomial tree based ScELA framework, similar approach can be applied to node level caching on other startup mechanisms such as the ring based MPD in MPICH2 [16].

In the future, we propose to study node level caches and other optimizations over much larger clusters. We plan to evaluate distributed caching to reduce memory usage further in the presence of hundreds of thousands of MPI processes.

5.1 Software Distribution

Implementations of HCS and HCMAB schemes are integrated into the MVAPICH2 – a popular MPI library used by over 925 organizations Worldwide.

Acknowledgment

We would like to thank Mr. Jonathan L. Perkins from The Ohio State University for participating in the design process of the original ScELA framework.

References

1. TOP 500 Project: TOP 500 Supercomputer Sites (2009), <http://www.top500.org>
2. Los Alamos National Laboratory: Roadrunner, <http://www.lanl.gov/roadrunner/>
3. Sandia National Laboratories: Thunderbird Linux Cluster, <http://www.cs.sandia.gov/platforms/Thunderbird.html>
4. Texas Advanced Computing Center: HPC Systems, <http://www.tacc.utexas.edu/resources/hpcsystems/>
5. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard (1994)

6. Koop, M., Sridhar, J., Panda, D.K.: Scalable MPI Design over InfiniBand using eXtended Reliable Connection. In: IEEE Int'l Conference on Cluster Computing (Cluster 2008) (2008)
7. Koop, M., Jones, T., Panda, D.K.: MVAPICH-Aptus: Scalable High-Performance Multi-Transport MPI over InfiniBand. In: IEEE Int'l Parallel and Distributed Processing Symposium (IPDPS 2008) (2008)
8. Sur, S., Koop, M.J., Panda, D.K.: High-Performance and Scalable MPI over InfiniBand with Reduced Memory Usage: An In-Depth Performance Analysis. In: Super Computing (2006)
9. Sur, S., Chai, L., Jin, H.-W., Panda, D.K.: Shared Receive Queue Based Scalable MPI Design for InfiniBand Clusters. In: International Parallel and Distributed Processing Symposium (IPDPS) (2006)
10. Shipman, G., Woodall, T., Graham, R., Maccabe, A.: InfiniBand Scalability in Open MPI. In: International Parallel and Distributed Processing Symposium (IPDPS) (2006)
11. Yu, W., Gao, Q., Panda, D.K.: Adaptive Connection Management for Scalable MPI over InfiniBand. In: International Parallel and Distributed Processing Symposium (IPDPS) (2006)
12. Sridhar, J.K., Koop, M.J., Perkins, J.L., Panda, D.K.: ScELA: Scalable and Extensible Launching Architecture for Clusters. In: Sadayappan, P., Parashar, M., Badrinath, R., Prasanna, V.K. (eds.) HiPC 2008. LNCS, vol. 5374, pp. 323–335. Springer, Heidelberg (2008)
13. Network-based Computing Laboratory: MVAPICH: MPI over InfiniBand and iWARP, <http://mvapich.cse.ohio-state.edu>
14. Moody, A.: PMGR Collective Startup, <http://sourceforge.net/projects/pmgrcollective>
15. Huang, W., Santhanaraman, G., Jin, H.-W., Gao, Q., Panda, D.K.: Design of High Performance MVAPICH2: MPI2 over InfiniBand. In: Sixth IEEE International Symposium on Cluster Computing and the Grid, CCGRID 2006 (2006)
16. Argonne National Laboratory: MPICH2: High-performance and Widely Portable MPI, <http://www.mcs.anl.gov/research/projects/mpich2/>
17. Intel Corporation: Intel MPI Library, <http://software.intel.com/en-us/intel-mpi-library/>
18. Argonne National Laboratory: PMI v2 API, http://wiki.mcs.anl.gov/mpich2/index.php/PMI_v2_API

Towards Efficient MapReduce Using MPI

Torsten Hoefler¹, Andrew Lumsdaine¹, and Jack Dongarra²

¹ Indiana University, Open Systems Lab, Bloomington, IN, USA

{htor,lums}@cs.indiana.edu

² Department of Computer Science, University of Tennessee Knoxville

dongarra@eecs.utk.edu

Abstract. MapReduce is an emerging programming paradigm for data-parallel applications. We discuss common strategies to implement a MapReduce runtime and propose an optimized implementation on top of MPI. Our implementation combines redistribution and reduce and moves them *into the network*. This approach especially benefits applications with a limited number of output keys in the map phase. We also show how anticipated MPI-2.2 and MPI-3 features, such as `MPI_Reduce_Local` and nonblocking collective operations, can be used to implement and optimize MapReduce with a performance improvement of up to 25% on 127 cluster nodes. Finally, we discuss additional features that would enable MPI to more efficiently support all MapReduce applications.

1 Introduction

MapReduce [1,2] is an emerging programming framework to express data-parallel applications and algorithms. After its original cluster-based implementation, it has been implemented and evaluated on various architectures, including the Cell B.E. [3], GPUs [4], and multi-core processors [5]. Given the large problem sizes that are addressed using MapReduce, and given the popularity of MapReduce as an implementation paradigm, it seems natural to explore its use on traditional HPC platforms. Accordingly, in this work we discuss implementation and optimization of MapReduce functionality using MPI. Based on our experiences, we discuss several extensions to MPI that would enable more natural support of MapReduce.

The key functionality in MapReduce, i.e., **Map** and **Reduce**, are analogous to the functional programming constructs *map* and *fold*. These functional constructs are found in many modern programming languages. For example, C++’s Standard Library offers `std::transform()` and `std::accumulate()` that respectively provide functionality similar to map and fold.

The MapReduce model defines a two-step execution scheme that can be effectively parallelized. The user only defines two functions, $\mathcal{M} : (K_m \times V_m) \mapsto (K_r \times V_r)$ (map) which accepts input key-value pairs $(k, v) k \in K_m, v \in V_m$ and emits output key-value pairs $(g, w) g \in K_r, w \in V_r$, and a function $\mathcal{R} : (K_r, V_r^{\mathbb{N}}) \mapsto (K_r, V_r)$ (reduce) which accepts a key $g \in K_r$ and a list of values $v \in V_r^{\mathbb{N}}$ and generates a single return value. The MapReduce framework accepts

a list of input values $(K_m \times V_m)^N$, $N \in \mathbb{N}$, calls \mathcal{M} for each of the N inputs and collects the *emitted* result pairs. Then it groups all result pairs by their key g and calls \mathcal{R} for each key and the associated list of values. The functions \mathcal{M} and \mathcal{R} are defined to be pure functions (without side effects), i.e., they solely compute the output based on immutable inputs (without internal state). This means that the order of application of \mathcal{M} (and \mathcal{R} for different g) is independent and can be executed in parallel. Ordering constraints are only imposed by data-dependencies (i.e., all data must be produced before it can be consumed), such that no synchronization is necessary. However, we note that in the general case where any map task can emit any key $g \in K_r$, an implicit barrier exists between the map and reduce tasks. This will be discussed in Section 2.

Many data-parallel algorithms can be expressed in this framework. Prominent examples are sorting, counting elements in lists (e.g., words in documents), distributed search (grep), or transposition of graphs or lists [1]. More complex algorithms, such as Bellman Ford single source shortest path or PageRank [6] can be modeled by iteratively invoking MapReduce. It has also been shown that MapReduce can be applied in the context of advanced research in machine learning [7]. The purity of the two functions allows the runtime to perform several optimizations. The most obvious strategy is the concurrent execution of map and reduce tasks in the two phases. This efficient auto-parallelization has been proposed in [1]. Another very useful characteristic of MapReduce in large-scale systems is its inherent fault resiliency because map or reduce tasks on failed or slow nodes can simply be restarted on other nodes. Thus, MapReduce allows the application developer to focus on the important algorithmic aspects of his problem while allowing him to ignore issues like data distribution, synchronization, parallel execution, fault tolerance, and monitoring. Thus, MapReduce also gained significant popularity in education [8].

2 MapReduce Communication Requirements

We start by discussing the communication requirements of MapReduce and possible implementations. Figure 1 shows the two phases of MapReduce and the necessary communication (data dependencies). Data dependencies are:

- a **reading of input for \mathcal{M} :** the map tasks need to read (possibly large amounts of) input data of size $\Omega(N)$ (with N input pairs)
- b **building input lists for \mathcal{R} :** all pairs need to be ordered by keys $g \in K_r$ and the lists of total size $\mathcal{O}(N)$ need to be transferred to the reduce tasks
- c **output data of \mathcal{R} :** the reduce tasks output the data; this is usually a negligible operation of $\mathcal{O}(|K_r|)$ if we assume the common case $|K_r| \ll N$

Thus, the two critical data movements are collecting the input for the map tasks and arranging the output of the map tasks as input for the reduce tasks.

Required data movement obviously depends on the parallelization strategy. For example, if all map and reduce tasks are executed on a single node, then no data movement is necessary. However, parallelism is the most important feature of the

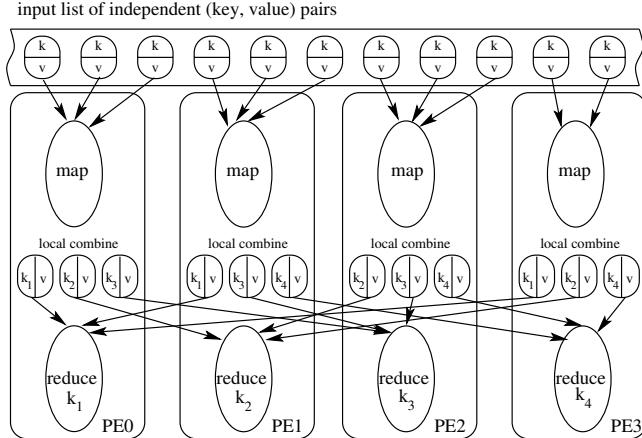


Fig. 1. MapReduce Communication Scheme

MapReduce model. The maximum parallelism of the embarrassingly parallel map phase is only limited by the number of input pairs N . At extremum, each application of \mathcal{M} is done as a separate map task on a separate processing element (PE). However, if P PEs are available, it is usually not advisable to process each reduce separately because the overhead of starting and administering tasks can be very high compared to a single application of \mathcal{M} and often $N \gg P$ holds. Thus, the input is commonly divided in reasonably sized work packets (e.g., 64 MiB in [1,9]) that are processed in parallel on all available PEs. A manager process can be used to administer the distribution of those packets and ensure proper actions in case of node-failures (e.g., resubmit the map tasks to other nodes). Multiple different strategies can be defined for reading the input data for each map task. The most common case is that a shared filesystem is available and the tasks just read the data directly from (local) disk. The manager tries to assign map tasks to nodes that are close to the needed data. However, mapping tasks close to data might not be possible in certain environments so that the map tasks need to collect the data before they can start. In the extreme case, only the manager process has access to the data and needs to distribute it to the workers.

The parallelism in the reduction phase is limited by the number of different output keys of the map phase ($|K_r|$) which highly depends on the implemented algorithm *and* the input data. Additionally, each application of \mathcal{R} needs *all* results of the map phase with the same key. Depending on the key distribution in the input pairs, this can effectively be an (irregular) all-to-all exchange with $P \times |K_r| \in \mathcal{O}(|K_r|^2)$ messages. It can again be tried to minimize the data movement with appropriate placement of the reduce tasks (close to their keys [10, 11]), however, this heavily depends on the input set and many applications do not seem to exhibit such a regularity (cf. parallel sort or string search). Thus, we have to assume the worst case, a synchronizing all-to-all exchange.

Throughout the remainder of the paper, we assume the hardest case where the input pairs are only available on the master process and the keys in K_r are

evenly distributed among all items in the input data. We note that MapReduce is often used to transform very large datasets, where the map tasks need to be moved to the data. However, the redistribution before the reduce phase can not easily be optimized for the general application in this way.

2.1 A Simple MPI Implementation

A straight-forward MPI implementation which is very similar to existing frameworks such as Hadoop [9] or Google’s implementation [1] can be easily implemented with MPI point-to-point communication. Each idle worker process queries the master for work packets. In the first stage, the master assigns map tasks to the processing elements. After all map tasks are done, the master disseminates reduce tasks. Each reduce task queries all available PEs for the values associated with its key. This method seems rather unfavorable because it tends to create many hot spots in the network (e.g., at the master process) and communication between map and reduce can easily be unbalanced potentially resulting in a severe performance degradation.

MPI has several mechanisms to optimize parallel computations. Two common optimization possibilities are (1) collective operations, which apply intelligent communication patterns to reduce congestion and minimize the number of message transmissions, and (2) overlapping communication and computation. Thus, a possible improvement would be to assign all reduce tasks to PEs before the second data exchange is started and perform it as a collective MPI_Alltoallv operation. While this has some potential to reduce network congestion, it is very tricky to optimize the general MPI_Alltoallv operation and it is generally accepted to be the least-scalable collective operation in MPI (due to the high number of unpredictable message exchanges $\mathcal{O}(P^2)$). Overlapping communication and computation can be used up to a certain extent if the master sends the input data for the map tasks in a pipelined fashion. However, it seems unreasonable to apply such techniques to the redistribution phase because reduce processes can only start when *all* input data is available.

This trivial implementation does not fully utilize all possible features of high-performance computing (HPC) systems. Thus, in the following section, we propose an orthogonal approach in order to take advantage of more scalable MPI functions and advanced optimization techniques.

3 Scalable MapReduce in MPI

In this section, we describe a different HPC-centric approach for the solution of the redistribution problem. We use a typical MapReduce program, the search for the number of given strings in a file, as example. The map function (Listing 1.1) accepts an input file (or a part of it) and a vector of strings s . It searches the input and emits the pair $(s, 1)$ for each occurrence of string s . The reduce function (Listing 1.2), which is also used as local combiner, simply counts the number of elements for a given s .

```

void map(filem f, keys strs) {
    for(i=0; i<strs.size(); i++) {
        char *ptr=f.start_addr();
        while(ptr<f.end_addr()-strs.len) {
            if(!memcmp(ptr, str[i], strs.len))
                EmitIntermediate(i, 1);
            ptr++;
        } }
}

```

Listing 1.1. Map Function

```

void reduce(key str, values num) {
    int sum=0;
    for(i=0; i<values.size(); i++) {
        sum += values[i];
    }
    Emit(sum);
}

```

Listing 1.2. Reduce Function

As discussed before, MapReduce readily lends itself to be implemented in a master-worker model, i.e., the computation (map and reduce tasks) can be scheduled by a central master process and executed by worker processes. MPI is well suited to implement this concept by using rank 0 as master process and all other $P - 1$ ranks as workers. We now discuss how to use collective operations to perform the map and reduce task. The map task can be implemented as a simple MPI_Scatter operation and the reduce task can use an MPI_Reduce operation. This basically moves the reduction function \mathcal{R} into the MPI library (i.e., the network layer). Either built-in reductions or MPI user-defined reduction operations, as described in [12] Section 5.9.5, can be used as \mathcal{R} . The implementation as user-defined reduction puts several limitations on \mathcal{R} and the input data:

- I \mathcal{R} must be associative (MPI reduction operations are generally assumed to be associative)
- II the number of different keys $|K_r|$ must be known by each process (or communicated in advance, e.g., with MPI_Allreduce) and, the values of all keys $g \in K_r$ must be fixed-size elements (can be arbitrary MPI datatypes). The output lists for each key can be reduced locally with a *combiner* [1]. MPI-2.2 will very likely include MPI_Reduce_local which can be used to implement the combiner trivially.
- III if not all processes have a value for each key $g \in K_r$, then an identity element with respect to \mathcal{R} has to exist and must be supplied by the processes without values.

Using scalable collective MPI reductions has a high optimization potential. It enables the application to take advantage of optimized implementations of collective operations. Several architectures, such as BlueGene [13], or networks, such as Quadrics [14], support hardware-optimized collective operations (they even perform some simple reduction operations directly in the network hardware). This implementation inherently supports the problematic case where the number of keys is small (the parallelism in the reduce phase is limited). In this case, tree-based algorithms can be used to reduce the number of messages from $\mathcal{O}(|K_r|^2)$ (cf. Section 2) down to $\mathcal{O}(\log_2(|K_r|))$ on fully connected networks. However, if many keys need to be reduced, the transmission time is likely going to be $\mathcal{O}(|K_r|)$ due to bandwidth limitations at the master node. The adaptation to different network topologies can be done with specialized collective operations. The reduction in our example fulfills all requirements and can be implemented as a

simple MPI_SUM reduction operation. We note that this approach to implement \mathcal{R} is fundamentally different to previous approaches. The reduction operations are not applied in parallel, but the data is reduced during the communication itself. Obviously, local memory and the size of each value limits the maximum number of keys. If this is an issue, the MapReduce framework could just fall back to the all-to-all-based parallel reduction approach or perform the reduce phase in multiple MPI_Reduce steps.

3.1 A Simple MapReduce Example

The string search example shows the applicability of the MPI-based MapReduce scheme well. However, we decided to implement a more flexible application that is able to simulate a large class of MapReduce programs. This application accepts four parameters: the number of tasks, minimum and maximum task duration t_{min}, t_{max} , and the size of the reduction operation s_{red} . The application issues work-packets that take a random time in the closed interval $[t_{min}, t_{max}]$. The times are uniformly distributed. A worker process retrieves a work-packet and simulates the map function \mathcal{M} by computing for the duration of time indicated in the packet. After the computation is finished, the worker starts the parallel reduction $\mathcal{R}=\text{MPI_SUM}$ of size s_{red} (by calling MPI_Reduce). The execution scheme is shown in Figure 2(a).

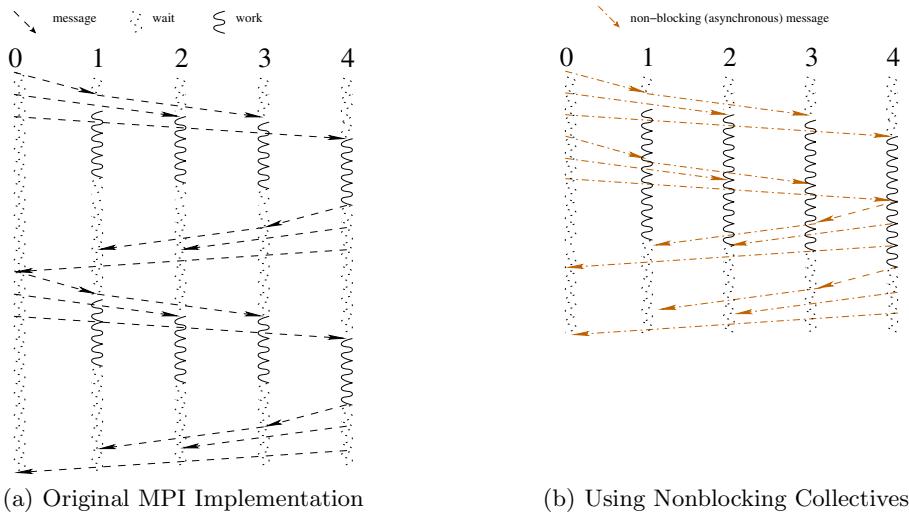


Fig. 2. HPC-centric MapReduce Scheme assuming 5 processes executing 8 tasks and a binomial tree scatter and reduction algorithm

3.2 Further Optimization Possibilities

Common optimizations for parallel programs are optimizing communication patterns with collective operations, which we have already done, and hiding latency

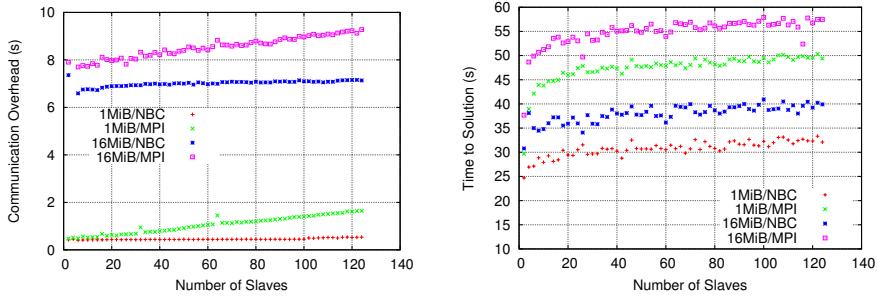
by overlapping communication and computation. We discussed in Section 2.1 that overlapping can be done with pipelining data in the map phase but that the redistribution phase can not be transformed easily. In our optimized scheme, where the data redistribution and the reduce phase are merged, a similar pipelining scheme can be utilized. However, to retain the benefits of optimized collective operations, we would need a nonblocking version of those operations to enable overlap. Such nonblocking collective operations are proposed for MPI-3 [15] and a reference implementation exists with LibNBC [16].

We can easily apply nonblocking collective operations to the map as well as the reduce functionality. In the map-case, the master starts w_m nonblocking `NBC_Iscatter` operations at the beginning. Then, it waits for the first one to complete and starts the next operation after one completed. This efficiently creates a *window* of scatter operations that run in the background. Their latency can be ignored if the window size is large enough and the work-packets take long enough to compute. The trade-off there is that the memory requirements grow with the window size, i.e., data for all running operations has to be in main memory at the root process. The worker processes similarly start w_m non-blocking scatter operations and re-post new non-blocking scatter until the signal to exit is received. In the reduce case, we have to define a set of w_r buffers to support w_r outstanding nonblocking operations. A similar window-technique as in the map operation is used to have w_r outstanding `NBC_Ireduce` operations at any time. If all n buffers are in use, the *oldest* reduction has to be finished by calling `NBC_Wait` on the master process and re-posting a new `NBC_Ireduce`. The remaining outstanding communications have to be finished and their buffers reduced when all tasks are completed. The resulting execution scheme is shown in Figure 2(b).

3.3 Performance Results

MapReduce applications typically process large amounts of data that have to be read from either the network or local disks. Thus, we assume that the I/O bandwidth is not sufficient to keep multiple processing elements busy. However, most of today’s systems are multi-core or SMP systems such that there are idle cores available to offload the communication. We use the threaded InfiniBand-optimized version of LibNBC [17, 18] for all benchmarks. This efficiently results in offloading the reduce task to another core (the reduce operation is a part of the `NBC_Reduce` communication) and thus utilizes another level of functional parallelism transparently to the application developer. Benchmarks of the simple string-search example were also covered by the more extensive simulator and delivered exactly the same results. Thus, we only present benchmark results for the different configurations of the simulator.

We benchmarked two different workload-scenarios with 1 to 126 worker nodes with 10 tasks per process. We compared the threaded version of LibNBC with a maximum of 5 outstanding collective operations with Open MPI 1.2.6. We also varied the data-size of the reduction operation (in our example, we used `MPI_SUM` as the reduction operation).



(a) Communication Overhead of Static Workload
(b) Time to Solution of Dynamic Workload

Fig. 3. Overhead and Time to Solution for Static and Dynamic Workloads for different Number of Workers

Figure 3(a) shows the communication and synchronization overhead for a static workload of 1 second per packet. Using nonblocking collective results in a significant performance increase because nearly all communication can be overlapped. The remaining communication overhead is due to InfiniBand's memory registration which is done on the host CPU. The graphs show a reduction of communication and synchronization overhead of up to 27%. Figure 3(b) shows the influence of nonblocking collectives to dynamic workloads varying between 1 ms and 10 s. The significant performance increase is due to avoidance of synchronization and the use of communication/computation overlap. This clearly shows that our technique can be used to benefit MapReduce-like applications significantly. The dynamic example shows improvements in time to solution of up to 25% over the unoptimized implementation.

4 What Is MPI Missing?

MapReduce was not intended to be implemented on top of MPI, and MPI lacks several features that are needed to implement it efficiently.

One of the most important features of MapReduce is its ability to handle faults transparently. However, the default error handling in MPI (`MPI_ERRORS_ARE_FATAL`) is to abort the job. The user can change this default to return an error from the failed function (`MPI_ERRORS_RETURN`). While this failure mode might help for point-to-point communications, collective communication can not easily recover from such an error state. Moreover, checking if a collective operation completed successfully on all nodes is not trivial (or very expensive). One alternative is the use of intercommunicators for point-to-point communications [19], but again, collective communication is not handled. An efficient and fault-tolerant implementation would require extended fault tolerance support in MPI. This does not necessarily mean changes in the API [19], but it should support rebuilding of communicators and (partial) restarting of collective communications. First results in this direction have been demonstrated by FT-MPI [20].

A second barrier in the usage of MPI for general MapReduce algorithms are the restrictions of the intermediate data and of the function \mathcal{R} as discussed in Section 3. In order to support arbitrary MapReduce operations, MPI would need variable-sized reduction operations because \mathcal{R} can return values that are of a different size than the input values, e.g., string concatenations. Such a feature would also be desired in the support of higher languages, such as C# [21].

Another feature that can be used to optimize the implementation as shown in Section 3.2 are nonblocking collective operations [15]. A proposal for this class of operations is discussed by the MPI Forum for inclusion in MPI-3.

5 Conclusions and Future Work

MapReduce and MPI were developed in two different communities that have traditionally been somewhat disjoint. However, as the needs and capabilities of these two communities continue to converge, it will be to the benefit of both to leverage their respective technologies. In the case of MapReduce and MPI, we have seen that is possible to efficiently implement MapReduce using MPI – with some limitations. For example, HPC-centric optimizations can be applied if the reduce function fulfills certain criteria. Additional performance gains are possible through upcoming MPI features. Using nonblocking collective operations, for example, provided a speedup of up to 25% over the blocking implementation.

Fully supporting MapReduce will require several additional features and capabilities from MPI. However, many of these features are generally recognized as being important, particularly as MPI evolves to support other modern programming and parallelization paradigms.

Acknowledgments

This work is supported by a grant from the Department of Energy project FastOS II (LAB 07-23) and a grant from the Lilly Endowment.

References

1. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 107–113 (2008)
2. Lämmel, R.: Google’s MapReduce programming model – Revisited. *Sci. Comput. Program.* 68, 208–237 (2007)
3. de Kruijf, M., Sankaralingam, K.: MapReduce for the CELL B.E. Architecture. *IBM Journal of Research and Development* 52 (2007)
4. He, B., Fang, W., Luo, Q., Govindaraju, N.K., Wang, T.: Mars: a MapReduce framework on graphics processors. In: *PACT 2008: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pp. 260–269. ACM, New York (2008)

5. Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., Kozyrakis, C.: Evaluating MapReduce for Multi-core and Multiprocessor Systems. In: HPCA 2007: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture, Washington, DC, USA, pp. 13–24. IEEE Computer Society, Los Alamitos (2007)
6. Langville, A.N., Meyer, C.D.: Google's PageRank and Beyond: The Science of Search Engine Rankings. Princeton University Press, Princeton (2006)
7. Chu, C.T., Kim, S.K., Lin, Y.A., Yu, Y., Bradski, G.R., Ng, A.Y., Olukotun, K.: Map-Reduce for Machine Learning on Multicore. In: Schölkopf, B., Platt, J.C., Hoffman, T. (eds.) NIPS, pp. 281–288. MIT Press, Cambridge (2006)
8. Kimball, A., Michels-Slettvet, S., Bisciglia, C.: Cluster computing for web-scale data processing. SIGCSE Bull. 40, 116–120 (2008)
9. Hadoop (2009), <http://hadoop.apache.org>
10. Pike, R., Dorward, S., Griesemer, R., Quinlan, S.: Interpreting the data: Parallel analysis with Sawzall. Scientific Programming 13, 277–298 (2005)
11. Ghemawat, S., Gobioff, H., Leung, S.T.: The Google file system. SIGOPS Oper. Syst. Rev. 37, 29–43 (2003)
12. Message Passing Interface Forum: MPI: A Message Passing Interface Standard, Version 2.1 (2008)
13. Gara, A., et al.: Overview of the Blue Gene/L system architecture. IBM Journal of Research and Development 49, 195–213 (2005)
14. Petrini, F., Frachtenberg, E., Hoisie, A., Coll, S.: Performance Evaluation of the Quadrics Interconnection Network. Journal of Cluster Computing 6 (2003)
15. Hoefler, T., Kambadur, P., Graham, R.L., Shipman, G., Lumsdaine, A.: A Case for Standard Non-Blocking Collective Operations. In: Cappello, F., Herault, T., Dongarra, J. (eds.) PVM/MPI 2007. LNCS, vol. 4757, pp. 125–134. Springer, Heidelberg (2007)
16. Hoefler, T., Lumsdaine, A., Rehm, W.: Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI. In: Proceedings of the 2007 International Conference on High Performance Computing, Networking, Storage and Analysis, SC 2007. IEEE Computer Society/ACM (2007)
17. Hoefler, T., Lumsdaine, A.: Optimizing non-blocking Collective Operations for InfiniBand. In: Proceedings of the 22nd IEEE International Parallel & Distributed Processing Symposium, IPDPS (2008)
18. Hoefler, T., Lumsdaine, A.: Message Progression in Parallel Computing - To Thread or not to Thread? In: Proceedings of the 2008 IEEE International Conference on Cluster Computing. IEEE Computer Society Press, Los Alamitos (2008)
19. Gropp, W., Lusk, E.: Fault Tolerance in MPI Programs. Special issue of the Journal High Performance Computing Applications (IJHPCA) 18, 363–372 (2002)
20. Fagg, G.E., Angskun, T., Bosilca, G., Pjesivac-Grbovic, J., Dongarra, J.: Scalable Fault Tolerant MPI: Extending the Recovery Algorithm. In: Di Martino, B., Kranzmüller, D., Dongarra, J. (eds.) EuroPVM/MPI 2005. LNCS, vol. 3666, pp. 67–75. Springer, Heidelberg (2005)
21. Gregor, D., Lumsdaine, A.: Design and implementation of a high-performance MPI for C# and the common language infrastructure. In: PPoPP 2008: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 133–142. ACM Press, New York (2008)

Process Arrival Pattern and Shared Memory Aware Alltoall on InfiniBand

Ying Qian and Ahmad Afsahi

Department of Electrical and Computer Engineering
Queen's University

Kingston, On, Canada K7L 3N6

ying.qian@ece.queensu.ca, ahmad.afsahi@queensu.ca

Abstract. Recent studies have shown that processes in real applications can arrive at the collective calls at different times. This imbalanced process arrival pattern can significantly affect the performance of the collective operations. MPI_Alltoall() is a communication-intensive collective operation that is used in many parallel scientific applications. Its efficient implementation under different process arrival patterns is critical to the performance of applications that use them frequently. In this paper, we propose novel RDMA-based process arrival pattern aware MPI_Alltoall() algorithms over InfiniBand clusters. We extend the algorithms to be shared memory aware for small to medium size messages. The micro-benchmark and application results indicate that the proposed algorithms outperform the native implementation as well as their non-process arrival pattern aware counterparts when processes arrive at different times.

Keywords: Process Arrival Pattern, Collective Communications, MPI_Alltoall(), RDMA, InfiniBand.

1 Introduction

Most scientific applications are developed on top of the *Message Passing Interface* (MPI) [1]. Such applications extensively use MPI collective communication operations. Most research on developing and implementing efficient collective communication algorithms assume all MPI processes involved in the operation arrive at the same time at the collective call. However, it has been recently shown that *process arrival patterns* (PAP) for collectives are adequately imbalanced that will adversely affect the performance of collective communications [2]. In addition, it has been found that different collective communication algorithms react differently to PAP [2]. In this regard, the authors in [3] have recently proposed PAP aware *MPI_Bcast()* algorithms and implemented them using MPI point-to-point primitives.

InfiniBand [4] has been introduced to support the ever-increasing demand for efficient communication, scalability, and higher performance in clusters. It supports *Remote Direct Memory Access* (RDMA) operations that allow a process

to directly access the exposed memory areas of a remote process. RDMA is a one-sided read, write, or atomic operation, offloaded to the network interface card. MPI implementations over RDMA-enabled networks such as InfiniBand are able to effectively bypass the operating system overhead and lower the CPU utilization.

In this paper, we take on the challenge to design and efficiently implement PAP aware *MPI_Alltoall()* collective on top of InfiniBand. In *MPI_Alltoall()*, each process has a distinct message for every other process. It is a communication-intensive operation that is typically used in linear algebra operations, matrix multiplication, matrix transpose, and multi-dimensional FFT. Therefore, it is very important to optimize its performance on emerging multi-core clusters in the presence of different process arrival patterns. Our research is along the work in [3], however it has a number of significant differences. First, the authors in [3] have incorporated control messages in their algorithms at the MPI layer to make the processes aware of and adapt to the PAP. These control messages incur high overhead, especially for short messages. Our proposed PAP aware *MPI_Alltoall()* algorithms instead is RDMA-based, and we use its inherent mechanism for notification purposes. Therefore, there are no control messages involved and thus there is no overhead. Secondly, while [3] is targeted at large messages, we propose and evaluate two RDMA-based schemes for small and large message. Thirdly, we propose an intra-node PAP and shared memory aware scatter operation to boost the performance for small messages.

Our performance results indicate that the proposed PAP aware *MPI_Alltoall()* algorithms perform better than the native MVAPICH [5] and the non-PAP aware algorithms when processes arrive at different times. Our RDMA-based PAP and shared memory aware algorithm is the best algorithm up to 256B messages and gains up to 3.5 times improvement over MVAPICH. The proposed RDMA-based PAP aware algorithm is the algorithm of choice for 512B to 1MB messages; it outperforms the native MVAPICH by a factor of 3.1 for 8KB messages.

The rest of the paper is organized as follows. In Sect. 2, we provide an overview of InfiniBand and ConnectX adapter from Mellanox Technologies [6], as well as the implementation of *MPI_Alltoall()* in MVAPICH. Section 3 discusses the motivation behind this work by presenting the performance of *MPI_Alltoall()* in MVAPICH when processes arrive at the collective at random times. Section 4 presents our proposed PAP and shared memory aware *MPI_Alltoall()* algorithms. In Sect. 5, the performance of the proposed algorithms on a four-node multi-core cluster is presented. Related work is discussed in Sect. 6. Section 7 concludes the paper.

2 Background

InfiniBand [4] is an I/O interconnection technology consisting of end nodes and switches managed by a central subnet-manager. End nodes use Host Channel Adapters (HCA) to connect to the network. InfiniBand verbs form the lowest level of software to access the InfiniBand protocol-processing engine offloaded to

the HCA. The verbs layer has queue-pair based semantics, in which processes post send or receive work requests to send or receive queues, respectively. InfiniBand supports both the channel semantics (send-receive) and the memory semantics (using one-sided RDMA operations). RDMA-based communication requires the source and destination buffers to be registered to avoid swapping memory buffers before the DMA engine can access them.

ConnectX [6] is the most recent generation of InfiniBand HCAs by Mellanox Technologies. Its architecture includes a stateless offload engine for protocol processing that improves the performance by having hardware schedule the packet processing directly. This technique allows the ConnectX to have a better performance for processing simultaneous network transactions, as used in our algorithms.

In MVAPICH [5], point-to-point and some MPI collective communications have been implemented directly using RDMA operations. However, `MPI_Alltoall()` uses the two-sided MPI send and receive primitives, which transparently uses RDMA. Different algorithms are employed in `MPI_Alltoall()` for different message sizes: the *Bruck* [7] algorithm for small messages, the *Recursive-Doubling* [8] algorithm for large messages and power of two number of processes, and the *Direct* algorithm for large messages and non-power of two number of processes.

3 Motivation

To show how the native MVAPICH `MPI_Alltoall()` perform on our platform under random PAP, we use a micro-benchmark similar to [3]. Processes first synchronize using an *MPI_Barrier()*. Then, they execute a random computation before entering the `MPI_Alltoall()`. The random computation time is bounded by a maximum value MIF (*maximum imbalanced factor* [3]) times the time it takes to send a message.

The experiment was conducted on a 4-node, 16-core ConnectX cluster, described in Sect. 5. To get the performance of `MPI_Alltoall()`, a high-resolution timer is inserted before and after the `MPI_Alltoall()` operation. The completion time is reported as the average execution time across all the processes. Figure 1 presents the performance of MVAPICH `MPI_Alltoall()` when MIF is 1, 32, 128 and 512, respectively. Clearly, the completion time is greatly affected by the increasing amount of random computation. The results confirm the findings in [2] that PAP can have significant impact on the performance of collectives. It is therefore crucial to design and implement PAP aware collectives to improve their performance and consequently the performance of the applications that use them frequently.

4 The Proposed Process Arrival Pattern Aware `MPI_Alltoall()`

In this section, we propose our PAP and shared memory aware algorithms for `MPIAlltoall()`. The basic idea in our algorithms is to let the early-arrival

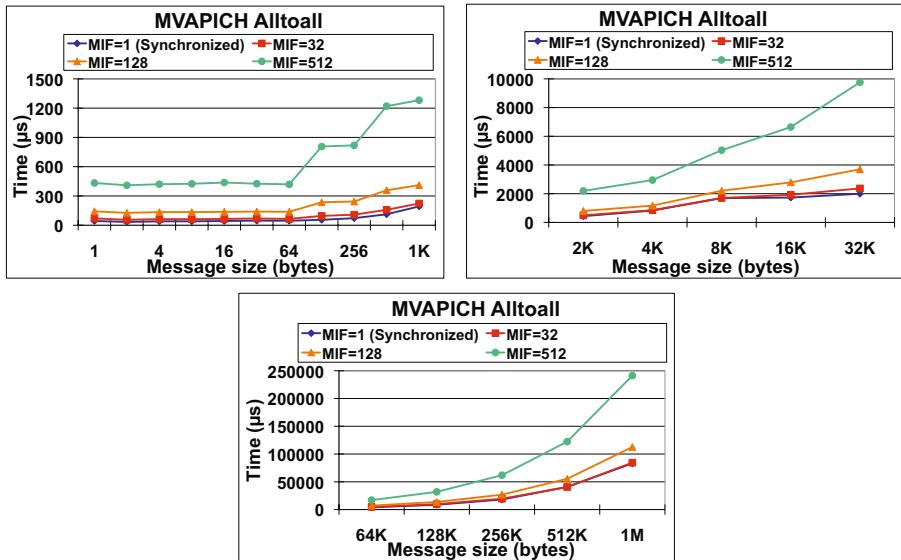


Fig. 1. Performance of MVAPICH MPI_Alltoall() for different process arrival patterns

processes do as much work as possible. One critical issue in our PAP aware algorithms is how to let every other process know who has already arrived at the call. Previous work on PAP aware MPI_Bcast() [3] has introduced control messages that would add extra overhead, especially for small messages. However, in our work we do not send distinct control messages and instead we utilize the inherent features of RDMA-based communication to notify the arrival of a process.

4.1 Notification Mechanisms for Early-Arrival Processes

The basic idea in our PAP aware MPI_Alltoall() is for each process to send its distinct data to the already-arrived processes as soon as possible. It is therefore very important to have an efficient mechanism in place to inform others of the early-arrival processes. For this, we have devised two different notification mechanisms for *zero-copy* and *copy-based* schemes used in RDMA-based communications. These notification mechanisms do not incur any communication overhead.

In the zero-copy approach, where the cost of data copy is prohibitive for large messages, the application buffers are registered to be directly used for data transfer. However, for an RDMA Write message transfer to take place each source process needs to know the address of the remote destination buffers. For this, each process will advertise its registered destination buffer addresses to all other processes by writing into their pre-registered and pre-advertised control buffers. This inherent destination address advertisement mechanism can be interpreted as a control message to indicate a process has arrived at the MPI_Alltoall() call.

Therefore, processes can poll their control buffers to understand which other process has already arrived at the collective call.

To avoid the high cost of application buffer registration for small messages, the copy-based technique involves a data copy to pre-registered and pre-advertised intermediate data buffers at both send and receive sides. The sending process can copy its messages to the pre-registered intermediate destination buffers using RDMA Write. Therefore, the received data in the pre-registered intermediate destination buffer can be used as a signal that the sending process has already arrived at the site. This can be checked out easily by polling the intermediate destination buffer.

4.2 RDMA-Based Process Arrival Pattern Aware Direct Algorithm

Our base algorithm is the Direct alloverall algorithm. Let N be the total number of processes involved in the operation. In this algorithm, at step i , process p sends its message to process $(p + i) \bmod N$, and receives a message from process $(p - i) \bmod N$. To implement this algorithm, each process p first posts its RDMA Writes to all other processes in sequence (after it receives the destination buffer addresses). It then polls the completion queues to make sure its messages have been sent to all other processes. Finally, it waits to receive the incoming messages from all processes.

To make this algorithm PAP aware using zero-copy scheme, each process p polls its control buffers for the advertised remote destination buffer addresses starting from process $(p + i) \bmod N$. It then sends its distinct data to the final destination buffers of the early-arrived processes using RDMA Write. Subsequently, it waits for the remaining processes to arrive in order to send its messages to them. Finally, each process waits for all incoming messages by polling its own destination buffers. The beauty of this PAP aware algorithm over the non-PAP aware algorithm is that a sending process will never get stuck for a particular process to arrive in order to proceed with the next message transfer.

Under the copy-based scheme, each process p polls its intermediate destination buffers, starting from process $(p - i) \bmod N$. Any received data indicates that the corresponding process has already arrived. The process p then copies its messages using RDMA Write to all early-arrived processes. It then sends its data to the rest of processes who have not yet arrived. All processes also need to wait to receive messages from all other processes into their intermediate buffers, and then copy them to their final destination buffers.

4.3 RDMA-Based Process Arrival Pattern and Shared Memory Aware Direct Algorithm

Up to this point, we have utilized the RDMA features of InfiniBand along with the PAP awareness in an effort to improve the performance of MPI_Alltoall(). Previous research has shown that shared memory intra-node communication can improve the performance of collectives for small to medium size messages [9]. It is interesting to see how this might affect the performance under different PAP.

For this, we propose a shared memory and RDMA-based PAP aware Direct algorithm for MPI_Alltoall() that has the following three phases:

Phase 1: Intra-node shared memory gather performed by a master process.

Phase 2: Inter-node process PAP aware Direct alltoall among the masters.

Phase 3: Intra-node PAP and shared memory aware scatter by a master.

A master process is selected for each node (without loss of generality, the first process in each node). Phase 1 cannot be PAP aware because the master process has to wait for all intra-node messages to arrive into a shared buffer before moving on to the PAP aware Phase 2. Phase 2 is the same as the algorithm proposed in Sect. 4.2, and is performed among the master processes. In Phase 3, an intra-node shared memory and PAP aware scatter is devised. Because the master processes may arrive in Phase 2 at different times, this awareness can be passed on to Phase 3 by allowing the intra-node processes to copy their destined data available in the shared buffer to their final destinations without having to wait for data from all other masters.

In a shared memory but non-PAP aware Phase 3, a master process waits to receive the data from all other master processes. It then copies them all to a shared buffer and sets a shared *done* flag. All other intra-node processes poll on this flag, and once set they start copying their own data from the shared buffer to their final destinations.

In the shared memory and PAP aware Phase 3, we consider multiple shared done flags, one for the data from each master process (four flags in our 4-node cluster). As soon as a master process receives data from any other master process, it copies it to the shared buffer and then sets the corresponding *done* flag. All other intra-node processes poll on all *done* flags, and as soon as any partial data is found in the shared buffer they copy them to their final destination buffers.

5 Experimental Results

The experiments were conducted on a 4-node dedicated multi-core cluster, where each node is a Dell PowerEdge 2850 server having two dual-core 2.8GHz Intel Xeon EM64T processors (2MB of L2 cache per core) and 4GB of DDR-2 SDRAM. Each node has a two-port Mellanox ConnectX InfiniBand HCA installed on an x8 PCI-Express slot. Our experiments were done under only one port of the ConnectX HCA operating in InfiniBand mode. The machines are interconnected through a Mellanox 24-port MT47396 Infiniscale-III switch. In terms of software, we used the OpenFabrics Enterprise Distribution, OFED-1.2.5 [10], installed over Linux Fedora Core 5, kernel 2.6.20. For MPI, we used MVAPICH-1.0.0-1625.

5.1 Micro-benchmark Results

In this section, we present the performance results of the proposed algorithms, the RDMA-based PAP aware Direct (*PAP_Direct*), and RDMA-based PAP and

Shared-memory aware Direct (*PAP_Shm_Direct*), and compare them with the non-PAP aware RDMA-based Direct (*Direct*) and RDMA-based and Shared-memory aware Direct (*Shm_Direct*) algorithms as well as with the native MVAPICH on our cluster.

We have evaluated the proposed algorithms using both copy-based and zero-copy techniques for 1B to 1MB messages. The results shown in this section are the best results of the two schemes for each algorithm. We use cycle-accurate timer to record the time spent in an MPI_Alltoall() (1000 iterations) for each process, and then calculate the average time across all processes.

Figure 2 compares our PAP aware algorithms with the native MVAPICH implementation and non-PAP aware versions, with MIF equal to 32 and 512. Clearly, the PAP aware algorithms, *PAP_Direct* and *PAP_Shm_Direct*, are better than their non-PAP aware counterparts for all message sizes. This shows that indeed such algorithms can adapt themselves well with different PAP. Our

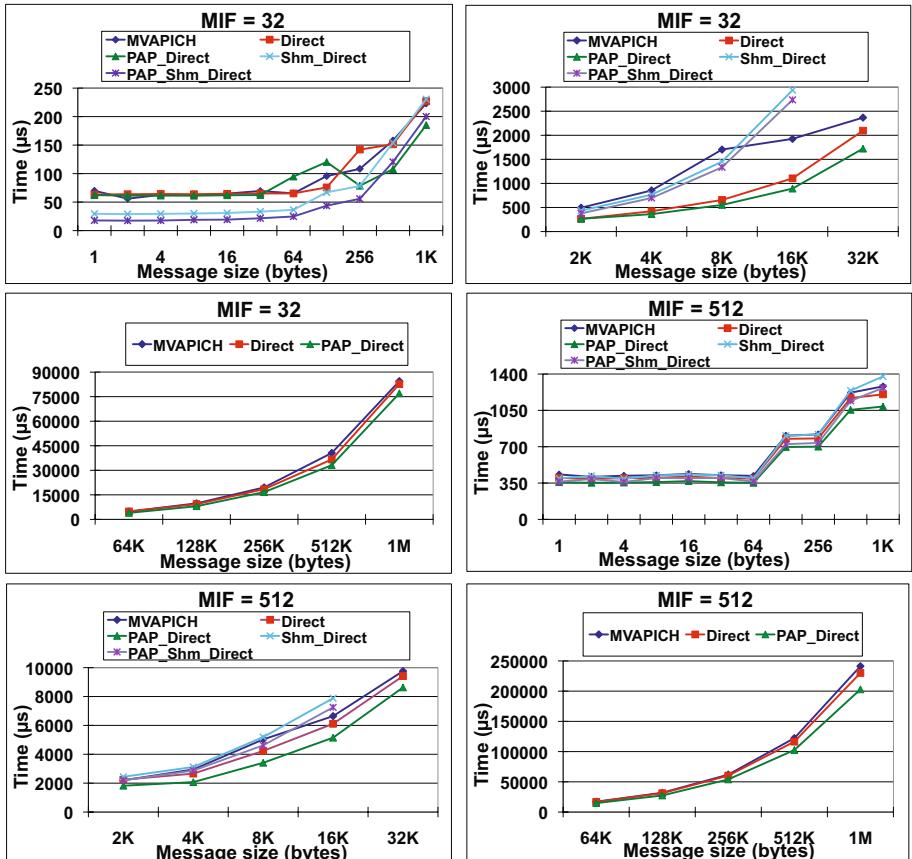


Fig. 2. Performance of MPI_Alltoall() running with 16 processes

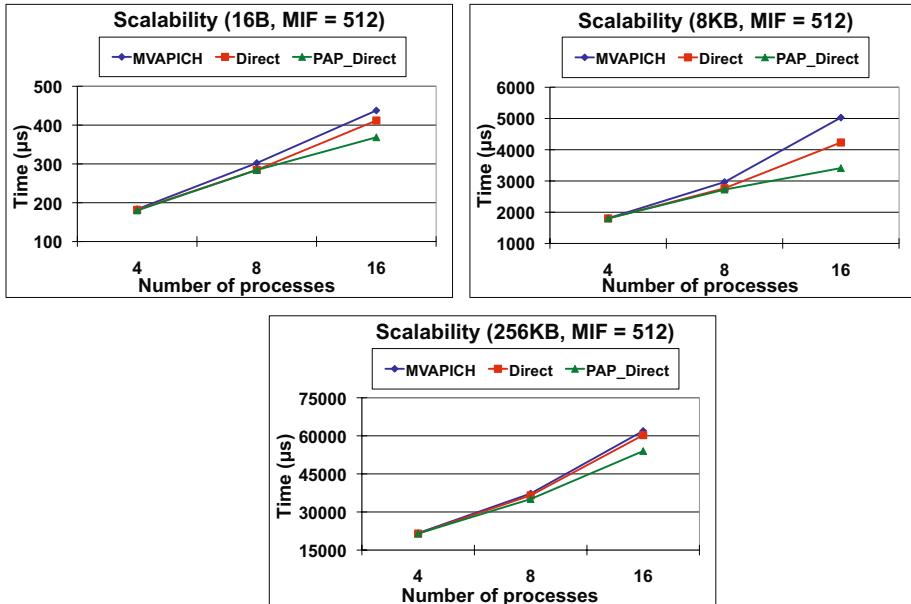


Fig. 3. MPI_Alltoall() scalability

algorithms are also superior to the native MVAPICH, with an improvement factor of 3.1 at 8KB for PAP_Direct and 3.5 at 4B for PAP_Shm_Direct, with MIF equal to 32. With a larger MIF of 512, the improvements are 1.5 and 1.2, respectively.

Comparing the PAP_Shm_Direct with PAP_Direct, one can see that the PAP_Shm_Direct is the algorithm of choice up to 256 bytes for MIF equal to 32. However, this is not the case for MIF of 512 where processes may arrive at the call with more delay with respect to each other. This shows that the shared memory version of our algorithm introduces some sort of implicit synchronization in Phase 1 that may degrade its performance under large maximum imbalanced factors.

To evaluate the scalability, we compare the performance of the PAP_Direct MPI_Alltoall() with those of MVAPICH and Direct for 4, 8, and 16 processes, as shown in Fig. 3 (shared memory algorithms are not shown due to limited data points). One can see that the proposed PAP aware algorithm has scalable performance and is always superior to the non-PAP aware algorithms. We have found similar results for other MIFs and message sizes.

In the previous micro-benchmark, the arrival time of each process is random. In another micro-benchmark, similar to [3], we control the number of late processes. In Fig. 4, we present the results for MIF equal to 128 when 25% or 75% of processes arrive late. Our proposed algorithms are always better than their counterparts for the 25% case, and mostly better in the 75% case. The PAP_Shm_Direct is always better than MVAPICH, although with a less margin in the 75% case.

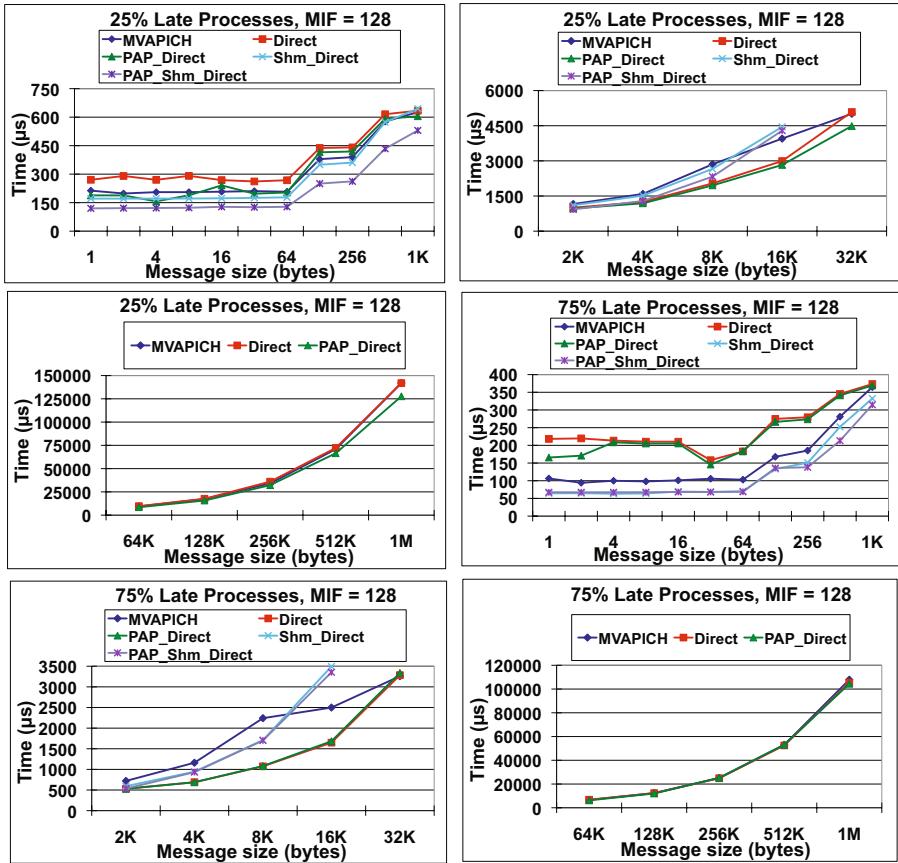


Fig. 4. Performance of MPI_Alltoall() with 25% and 75% late processes, 16 processes

Table 1. PAP_Direct MPI_Alltoall() speedup over native MVAPICH and the Direct algorithms for NAS FT running with different number of processes and classes

	Speedup over native MVAPICH algorithm		Speedup over Direct algorithm	
	FT (class B)	FT (class C)	FT (class B)	FT (class C)
4 processes	1.08	1.01	1.16	1.04
8 processes	1.10	1.04	1.04	1.14
16 processes	1.14	1.17	1.42	1.63

5.2 Application Results

In this section, we consider the FT application benchmark from NAS 2.4 [11] to evaluate the performance and scalability of the proposed PAP aware MPI_Alltoall(). FT uses MPI_Alltoall() as well as a few other collectives. We have experimented

with class B and C of FT, running with different number of processes, which use payloads larger than 2MB. Table 1 shows the PAP aware MPI_Alltoall() speedup over the native MVAPICH and the Direct algorithms for FT running with 4, 8, and 16 processes. Clearly, the proposed algorithm outperforms the conventional algorithms. The results also show that the PAP aware MPI_Alltoall() has modest scalability as speedup improves with increasing number of processes.

6 Related Work

Study of collective communications has been an active area of research. Thakur and his colleagues [8] discussed recent collective algorithms used in MPICH [12]. They have shown some algorithms perform better depending on the message size and the number of processes involved in the operation.

Faraj and his associates [2] discussed the PAP in a set of MPI programs, which denotes the timing when different processes arrive at an MPI collective operation. They discovered that the time difference between the arrivals of each process at a collective call is usually large enough to affect the performance of the collective. An MPI broadcast across different PAP was proposed by Patarasul and Yuan [3], in which they inserted control messages in their algorithms to make the processes aware of and adapt to the PAP. Their algorithms were implemented at the MPI layer using MPI point-to-point operations.

Sur and others [13] proposed RDMA-based MPI_Alltoall() algorithms for InfiniBand clusters. Buntinas et al. [14] used different mechanisms to improve large data transfers in SMP systems. Tippalraj and others overlapped the shared memory and remote memory access communications in devising collectives [15]. Qian and Afsahi proposed efficient RDMA-based and shared memory aware all-gather at the Elan-level over multi-rail QsNet^{II} clusters [9], and on InfiniBand ConnectX using its multi-connection capabilities [16].

7 Conclusions and Future Work

MPI_Alltoall() is one of the most communication-intensive primitives in MPI. Imbalanced PAP has an adverse impact on its performance. In this paper, we proposed RDMA-based and shared memory PAP aware MPI_Alltoall() algorithms without introducing any extra control messages.

The performance results indicate that the proposed algorithms perform better than their non-process arrival pattern counterparts when processes arrive at different times. They also outperform the native MVAPICH implementation by a large margin, up to 3.1 times at 8KB for PAP_Direct and 3.5 times at 4B for PAP_Shm_Direct.

While this study was focused at MPI_Alltoall(), it can be directly extended to other collectives. The proposed techniques can be applied to other alltoall algorithms such as Bruck or recursive doubling. However, one has to bear in mind that due to synchronization between different steps of these algorithms they may not achieve the highest performance as in the Direct algorithm. We are currently

investigating this. We also intend to experiment with a larger testbed and study other collectives to understand the true potential of PAP aware algorithms.

Acknowledgments. This work was supported in part by the Natural Sciences and Engineering Research Council of Canada Grant #RGPIN/238964-2005, and Canada Foundation for Innovation and Ontario Innovation Trust Grant #7154. We would like to thank Mellanox Technologies for the resources.

References

1. MPI: A Message Passing Interface standard (1997)
2. Faraj, A., Patarasuk, P., Yuan, X.: A Study of Process Arrival Patterns for MPI Collective Operations. *International Journal of Parallel Programming* 36(6), 543–570 (2008)
3. Patarasuk, P., Yuan, X.: Efficient MPI_Bcast across Different Process Arrival Patterns. In: 22nd International Parallel and Distributed Processing Symposium, IPDPS (2008)
4. InfiniBand Architecture, <http://www.infinibandta.org>
5. MVAPICH, <http://mvapich.cse.ohio-state.edu>
6. Mellanox Technologies, <http://www.mellanox.com>
7. Bruck, J., Ho, C.-T., Kipnis, S., Upfal, E., Weathersby, D.: Efficient Algorithms for All-to-all Communications in Multiport Message-passing Systems. *IEEE Transactions on Parallel and Distributed Systems* 8(11), 1143–1156 (1997)
8. Thakur, R., Rabenseifner, R., Gropp, W.: Optimization of Collective Communication Operations in MPICH. *International Journal of High Performance Computing Applications* 19(1), 49–66 (2005)
9. Qian, Y., Afsahi, A.: Efficient Shared Memory and RDMA based Collectives on Multi-rail QsNet^{II} SMP Clusters. *Cluster Computing, Journal of Networks, Software Tools and Applications* 11(4), 341–354 (2008)
10. OpenFabrics Alliance Homepage, <http://www.openfabrics.org>
11. NAS Benchmarks, version 2.4,
<http://www.nas.nasa.gov/Resources/Software/npb.html>
12. MPICH, <http://www.mcs.anl.gov/research/projects/mpich2>
13. Sur, S., Jin, H.-W., Panda, D.K.: Efficient and Scalable All-to-all Personalized Exchange for InfiniBand Clusters. In: 33rd International Conference on Parallel Processing (ICCP), pp. 275–282 (2004)
14. Buntinas, D., Mercier, G., Gropp, W.: Data Transfers Between Processes in an SMP System: Performance Study and Application to MPI. In: 35th International Conference on Parallel Processing (ICPP), pp. 487–496 (2006)
15. Tippalaju, V., Nieplocha, J., Panda, D.K.: Fast Collective Operations using Shared and Remote Memory Access Protocols on Clusters. In: 17th International Parallel and Distributed Processing Symposium, IPDPS (2003)
16. Qian, Y., Rashti, M.J., Afsahi, A.: Multi-connection and Multi-core Aware All-Gather on InfiniBand Clusters. In: 20th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS), pp. 245–251 (2008)

How Formal Dynamic Verification Tools Facilitate Novel Concurrency Visualizations*

Sriram Aananthakrishnan¹, Michael DeLisi¹, Sarvani Vakkalanka¹,
Anh Vo¹, Ganesh Gopalakrishnan¹, Robert M. Kirby¹, and Rajeev Thakur²

¹ School of Computing, Univ. of Utah, Salt Lake City, UT 84112, USA

² Math. and Comp. Sci. Div., Argonne Nat. Lab., Argonne, IL 60439, USA

Abstract. With the exploding scale of concurrency, presenting valuable pieces of information collected by formal verification tools intuitively and graphically can greatly enhance concurrent system debugging. Traditional MPI program debuggers present trace views of MPI program executions. Such views are redundant, often containing equivalent traces that permute independent MPI calls. In our ISP formal dynamic verifier for MPI programs, we present a collection of alternate views made possible by the use of formal dynamic verification. Some of ISP's views help pinpoint errors, some facilitate discerning errors by eliminating redundancy, while others help understand the program better by displaying concurrent even orderings that must be respected by *all* MPI implementations, in the form of *completes-before* graphs. In this paper, we describe ISP's graphical user interface (GUI) capabilities in all these areas which are currently supported by a portable Java based GUI, a Microsoft Visual Studio GUI, and an Eclipse based GUI whose development is in progress.

1 Introduction

Program debugging tools and their graphical user interfaces must strive to meet three goals: (i) locate errors in the tool's range reliably, and display the errors intuitively, (ii) eliminate redundant work in locating the errors, and correspondingly avoid presenting redundant work, (iii) depict items of interest so that users gather deep insights about their programs and the APIs/libraries they use. These three goals are even more important to meet for concurrent program debuggers, because concurrent program executions are often far less intuitive than sequential program executions. This paper is on a family of new graphical user interfaces (GUI) that we have equipped our previously reported In-Situ Partial order (ISP, [1,2,3,4]) tool with, as our first attempt to meet the above goals. Information is presented by the ISP tool through a portable Java based GUI and an optional

* Supported in part by NSF CNS-00509379, Microsoft HPC Institutes Program, and the Mathematical, Information, and Computational Science Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

Microsoft Visual Studio GUI, and an Eclipse GUI (development in progress) that will integrate and extend these views.

Being a formal verification tool, ISP guarantees to locate deadlocks, C assertion violations, resource leaks, and many invalid uses of MPI calls for a chosen application and its test harness. Being a dynamic verification tool, ISP does all its verification by “cleverly” running the MPI application on a standard computer (say, a laptop). The main idea (cleverness) in ISP is in the fact that it explores only *certain* process schedules, after which it proclaims the program to be free from these classes of errors. A concurrent program with five processes, with each process taking part in five sequential steps has potentially 10 billion interleavings (schedules).¹ However, it can be easily seen that large groups of interleavings are totally equivalent: either none of them trigger a bug, or they all do. For example, an interleaving in which an `MPI_send` is posted before its matching non-deterministic `MPI_Recv` is equivalent to the one in which things are posted oppositely (these are *independent actions* [5]). A conventional testing tool does not take advantage of this independence, and may present all the interleavings as traces, when in fact the essence of all interleavings might have been summarized by picking *any one* of the interleavings. By avoiding this exponential interleaving exploration, ISP can both run far more efficiently and also present only the *relevant* interleavings.

ISP determines what is a relevant interleaving at runtime. The main idea can be explained using the example of an MPI program containing one wildcard receive (`P0:: recv(from *)`) and three potentially matching MPI sends (`P1::send(to P0)`, `P2::send(to P0)`, and `P3::send(to P0)`). ISP will, at runtime, (i) determine all MPI sends that can match the wildcard receive (say only the sends from P1 and P2 can match P0’s receive), (ii) dynamically rewrite the wildcard receive to the two specific receives, and (iii) replay the entire program for each such match (*i.e.*, issue `P0:: recv(from P1)`, run the whole program, restart all MPI processes, and run again, now issuing `P0:: recv(from P2)`). In this manner, ISP generates relevant alternate interleavings only if non-deterministic actions are present. For programs that are heavily non-deterministic, ISP will explore a large number of interleavings. However, in practice, most MPI programs are deterministic. For ParMETIS, a 14k LOC program, ISP generated exactly one interleaving [2]; the number of possible interleavings is astronomical.

The results produced by any tool (including ISP) cannot be trusted if the underlying MPI library does not conform to the MPI standard, if the user does not model a sufficient number of processes, or the user drives the application with limited ranges of data. More than these well-known restrictions of any tool, there are some specific issues associated with MPI debugging tools. The MPI standard provides significant latitude in many places (for example, in terms of forward progress guarantees), and different MPI library implementations exploit this latitude in different (and often incompatible) ways. Since ISP is run using one of these libraries, it must be ideally ensured that its findings are applicable

¹ The mathematical derivation to calculate the number of riffle shuffles of five decks of five cards applies here.

across all MPI libraries. In this connection, one of the most important features in the ISP tool is the notion of *completes-before* (CB). We define *completes-before* to be the partial order of MPI action occurrences that *must be guaranteed by all standards-compliant MPI libraries*. When ISP runs, it builds the completes-before graph, and takes scheduling decisions based on it. This ensures that ISP can discover the full extent of non-determinism (for instance, all sends matching a wildcard receive) regardless of the specific MPI library being coupled with ISP, *so long as the library is MPI standard compliant*. In terms of specific details, ISP achieves this end goal by intercepting user's MPI program actions using the PMPI mechanism, and often deliberately delays and/or rearranges the issue order of these actions into the MPI runtime.

After encountering many non-intuitive (but correct) MPI examples, we realized that the completes-before graph can help users understand such examples. The display of completes-before is an important part of ISP's GUI.

Since MPI is a complex and rich API, it is important for MPI program verification tools to display helpful information about MPI operation scheduling. With the growing use of API functions with complex non-blocking and/or out of order completion semantics in concurrent programming, concurrency debugging tools that explicate API behavior can prove to be invaluable to their users. We later give an example concerning MPI probes that illustrates this issue.

2 Background, Related Work

Frameworks such as the Eclipse Parallel Tools Platform (PTP, [6]), Microsoft's Visual Studio, and TotalView are popular among MPI programmers. Shaeli [7] has explored MPI program trace visualizations. These tools do not have ideas similar to ours. The tool CHESS [8,9] from Microsoft Research focuses on thread verification. It includes many useful features such as displaying schedule strings that lead to an error, and pre-emption control techniques that help users root-cause errors. Because of the differences between threading and message passing, these ideas are not directly applicable to MPI.

A Jumpshot (or other [10]) view of an MPI program execution driven by simple test harnesses often shows the temporal clustering and recurrence of MPI communications. Such tests (and hence visualizations) rarely involve the elusive 'corner case' interleavings that lead to bugs. A designer may try to 'jiggle' the schedule by inserting random delay statements. While often effective for thread programs, this rarely has the intended effect with MPI programs for several reasons, the main reason being that in most MPI programs, executions fall into a handful of equivalence classes. Thus, most schedule perturbations cause only the order of independent actions to swap. In addition, random delay statements add delays that affect the computational speed even where things do not matter.

Consider a simple example of an MPI program execution where three MPI commands are posted – one being a wildcard receive and the other two being two competing sends that try to match this receive. After all these commands are posted, the external world has *no* control over which send matches the wildcard

receive. Inserting noisemakers into MPI library implementation codes is infeasible in most cases (*e.g.*, commercial MPI libraries). Even if arranged through PMPI, there is only so much MPI runtime control one can obtain. In other words, a designer is left with ineffective control over permuting MPI non-deterministic choices. We need a systematic search over all such *dependency inducing* situations. Dynamic partial order reduction (DPOR, [11]) offers that route. Our brand of MPI-specific DPOR is Partial Order avoiding Elusive interleavings (POE) [1] – the workhorse behind our ISP tool [1,2,3,4]. This paper is on equipping ISP – that has powerful core algorithms – with a commensurately powerful set of user-interface facilities. With this combination, users are not flooded in their visual displays with unnecessary (equivalent) schedule variations – a major hindrance when learning MPI and/or when debugging a large-scale MPI program.

High-performance libraries such as MPI have many subtle features that routinely confound even MPI experts. We have given the following “quiz” to many MPI experts and found them most often failing the test. The quiz is: *Can the MPI_Irecv of P2 be matched by the MPI_Isend(to P2) issued by P1?*².

```
P0: MPI_Isend(to P2, &h) ; MPI_Barrier() ; MPI_Wait(&h);...
P1: MPI_Barrier() ; MPI_Isend(to P2, &h); MPI_Wait(&h);...
P2: MPI_Irecv(from *, &h) ; MPI_Barrier() ; MPI_Wait(&h);...
```

To summarize, today’s MPI debugging tools have the following deficiencies:

- They flood the graphical view with relevant (few) and irrelevant (most) variations of executions, unable to highlight which is which. This causes cognitive overload.
- They are not equipped with means to determine *relevant* schedules (nothing like a DPOR algorithm for instance), as well as means to *enforce* these schedules during dynamic verification. This causes bugs to be missed. A revealing display is the short programs listed in [12] where such omissions can be starkly seen.
- They do not educate MPI users concerning the subtleties of the API. They do not inform MPI users to anticipate portability bugs – where speed-paths change. (In the above quiz, in some platforms, P0’s send may be the one found matching the wildcard receive most of the time; in a new platform, P1’s send may be the one.)

POE Recap: Here, we summarize how POE handles our ‘quiz’ above: ISP will (i) trap the MPI operations using PMPI, (ii) delay issuing the `Isend` from P0 and `Irecv` from P2, (iii) come to `MPI_Barrier` which is now issued into the MPI runtime, and (iv) now find that `MPI_Isend` of P1 is also eligible to match with P2’s `Irecv`. At this point, ISP dynamically rewrites `MPI_Irecv(from *)` into `MPI_Irecv(from P0)` and `MPI_Irecv(from P1)`, and does the following. It first issues the set $\{P2:\text{MPI_Irecv}(\text{from P0}), P0:\text{MPI_Isend}(\text{to P2})\}$ and finishes the rest of the program according to POE. Then, in another restart and replay of the entire program, it will issue the set $\{P2:\text{MPI_Irecv}(\text{from P1}), P0:\text{MPI_Isend}(\text{to P2})\}$.

² The sly answer is: “Yes, otherwise we would not be asking this question!”

P1:MPI_Isend(to P2)} and finishes the rest of the program according to POE. Changing the order in which the MPI operations from P0 and P1 are processed is justified because of their *independence* [5] – an assumption met by all MPI libraries we have come across. In fact, what POE is trying is to simulate all possible orderings and platform conditions under which the executions may happen. Notice that ISP does not generate different interleavings corresponding to the posting order of P0' and P2's first MPI calls.

Our Contributions: ISP has been released [13] for free download, runs on Windows, MAC OS/X, and Linux, handles several MPI libraries, and has been very effective in finding subtle bugs in large programs [2]. ISP has been used to verify the following large-scale programs, including: (1) the 14k LOC ParMETIS hypergraph partitioner (for two processes, we can verify ParMETIS in under five seconds on a laptop), (2) MADRE [14], (3) the MPI-BLAST program for genome sequencing [15], (4) the ADLB program [16], and (5) the Inter Radiosity Solver (IRS) from LLNL [17]. ISP has also handled most of the examples from Pacheco's widely used MPI book [18], thus making it an ideal tool for learning MPI. We now summarize features of ISP's GUI.

- All displays of ISP's user interface are arranged through a detailed trace file that is post-verification processed. The size of the trace file is small because of POE that generates and displays only the relevant interleavings.
- For alternate wildcard matches, ISP uses intuitive graphics (dotted lines for alternative matches). This way, one can visualize non-deterministic choices without “display explosion”.
- ISP has Visual Studio (VS) integration as well as a Java GUI. Its Eclipse integration is in progress. The VS interface makes ISP truly a push-button model-checker of actual MPI codes – a user-friendly debugger look and feel.
- The Visual Studio interface displays all communication matches, opening windows dynamically based on the number of matches to be shown.
- It allows the user to cut into the underlying Visual Studio debugger at any selected highlight point (*e.g.* communication match or deadlock). This allows a smooth transition into conventional debugging when needed (a facility that we hope to develop further during ISP's integration into the Eclipse PTP).
- The fact that many collectives do not possess the barrier semantics comes out elegantly through the CB graph.
- Mouse-driven tool tips reveal the details of the underlying MPI commands.

3 Basics of ISP's User Interface

Consider the MPI example:

```
P0: Isend(to 1, &h) ; Barrier() ; Wait(&h) ; ...
P1: Irecv(from *, &h); Barrier() ; Recv(from 2); Wait(&h); ...
P2: Barrier() ; Isend(to 1, &h); Wait(&h) ; ...
```

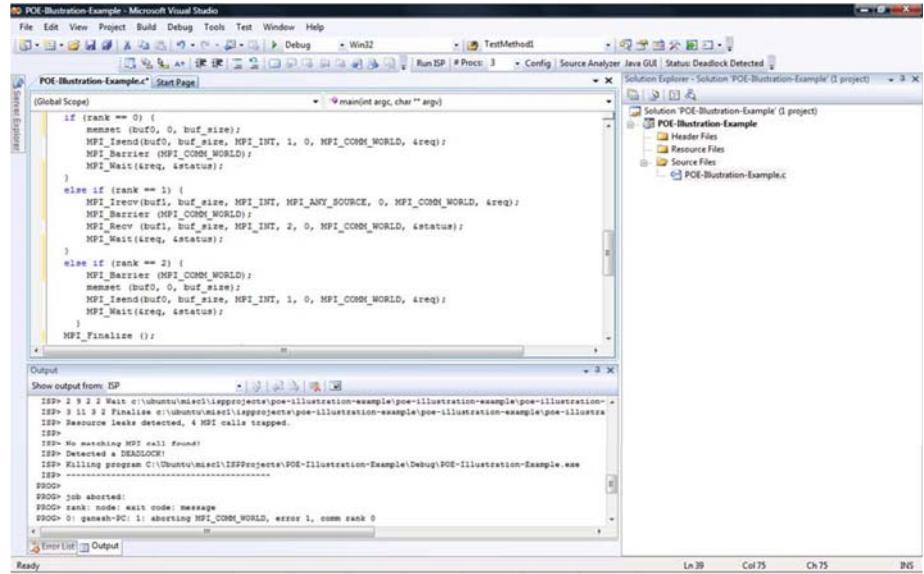


Fig. 1. Visual Studio Plug-in of ISP

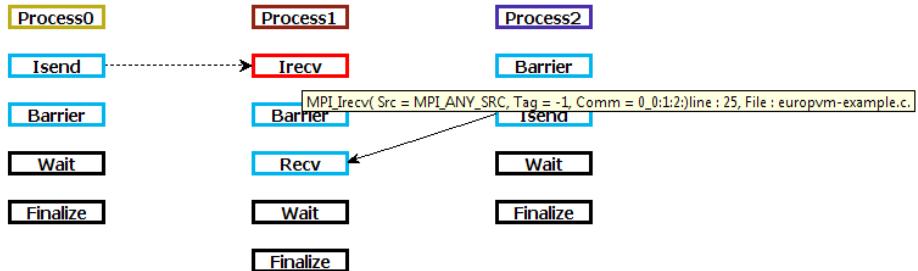


Fig. 2. Non-deadlocking Interleaving, Communication Matches, and Tool Tip

We highlight the MPI communication sources and targets and the communication handles. A user wanting to formally verify this program under ISP can launch the intuitive VS Plug-in of ISP, displaying the figure shown in Figure 1.

Next, the user runs ISP whose POE algorithm will recognize that the `Irecv` of P1 has two potential send matches. (Note: there are many VS GUI views in this sequel; we refrain from presenting them owing to the lack of clarity on paper.) It runs two interleavings, finding a deadlock in one interleaving. The user may then single-step the execution trace. When `Barrier` is encountered, the display adjusts itself, showing all processes involved in the barrier. Soon the user reaches the deadlock point, at which time the user invokes the transition navigator, obtaining a tree display of the transitions invoked. Those transitions that are not matched yet are highlighted in red color in this display. At this point, the user may cut into Visual Studio for more incisive debugging.

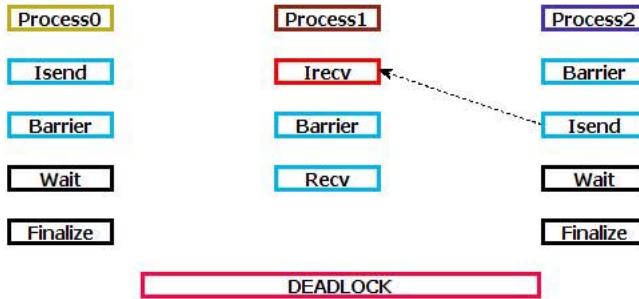


Fig. 3. Communication Match Resulting in Deadlock

Now, the user may be interested in finding the root cause of the bug. They display the communication matches which shows the two interleavings that this example has. The wildcard receive – source of non-determinism – is shown in red. The interleaving in which P0’s Isend matches this receive (Figure 2) does not lead to a deadlock. The tool tips show the details of each icon (MPI command, file/line). The interleaving in which P2’s Isend matches this receive (Figure 3) does lead to a deadlock.

The user further wants to drill down and locate the bug. They display the full completes-before graph with respect to a chosen group of nodes. Completes-before (CB) has two components: *intraCB* which shows the process-local part of CB, and *interCB* which shows how the CB graph builds across processes. They may merely display the *intra* completes-before relation (Figure 4) that shows the completes-before graph corresponding to the deadlocking interleaving. The completes-before graph shows that an **Isend** issued before an **MPI_Barrier** need not complete before the barrier; it can complete afterwards! This tells the user that both wildcard matches are possible. The CB graph also tells the user how different platforms may process the MPI commands out of order. They may choose to display both intraCB and interCB as in Figure 5. This tour should

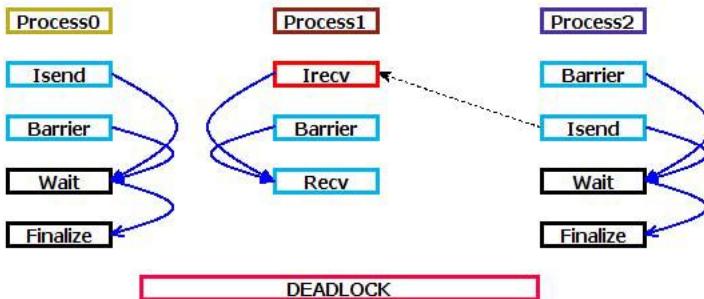


Fig. 4. IntraCB of Deadlocking Interleaving

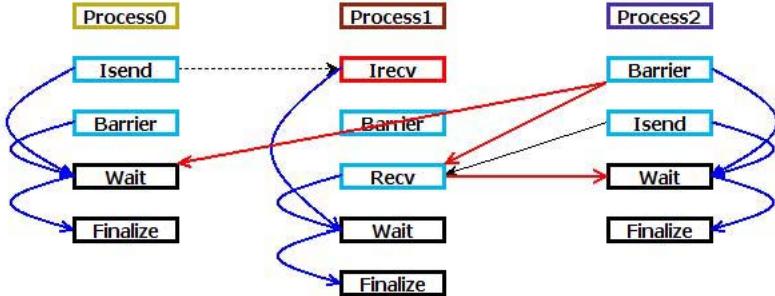


Fig. 5. Completes-Before and Communication Matches (Non-deadlocking)

tell the reader that ISP’s MPI-specific GUIs are able to say a lot within a small amount of space. We also wish to point out the conceptual depth of these displays that can help a beginner learn MPI programming reliably.

How Completes-Before (CB) is determined. The MPI standard requires non-overtaking in the sense that two MPI messages send from Pi and Pj arrive in that order (per tag and communicator). Thus, if there is a Send(to P1) followed by another Send(to P1) in an MPI program, these commands must *complete* in order (of course they are always issued in program order). On the other hand, if a Send(to P1) is followed by a Send(to P2) in an MPI process, these sends may finish out of order. Imagine the first one sending a large message while the second one sends a short message; in this case, it would be inefficient to wait for the first send to finish. It also is no violation of non-overtaking to allow the second send to finish. The full definition of the CB relation appears in [1]. Our formalization of CB allows us to achieve two objectives at once: (i) schedule actions within POE so that the maximal extent of wildcard receive non-determinism is discovered (the same idea is also followed for wildcard probes); (ii) it also displays to the MPI user how each platform may reorder the commands issued from *the very same* MPI process. Of course, the display of relevant interleavings by ISP avoids exploding the view with a display of interleaving variations of commands issued from *different* processes.

Probes De-Mystified. An adaptation of a Fortran example [19] concerning probes is as follows:

```
P0: Ssend(to 1); ...
P1: Probe(from *); Recv(from *); Recv(from *); ...
P2: Ssend(to 1); ...
```

In the above example, either P0’s Ssend(to 1) or P2’s Ssend(to 1) can enable P1’s Probe(from *). In a regular MPI platform, if P0’s Ssend(to 1) enables the P2’s Probe(from *), one can expect P0’s Ssend(to 1) to match the Recv command of P1 following the Probe. If this program is ported and run on a different platform, P2’s Ssend(to 1) may match the same Recv command. From [19], a designer will

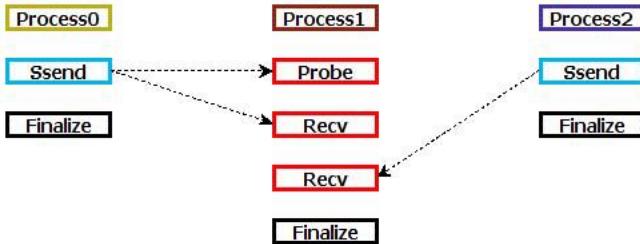


Fig. 6. Iprobe Matches of Interleaving 1 on Example from [19]

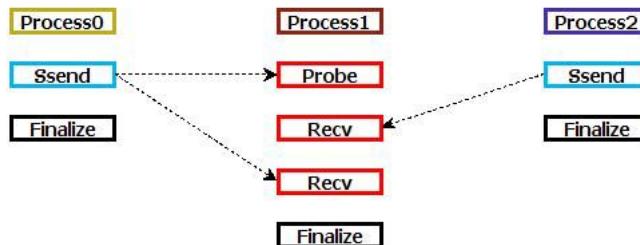


Fig. 7. Iprobe Match of Interleaving 2 on the Same Example as in Figure 6

likely recall that suppose a Probe matches a certain send, then the following Recv command *need not match the same send!* What if the expert forgets this fact and is baffled by the inundation of traces shown by a conventional debugger? ISP would, on the other hand produce a display shown in Figures 6 and 7. This display very clearly shows which send the probe matched, and which send that the following receive actually matched.

4 Conclusions

Formal verification tools that display concurrency concepts are useful for concurrent program understanding and incisive debugging. With the multi-core and peta-scale revolutions looming, such efforts are long overdue. Almost all concurrency debugging runs into exponential variations in executions. In addition to interleaving reduction through partial order reduction, a vast array of “exponential compression” methods await to be exploited sufficiently in the area of MPI programming: process symmetries and data-space symmetries are two examples. The current approach recommended with ISP (and most concurrency tools, for that matter) is to downscale a problem before such a tool is used. This is not sufficient for many bug classes, including resource bugs, and much work remains in this regard.

References

1. Vakkalanka, S., Gopalakrishnan, G.C., Kirby, R.M.: Dynamic verification of MPI programs with reductions in presence of split operations and relaxed orderings. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 66–79. Springer, Heidelberg (2008)
2. Vo, A., Vakkalanka, S., DeLisi, M., Gopalakrishnan, G., Kirby, R.M., Thakur, R.: Formal verification of practical MPI programs. In: Principles and Practices of Parallel Programming (PPoPP), pp. 261–269 (2009)
3. Sharma, S., Vakkalanka, S., Gopalakrishnan, G.C., Kirby, R.M., Thakur, R., Gropp, W.D.: A Formal Approach to Detect Functionally Irrelevant Barriers in MPI Programs. In: Lastovetsky, A., Kechadi, T., Dongarra, J. (eds.) EuroPVM/MPI 2008. LNCS, vol. 5205, pp. 265–273. Springer, Heidelberg (2008)
4. Vakkalanka, S., DeLisi, M., Gopalakrishnan, G.C., Kirby, R.M., Thakur, R., Gropp, W.D.: Implementing efficient dynamic formal verification methods for MPI programs. In: Lastovetsky, A., Kechadi, T., Dongarra, J. (eds.) EuroPVM/MPI 2008. LNCS, vol. 5205, pp. 248–256. Springer, Heidelberg (2008)
5. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (2000)
6. <http://www.eclipse.org/ptp>
7. Schaeli, B., Al-Shabibi, A., Hersch, R.D.: Visual debugging of MPI applications. In: Lastovetsky, A., Kechadi, T., Dongarra, J. (eds.) EuroPVM/MPI 2008. LNCS, vol. 5205, pp. 239–247. Springer, Heidelberg (2008)
8. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: PLDI, pp. 446–455 (2007)
9. <http://research.microsoft.com/CHESS>
10. Vuduc, R., Schulz, M., Quinlan, D., de Supinski, B., Saebjornsen, A.: Improved distributed memory applications testing by message perturbation. In: PADTAD (2006)
11. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: POPL 2005, pp. 110–121 (2005)
12. DeLisi, M.: http://www.cs.utah.edu/formal_verification/ISP_Tests
13. DeLisi, M.: http://www.cs.utah.edu/formal_verification/ISP-release
14. Siegel, S.F., Siegel, A.R.: MADRE: The Memory-Aware Data Redistribution Engine. In: Lastovetsky, A., Kechadi, T., Dongarra, J. (eds.) EuroPVM/MPI 2008. LNCS, vol. 5205, pp. 218–226. Springer, Heidelberg (2008)
15. <http://www.mpiblast.org>
16. Lusk, R., Pieper, S., Butler, R., Chan, A.: Asynchronous dynamic load balancing, <http://unedf.org/content/talks/Lusk-ADLB.pdf>
17. The IRS Benchmark Code, https://asc.llnl.gov/computing_resources/purple/archive/benchmarks/irs/
18. Pacheco, P.: Parallel Programming with MPI. Morgan Kaufmann, San Francisco (1997)
19. A note on the probe command, <http://www.mcs.anl.gov/research/projects/mpi/mpi-standard/mpi-report-1.1/node50.htm>

Sound and Efficient Dynamic Verification of MPI Programs with Probe Non-determinism*

Anh Vo¹, Sarvani Vakkalanka¹, Jason Williams¹, Ganesh Gopalakrishnan¹,
Robert M. Kirby¹, and Rajeev Thakur²

¹ School of Computing, Univ. of Utah, Salt Lake City, UT 84112, USA

² Math. and Comp. Sci. Div., Argonne Nat. Lab., Argonne, IL 60439, USA

Abstract. We consider the problem of verifying MPI programs that use `MPI_Probe` and `MPI_Iprobe`. Conventional testing tools, known to be inadequate in general, are even more so for testing MPI programs containing MPI probes. A few reasons are: (i) use of the `MPI_ANY_SOURCE` argument can make MPI probes non-deterministic, allowing them to match multiple senders, (ii) an `MPI_Recv` that follows an MPI probe need not match the `MPI_Send` that was successfully probed, and (iii) simply re-running the MPI program, even with schedule perturbations, is insufficient to bring out all behaviors of an MPI program using probes. We develop several key insights that help develop an elegant solution: prioritizing MPI processes during dynamic verification, handling non-determinism, and safe handling of probe loops. These solutions are incorporated into a new version of our dynamic verification tool ISP. ISP is now able to efficiently and soundly verify larger MPI examples, including MPI-BLAST and ADLB.

1 Introduction

The correctness of MPI programs is of paramount importance, especially considering the growing cost of conducting large-scale simulations, and the losses (opportunity costs and unreliable results) due to errant or crashing simulations. Conventional testing oriented approaches are woefully inadequate for finding bugs in MPI programs. To appreciate the magnitude of the problem, consider an MPI program with five processes where each process can make five MPI calls (called ‘*Generic MPI Example*’). Such a program has over 10 billion potential interleavings (schedules)! Unaware of which interleavings induce control flow variations down the execution path leading to bugs, testing tools often pursue a ‘best effort’ randomization of the interleavings. Unfortunately, given the exponentially growing execution space, these techniques do not attain adequate coverage of the *relevant interleavings* [1].

* Supported in part by Microsoft, NSF CNS-0509379, CCF-0811429, and the Mathematical, Information, and Computational Science Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

While static analysis and model-based verification (*e.g.*, modeling programs in dialects (*e.g.* [17]) of SPIN [13]) are two widely followed *formal* approaches, the former generate false alarms causing a designer to spend weeks confirming whether a reported bug is genuine [12], while the latter requires designers to re-express an MPI program in an alternate notation. Thus, both these approaches prove impractical for realistic MPI programs, especially when also faced with the need to hand-model the semantics of the MPI calls and the surrounding C code, repeating all this for *each* program tuning step. In contrast, our approach – dynamic formal verification – runs the program after replacing the native scheduler with a *formal* verification scheduler. Pioneered by Godefroid [11] and later improved in the dynamic partial order (DPOR) approach [10], dynamic verification enjoys a growing presence, in tools such as CHESS [2], MODIST [23], CREST [8], Bandera [9], Java PathFinder [3], and Inspect [24].

Our dynamic verification approach for MPI programs based on a customized DPOR algorithm called Partial Order avoiding Elusive interleavings (POE) [16,20,21,22] explores all relevant interleavings parsimoniously. Consider a special case of the *Generic MPI Example* with exactly one wild-card (`MPI_ANY_SOURCE`) receive “`Recv(from *)`” in process P0 which is matched by sends `Send1(to P0)` and `Send2(to P0)` issued by processes P1 and P2. Let all other MPI calls be point-to-point MPI calls. POE will explore just *two* interleavings, one where `Send1(to P0)` matches `Recv(from *)` and the other where `Send2(to P0)` matches `Recv(from *)`. because these cases can affect verification outcome by conveying different values to the wildcard receive. Moreover, POE *enforces* these matches by dynamically rewriting `Recv(from *)` into `Recv(from P1)` and `Recv(from P2)` over two successive replays of the program. Permuting the issue order of point-to-point calls is pointless for finding safety violations. ISP has been used to verify a number of real-world MPI programs (*e.g.*, ParMETIS [14], MADRE [18], and the Implicit Radiation Solver or IRS [7]) for the absence of deadlocks, resource leaks, communication races, and assertion violations [21,22]. ISP has been released [4] to run on Linux, Mac OS/X, and Windows, supporting the MPI libraries MPICH2, Open-MPI, and Microsoft MPI, supporting > 60 MPI 2.1 functions.

`MPI_Probe` detects the presence of a receivable message, while `MPI_Iprobe` call is its non-blocking counterpart. Here, we present the addition of `MPI_Probe` and `MPI_Iprobe` to ISP, so far deliberately postponed due to their highly non-deterministic behavior. Several key insights finally helped us design a sound and efficient algorithm, now allowing us to verify two large examples – MPI-BLAST and ADLB – that make heavy use of `Probes`.

Related Work: Pioneering work on formal methods for MPI began with MPI-SPIN [17], a SPIN-based model checker that can be used to verify MPI programs for deadlocks and safety errors. Its need for hand-built verification models was already mentioned. ISP is believed to be the only dynamic formal verification tool for MPI. Non-MPI dynamic formal verification tools (*e.g.*, [2,23]) have no awareness of the MPI semantics – hence inapplicable for MPI.

2 Dynamic Interleaving Reduction with MPI Probes

The reason why handling `Probe` and `Iprobe` is an involved process is captured by the following three examples. (*Note:* Unless otherwise indicated, we assume that `Iprobe` is called in a polling-loop until its flag is set to true – the most common usage of `Iprobes`. We also represent `MPI_ANY_SOURCE` by a *. For simplicity, we will ignore the data payloads, tags, and statuses of these messages and only focus on the destination/source of the sends and receives. *End Note:*)

Example-1:

```
P0: Isend (to P1); Ssend(to P2); Wait ();
P1: Iprobe (from *, &status); Recv (from status.MPI_SOURCE), Recv (from *)
P2: Recv(from P0), Ssend (to P1);
```

If this example is naively run on a regular MPI platform, one may find that P0’s `Isend`(to P1) enables P1’s `Iprobe` to exit its loop. This causes P1’s receive to actually receive P0’s `Isend`(to P1) itself (the message that was probed). Thereafter, P0’s `Ssend` will match P2’s `Recv`, and then P2’s `Ssend` will match P1’s `Recv`. If this program is ported and run on another machine, one may find a different outcome: (i) while P0’s `Isend` is active, P0’s `Ssend` can post. (ii) this causes P2’s `Recv` to fire, followed by P2’s `Ssend`. (iii) Now we can have both P0’s `Isend` and P2’s `Ssend` become candidates for P1’s `Iprobe` match (because this is a wildcard `Iprobe`, where * shows `MPI_ANY_SOURCE`).

Formal tools such as ISP must discover the most general set of such executions possible on any platform and find bugs in them. It must also discover these behaviors by parsimoniously exploring interleavings. Our new algorithms are designed with these goals in mind; in particular, the insight that allows us to handle Example-1 is as follows:

Recognize the priority level at which a dynamic verifier must fire probes. In other words, by delaying the wildcard `Iprobe`, we can allow both match possibilities the chance of being discovered. This approach is, fortunately, a fundamental part of POE in handling wildcard receives [21]. Our new algorithm now simply incorporates this approach for wildcard probes also. Also when multiple probe matches are found, our new algorithm also performs dynamic rewriting, as was done in ISP for wildcard receives, as explained in our Generic MPI Example.

An important issue that arises when dealing with `Iprobe` is that it is not *a priori* known how many times `MPI_Iprobe` will be called from within the probe loop until it returns true, even after a send operation that directs the probing

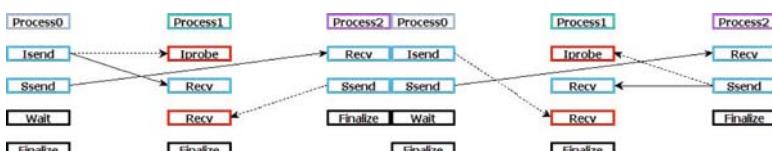


Fig. 1. Different Possible Matching of Iprobes

process has been launched. To overcome this non-deterministic behavior, our scheduler employs the following approach:

When eligible sends are found for Iprobes, the scheduler of ISP must busy-wait on the Iprobes until they return true. Since matching sends are already posted, the busy-wait loops are guaranteed to terminate. However, by doing this, we do not consider the case where matching sends have been initiated, but the `Iprobe` call can still non-deterministically return false. For our verification purposes, we assume that the false code path contains only harmless statements. A complete verification solution might require other analysis methods such as static analysis, for which we are also actively exploring.

There are other subtleties associated with probes; they are illustrated through the next two examples:

Example-2:

```
P0: Isend(to P1); Ssend(to P2); Wait();
P1: Iprobe(from *, &status); Ssend(to P2);
     Iprobe(from *, &status); Recv (status.MPI_SOURCE); Recv(from *);
P2: Recv(from P0); Recv (from P1); Ssend(to P1);
```

Despite similarities with Example-1, this example contains two `Iprobes` that look very much the same, yet have very different possible matching sends. Here, the first `Iprobe` in P1, having to complete before the `Ssend (to P2)` in P1, has only one possible matching send (namely `Isend(to P1)` issued by P0). After the `Ssend(to P2)` of P1 is matched with P2's `Recv(from P1)`, the situation is now played out similarly to the previous example. This allows the next `Iprobe` to see two possible matching sends, namely P0's `Isend(to P1)` and P2's `Ssend(to P1)`. ISP has to consider both these matches in two different interleavings.

The high level observations from this example are the following.
Recognize the local completion semantics of Iprobes. Iprobe calls have to be completed before any other following MPI calls in the same process. The delayed execution of ISP has to take this into account.

Example-3:

```
P0: Recv(from P1); Send (to P1);
P1: Iprobe(from P0, &flag, &status);
     if (flag == false) Send(to P0) else Recv (from P0);
```

From this example, it is straightforward to see that busy waiting on `Iprobe` inside the scheduler is not always safe (the singly threaded scheduler can go into an infinite loop, as no send has been posted). Busy-waiting is safe only when matching sends for `Iprobe` are already determined. This example describes the simplification of a pattern that is found repeatedly in many large MPI programs.

Our new algorithm handles such situations as follows: The `Iprobe` is delayed. *Recognize that busy-waiting for IProbes to return is not always safe. The Iprobe calls should be allowed to return, with the flag set to false, when the execution cannot progress anymore and a matching send has not yet been found.*

3 ISP Overview

At a high level, ISP works by intercepting the MPI calls made by the target program and making decisions on when to send these MPI calls to the MPI library. This is accomplished by the two main components of ISP: the Profiler and the Scheduler.

The Interposition Layer: The interception of MPI calls is accomplished by compiling the ISP interposition layer together with the target program source code. The interposition layer makes use of the profiling mechanism PMPI. It provides its own version of MPI_f for each corresponding MPI function f . Within each of these MPI_f , the profiler communicates with the scheduler using TCP sockets¹ to send information about the MPI call that the process wishes to make. It will then wait for the scheduler to make a decision on whether to send the MPI call to the MPI library or to postpone it until later. When permission to fire f is granted from the scheduler, the corresponding $PMPI_f$ will be issued to the MPI run-time. Since all MPI libraries come with functions such as $PMPI_f$ for every MPI function f , this approach provides a portable and light-weight instrumentation mechanism for MPI programs being verified.

The ISP Scheduler: The scheduler is where our main scheduling algorithm, namely POE (Partial Order avoiding Elusive interleavings) is carried out. The scheduler meets the following objectives: G1: discovers the maximal set of sends that can match a wildcard receive (viewed across all MPI-standard compliant MPI libraries); G2: accurately models the semantics of the global operations (such as barriers) of MPI. In MPI, not all MPI operations issued by a process complete in that order, and a proper modeling of this out-of-order completion semantics is essential in order to meet goals G1 and G2. For example, two `MPI_Isend` commands issued in succession by an MPI process P1 to the same target process (say P2) are forced to match in order, whereas if these `MPI_Isends` are targeted to two *different* MPI processes, then they *may match contrary to the issue order*. As another example, any operation following an `MPI_BARRIER` must complete only after the barrier has completed, while an operation issued *before* the barrier may linger across the barrier, and actually complete *after* the barrier!

Section 3 provides an ISP overview and summarizes the POE algorithm. Section 4 begins an elaboration of our main contributions, namely support for Probes and Iprobes.

Main Steps of the POE Algorithm: The POE algorithm works as follows. There are two classes of statements to be executed: (i) those statements of the embedding programming language (C, C++, and Fortran) that do not invoke MPI commands and (ii) the MPI function calls. The embedding statements in an MPI process are local in the sense that they have no interactions with those of another process. Hence, under POE, they are executed in program order. When

¹ When running within a local machine, ISP uses unix sockets to reduce communication overhead.

an MPI call f is encountered, the scheduler records it in its state; however, it does not (necessarily) issue this call into the MPI run-time. (Note: When we say that the scheduler issues/executes MPI call f , we mean that the scheduler grants permission to the process to issue the corresponding PMPI- f call to the MPI run-time). This process continues until the scheduler arrives at a *fence*, where a fence is defined as an MPI operation that cannot complete after any other MPI operation following it. Note that both MPI_Probe and MPI_Iprobe are considered fences. The list of such fences include MPI_Wait, MPI_Barrier, etc., and are formally defined in [21]. When all MPI processes are at their individual fences, the full extent of all senders that can match a wildcard receive becomes known, and dynamic rewriting can be performed with respect to these senders. The collection of sends and matching receives can then be issued. For details, please see [21].

Completes-Before Ordering: The Completes-Before (CB) ordering accurately captures when two MPI operations x and y issued from the same process in program order are guaranteed to complete in that order. For example, if an MPI process P1 issues an MPI_Isend that ships a large message to P2 and then issues MPI_Isend that ships a small message to P3, it is possible for the second MPI_Isend to complete first. A summary of the completes-before order of MPI is as follows: (i) **Send Order:** Two Isends sending data to the same destination complete in issue order. (ii) **Receive Order:** Two Irecvcs receiving data from the same source complete in issue order. (iii) **Wildcard Receive Order:** If a wildcard Irecv is followed by another Irecv (wildcard or not), the issue order is respected by the completion order. (iv) **Wait Order:** A Wait and another MPI operation following it complete in issue order. For a formal description of the CB relation, please see [21].

4 Implementation of Probe and Iprobe

Early issue of sends: As mentioned before, ISP does not allow the processes to issue MPI calls into the MPI runtime system until all processes reach a fence point (otherwise ISP cannot control from which send a wildcard receive chooses to receive). Consequently, all receives as well as non-blocking sends are delayed. *Note: MPI_Sends* are also treated as non-blocking, because they are usually buffered if the message size does not exceed the eager send limit *End Note:*. This *delayed issue* approach poses a major obstacle for implementing Probes and Iprobes: the returned status of a probe call cannot be determined. This problem can be solved by one of these two approaches:

- Having ISP manually manipulate the returned status, which would also require the trapping of MPI_Get_count.
- Issue the sends after the scheduler finishes collecting the envelope of the sends. This *early issue* approach for sends is considered safe because, unlike receives which can have non-deterministic behavior under the presence of wildcards, sends are always deterministic. This allows the MPI library to automatically take care of writing the returned status.

```

1: if (rank == 0) {
2:   MPI_Isend(to P1,&req); MPI_Barrier();
3:   MPI_Ssend(to P2); MPI_Wait(&req);}
4: else if (rank == 1) {
5:   MPI_Barrier(); while (!flag) MPI_Iprobe(from *, &flag, &status);
6:   MPI_Recv(from status.MPI_SOURCE); MPI_Recv(from *);}
7: else if (rank == 2) {
8:   MPI_Irecv(from 0,&req); MPI_Barrier();
8:   MPI_Wait(&req); MPI_Ssend(to P1);}

```

Fig. 2. Ordering Semantics and Operation Lifetimes

Although both methods are technically sound, we opted to go with the latter approach since trapping more MPI calls incurs higher overhead for the scheduler. However, the chosen approach also poses several difficulties, one of which can be described by the following example: In the following code, assume that `data` is large enough to exceed the eager send limit:

```

P0: Send (data, to P1); Probe(from P1, &status); Recv (from P1);
P1: Send (data, to P0); Probe(from P0, &status); Recv (from P0);

```

When the verification is run under no buffering mode², the program is obviously deadlocked, and ISP detects the deadlock easily since the program is instructed to execute all `MPI_Sends` as `MPI_Ssends`. The situation gets complicated when the program is verified in buffering mode, under which ISP will consider all `MPI_Sends` as non-blocking. However, because the size of data exceeds the eager send limit of the MPI library, the early issued `MPI_Sends` block within the processes without the scheduler being aware of its states. In order to address this problem, *all MPI_Sends are converted to MPI_Isend following by an MPI_Wait*. The `MPI_Sends` are issued early as `MPI_Isends` and they are completed by `MPI_Waits` when the scheduler finds a matching receive.

Basic POE algorithm with Probe: Consider Figure 2, which is a slightly modified version of Example-1 mentioned earlier. We will use this example to illustrate the basic steps of POE, with the *italicized text* indicating new changes (*i.e.* the new contributions of this work) made to POE to handle `Probes` and `Iprobes`.

- Collect `Isend` of line 2, *let the process issue it. Mark it as “issued”, but still not “matched”*
- Collect `Barrier` (line 2), and do not issue.
- `Barrier` is a fence, stop collecting from rank 0; switch to rank 1.
- Collect `Barrier` (line 4), and do not issue; switch to rank 2.
- Collect `Irecv` (line 7), and do not issue. Then collect `Barrier` (line 7), and do not issue.

² ISP can verify MPI programs under full buffering or no buffering. With the full buffering mode, all `MPI_Sends` are buffered. With the no buffering mode, all `MPI_Sends` are treated as `MPI_Ssends`.

- A fence has been reached in every rank. Now, form a match set in priority order, with the following priority order followed: barriers first, then non wildcard sends/receives/*probes*, and finally wildcard sends/receives/*probes*.
- In our current state, there is indeed a highest-priority match set formed by the barriers. Now, POE sends these **Barriers** into the MPI runtime through **PMPI_Barrier** calls.
- The next ordering points (fences) are attained at **Ssend** (line 3), **Iprobe** (line 4), **Wait** (line 8). Note that the **Ssend** is already issued by the process due to our *early issue of send* implementation (Apparently that process is now blocking).
- The non-wildcard match-set of **Ssend** (line 3) and **Irecv** (line 7) are both issued.
- Now we have reached the ordering point of **Wait** (line 3), **Iprobe** (line 4), and **Ssend** (line 8). No other higher priority match sets exists, we can now find out all the potential senders that can match the **Iprobe**.
- Dynamically rewrite **Iprobe(*)** into **Irecv(P0)** and **Iprobe(P2)**, in two different executions.
- Form the first match set of **Iprobe(from P0)** and **Isend(to P1)** of line 2. *Mark them both as matched. Only issue the Iprobe because the Isend was already issued earlier. In addition, the scheduler needs to ensure that it should not collect anymore calls from P0 until a receive is posted for the Isend of line 2. i.e., the fence point for P0 is still in place.* Since a matching send is already issued earlier, P0 will *busy-wait* on the **Iprobe** of line 8. Thus, the program loop should execute only once!
- Form the second match set of **Iprobe(from P2)** and **Ssend (to P1)** of line 8. Pursue this interleaving though re-execution of the MPI program.

Soundness: Verification under ISP without probes is argued sound in [21]. The addition of probes preserves all our earlier assumptions. The engineering of efficient support for probes is our main contribution here; the elementary semantics of our engineering are similar to how wildcard receives were handled earlier, and hence are sound.

5 Experimental Results

We have tested our new algorithm against all the aforementioned examples (the original algorithm was tested against various benchmarks and in many cases found deadlocks missed by conventional tools [1]). In all our tests, ISP verifies the program with the *correct* number of interleavings, *i.e.*, all the interleavings explored by ISP are relevant. We also verified the subtle example found in the MPI Standard (Example 3.18 [6]), where a **Recv** might end up not receiving what the process probed (because the **Recv** was used to receive from **MPI_ANY_SOURCE**, not the source returned by **Probe** through the status).

Consider Figure 3. Note that because the **Recv** of lines 9 and 11 are called as wildcard receives and do not use the status returned by **Probe**, the program becomes incorrect. ISP correctly sees that there are four possible matchings (two for the first Probe, two for the first Recv). Here is a summary of what happens:

```

1: if (rank == 0)
2:   MPI_Send (data1, MPI_INT, to P2);
3: else if (rank == 1)
4:   MPI_Send (data2, MPI_DOUBLE, to P2);
5: else if (rank == 2) {
6:   for (i = 0; i < 2; i++) {
7:     MPI_Probe(from *, &status);
8:     if (status.MPI_SOURCE == 0)
9:       MPI_Recv(data1, MPI_INT, from *);
10:    else
11:      MPI_Recv(data2, MPI_DOUBLE, from *);}}
```

Fig. 3. Pseudocode of pattern found in Example 3.18 in MPI 1.1 Standard

- All fence points reached at `Send` of line 2, line 4, and `Probe` of line 7.
- Two possible matchings for `Probe(*)`, one with `Send` of line 2, the other with `Send` of line 4.
- Pursue the 1st interleaving: rewrite `Probe(*)` into `Probe(P0)`.
- P2 reaches fence point again at `Recv(*)` of line 9. Both P0 and P1 still at previous fence points.
- Two possible matchings for `Recv(*)`, one with `Send` of line 2, the other with `Send` of line 4.
- Pursue the 1st interleaving: rewrite `Recv(*)` into `Recv(P0)`
- No extra interleaving is required for the `Probe` and `Recv` of the second iteration of the loop (only one matching send is possible).
- When the execution finishes, pursue second interleaving: rewrite `Probe(*)` of line 7 into `Probe(P1)`, `Recv(*)` of line 9 into `Recv(P0)`.
- Third interleaving: rewrite `Probe(*)` of line 7 into `Probe(P0)`, the `Recv(*)` of line 9 into `Recv(P1)`.
- Fourth interleaving: rewrite `Probe(*)` of line 7 into `Probe(P1)`, the `Recv(*)` of line 9 into `Recv(P1)`.

We have also tested ISP with MPI-Blast[5] and ADLB[15], two large and non-trivial MPI programs (about 70K lines of code for MPI-Blast and 8K lines of code for ADLB). Both of which have extensive usage of wildcard `Probe` and `Iprobe`. In both cases, the new algorithm successfully verified both programs for freedom from deadlocks and assertion violations. The verification was done for a small number of processes. We also observed that the number of interleavings grew very fast for high number of processes due to the high numbers of wildcard probes.

6 Conclusions

We described our new algorithm to formally verify MPI programs for deadlocks, resource leaks, and assertion violations under the presence of `MPI_Probe` and `MPI_Iprobe` calls. The algorithm is implemented in our tool ISP. To the best of our knowledge, ISP is the *only* dynamic verification tool for MPI that verifies

large and non-trivial MPI programs. The new algorithmic enhancements enable ISP to handle a wider range of MPI applications.

Earlier, we mentioned that one drawback of the current implementation of **Probe** is the large number of interleavings generated when the number of wild-card calls is high. This has recently been addressed by more intelligently handling MPI dependencies (future publication). We are also exploring other techniques that downscale an MPI program while preserving the targeted bugs.

References

1. http://www.cs.utah.edu/formal_verification/ISP_Tests/
2. <http://research.microsoft.com/en-us/projects/chess/>
3. <http://javapathfinder.sourceforge.net/>
4. http://www.cs.utah.edu/formal_verification/ISP-release/
5. <http://www.mpiblast.org>
6. Mpi standard 1.1., <http://www.mpi-forum.org/docs/mpi-11.ps>
7. The IRS Benchmark Code, https://asc.llnl.gov/computing_resources/purple/archive/benchmarks/irs/
8. Burnim, J., Sen, K.: Heuristics for scalable dynamic test generation. Technical Report UCB/EECS-2008-123, Univ. of California, Berkeley (September 2008)
9. Dwyer, M., Hatcliff, J., Schmidt, D.: Bandera: Tools for automated reasoning about software system behavior. In: ERCIM News, 36 (January 1999)
10. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: POPL 2005 (2005)
11. Godefroid, P., Hanmer, B., Jagadeesan, L.: Systematic software testing using VeriSoft: An analysis of the 4ess heart-beat monitor. Bell Labs Technical Journal, 3(2), April-June (1998)
12. Godefroid, P., Nagappan, N.: Concurrency at microsoft - an exploratory survey, EC2 (2008)
13. Holzmann, G.J.: The model checker spin. IEEE Transactions on Software Engineering 23(5), 279–295 (1997)
14. Karypis, G.: METIS and ParMETIS, <http://glaros.dtc.umn.edu/gkhome/views/metis>
15. Lusk, R., Pieper, S., Butler, R., Chan, A.: Asynchronous dynamic load balancing, <http://unedf.org/content/talks/Lusk-ADLB.pdf>
16. Sharma, S., Vakkalanka, S., Gopalakrishnan, G.C., Kirby, R.M., Thakur, R., Gropp, W.D.: A formal approach to detect functionally irrelevant barriers in MPI programs. In: Lastovetsky, A., Kechadi, T., Dongarra, J. (eds.) EuroPVM/MPI 2008. LNCS, vol. 5205, pp. 265–273. Springer, Heidelberg (2008)
17. Siegel, S.F., Avrunin, G.S.: Verification of MPI-based software for scientific computation. In: Graf, S., Mounier, L. (eds.) SPIN 2004. LNCS, vol. 2989, pp. 286–303. Springer, Heidelberg (2004)
18. Siegel, S.F., Siegel, A.R.: MADRE: The Memory-Aware Data Redistribution Engine. In: Lastovetsky, A., Kechadi, T., Dongarra, J. (eds.) EuroPVM/MPI 2008. LNCS, vol. 5205, pp. 218–226. Springer, Heidelberg (2008)
19. Vakkalanka, S., DeLisi, M., Gopalakrishnan, G., Kirby, R.M.: Scheduling considerations for building dynamic verification tools for MPI. In: PADTAD-VI 2008 (2008)

20. Vakkalanka, S., DeLisi, M., Gopalakrishnan, G.C., Kirby, R.M., Thakur, R., Gropp, W.D.: Implementing efficient dynamic formal verification methods for MPI programs. In: Lastovetsky, A., Kechadi, T., Dongarra, J. (eds.) EuroPVM/MPI 2008. LNCS, vol. 5205, pp. 248–256. Springer, Heidelberg (2008)
21. Vakkalanka, S., Gopalakrishnan, G., Kirby, R.M.: Dynamic verification of MPI programs with reductions in presence of split operations and relaxed orderings. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 66–79. Springer, Heidelberg (2008)
22. Vo, A., Vakkalanka, S., DeLisi, M., Gopalakrishnan, G., Kirby, R.M., Thakur, R.: Formal verification of practical mpi programs. In: PPoPP 2009, pp. 261–269 (2009)
23. Yang, J., et al.: MODIST: Transparent Model Checking of Unmodified Distributed System. In: NSDI 2009 (to appear)
24. Yang, Y., Chen, X., Gopalakrishnan, G., Kirby, R.M.: Efficient stateful dynamic partial order reduction. In: Havelund, K., Majumdar, R., Palsberg, J. (eds.) SPIN 2008. LNCS, vol. 5156, pp. 288–305. Springer, Heidelberg (2008)

Bringing Reverse Debugging to HPC

Chris Gottbrath

TotalView Technologies, 24 Prime Park Way, Natick, MA 01760

`Chris.Gottbrath@totalviewtech.com`

`http://www.totalviewtech.com`

Abstract. Reverse debugging is a technique for troubleshooting and analyzing software that allows developers to work directly from a software failure to the source code error that led to that failure. ReplayEngine makes this technique available for High Performance Computing (HPC) environments. This paper presents an exploration of the challenges we face and solutions that we are exploring as we develop ReplayEngine into a mature HPC reverse debugging solution.

Keywords: troubleshooting, record and replay debugging, MPI debugging.

1 Introduction

1.1 Reverse Debugging and ReplayEngine

Reverse debugging is a technique for troubleshooting program analysis that involves examining the execution of a program “backward in time” using an interactive debugger. This is desirable because often one sees something and wants to answer the simple question “how did this happen?” Since many fundamental operations in typical imperative languages destroy information about the previous program state it is rarely possible to start with the end state and derive an earlier state. The ReplayEngine analysis tool discussed in this paper records the execution of the running program in order to present users with the ability to examine the states that come later first, and then work their way backward toward the beginning of execution.

Having recorded information about the program execution, ReplayEngine allows the user to examine earlier and later states within this history by deterministically replaying the program being debugged. This replay occurs within a sandbox that prevents repeated operations within the process from having an impact on other processes running on the system. The fact that is replay occurs deterministically is critical because software is often written to operate in a highly dynamic way, reading input from external sources and simultaneously executing along multiple threads of execution. With multiple threads there are many possible trajectories along which a specific bit of code might evolve. During program analysis with ReplayEngine exactly one such trajectory is recorded and made available for examination.

Why Reverse Debugging?. Much of the frustration and complexity of debugging with current tools comes from the fact that programmers have to work in a very indirect way to make the connection between the result of a bug and the cause of that bug. This observation motivates TotalView Technologies' interest in reverse debugging. Our tool lets developers analyze their program in the same direction as their troubleshooting thought process – simplifying and streamlining the whole process.

Getting the program to crash or otherwise misbehave is only the beginning of troubleshooting. In non-trivial bugs a gulf exists between the point in the program where it does something obviously wrong (the point of failure) and the earlier point where the error occurred (the site of the bug). The programmer examines the state of the program after the failure looking for clues to help generate a troubleshooting hypothesis about the error.

In the absence of reverse debugging the process involves restarting the program from scratch. The developer is then looking at a fresh run of the program, before the error. The challenge is to run the program to a point that can help the developer evaluate their hypothesis. Even with a surplus of patience and care the developer may overshoot the goal; even one step too far can destroy information and require the process to be restarted from the beginning. Factors such as sensitivity to input, organizational challenges like having to wait in a queue for access to a machine, or stochastic bugs that don't reproduce every time that the developer re-runs the code only make the traditional debugging process more daunting.

Opening up the potential to go directly backward radically simplifies the task at hand. It allows the developer to focus more clearly on the problem and reduces the amount of effort and time that will typically be required to complete the analysis of bugs.

1.2 Is the Approach Applicable to HPC?

Reverse debugging is of particular applicability to HPC for a number of reasons. First, HPC systems are often highly oversubscribed and access to them is often carefully managed. Time on HPC hardware is a valuable resource and whenever developers want such time there may be a wait of hours or days. This makes the “restart cycle” debugging techniques reviewed in the previous section, tedious enough on a desktop or server, heroic on a large scale HPC system. Second, HPC applications are essentially distributed, and sometimes also multithreaded, concurrent applications. In these applications, stochastic hard-to-reproduce defects are more frequent than in serial codes. Finally, HPC applications are typically developed by collaborations of developers with different skill sets. Troubleshooting an HPC bug may involve following clues through code written by collaborators. Learning someone else's code on the fly is all the more reason for a debugging technique that is as straightforward and bullet-proof as possible.

Since fault tolerant MPI provides a framework that can be used to manually replay sections of an MPI job. A number of users and proponents of this technology have proposed its use for reverse debugging in MPI contexts. [1,2]

1.3 ReplayEngine Status

ReplayEngine, a commercial software development product, is an add-on to the TotalView debugger.[3] It is available for 32- and 64-bit x86-based systems running Linux, and supports reverse debugging applications written in C, C++, Fortran 77, Fortran 90 and UPC.

Version 1.0 was released in late 2008 and updated in early 2009 to include support for parallel applications that use specific configurations of popular MPI libraries including MPICH2[4], Open MPI[5], MVAPICH2, Intel MPI, HP-MPI and LAM. ReplayEngine version 2.0 is under development as this paper is being written and planned for release during 2009. ReplayEngine 2.0 is expected to include the features discussed below and will support analyzing parallel programs in a wide range of HPC environments.

This paper explores the challenges that we face bringing reverse debugging to HPC. Section Two will review the reverse debugging technology as it is currently available; Section Three highlights four major technical challenges that we face in applying that technology in the HPC context; Section Four discusses the ways in which we have overcome these challenges; Section Five summarizes this material and looks to the future.

2 Recording and Deterministic Replay Technology

2.1 ReplayEngine Architecture

The reverse debugging capability provided by ReplayEngine relies on the ability to record and deterministically replay the execution of each individual process in the program. The debugger arranges for each target process to load an instrumentation library that manages the process of recording and replaying execution history. For the most part, the upper layers of the debugger operate on each process as they would otherwise do to suspend execution, read and write registers and memory, set breakpoints, and resume the process or individual threads. In addition to these operations, the add-on module gives the debugger the ability to instruct the process to go to any previous point in execution history. The instrumentation library transparently takes any action necessary to put the process into the right state so that the debugger can then query registers and memory from that point in recorded history.

Recording Program Execution. One of the challenges tackled by the instrumentation library is recording enough of the process's execution to allow the process to be re-executed along exactly the same trajectory over and over again. This includes inputs such as data read from external files, data received over the network, and the return value of system calls. Care is taken to ensure that this recording is done efficiently, both in terms of the time that it takes to record the information and in terms of time and the usage of memory for data storage.

The instrumentation library does its work by progressively rewriting the application as it runs. As a new function is encountered it is instrumented, this

prevents the user from having to pay the cost, in terms of time and space, for the instrumentation. It also ensures that instrumentation effort is only applied to parts of the program that are actually executed during the troubleshooting session.

Deterministic Replay. Given a recording of program execution, the instrumentation library can reproduce any point on the trajectory of the process by starting from a previous known state and replaying the program forward in a controlled fashion. During this replay the simulated process reads data from the execution history instead of any external source and finds that system calls return the predetermined values that were recorded during the execution of the live process. The instrumentation manages a collection of recorded states in such a way that the most frequent backward-stepping operations can be done quickly. However, there may also be points in execution history that will take more than a few seconds to reach – especially in very long recordings.

Because history is immutable some of the operations that TotalView users are accustomed to using are not available during replay (e.g. setting program variables, "set PC," and full asynchronous thread control).

2.2 ReplayEngine Usage

Backward Stepping. The simplest, easiest to understand, and most frequently used mechanism for inspecting the historical state of the program is to step backward line-by-line through the program. This is an analog of forward stepping and the same kind of rules apply. If a developer is stepping through a loop construct, back stepping from the top line will take them to the conditional, then back into the bottom of the loop.

As with forward stepping, there are several slight variations on the stepping theme for backward stepping. The "back next" operation takes the target program backward over a single executable line of code at the same scope level, even if it contains one or more function calls. The "back step" operation takes the target program backward over a line of code – unless that line of code contains one or more functions, in which case it takes the target program to the most recently executed line of code even if that line occurs in one of the called subroutines. In either of the previous cases, backward stepping from the first line in a function will take the program to the point in time before the function was called. When anywhere in a function, "back out" will take the program to the calling function just before the current function was called.

Together, these operations give the developer easy access to the recent past of the program's execution. They can be coupled with forward-stepping commands – so the developer can step back and forth over a single section of code as many times as necessary to understand it. This makes unfamiliar programs much easier to work with since any nuances of the program's behavior that are missed at first can be picked up on further examination.

Reverse "Run to". While developers do often step line-by-line through interesting sections of programs, there are other sections of the program that are

simply not that interesting for a given problem, so developers need a way to skip back over longer stretches of the execution history. This is provided by the “run back to” operation that lets the developer examine the state of the program at the most recent time the program was at any given line. The selected line might simply be a line in the same function, a line in a calling function, a line in any other function that happened to have been called previously in the program, or a line in main. This is a direct analog of the TotalView “run to” command, which works the same way but in the forward direction.

This ability is particularly effective when the function being examined contains one or more loop constructs, and rather than stepping around the loop many times, developers can simply jump to the line before the loop. Another use is to return to an earlier program phase by selecting one of the lines that represent this phase in the top level program structure.

Random Access. In some cases there is no specific line that defines the point in the execution history that the developer wishes to examine. ReplayEngine gives the user the ability to specify a point at which to inspect the program at that point. This is analogous to simply continuing then halting a program after waiting for a designated time. The ReplayEngine mechanism records a time-like metric .. which increases as the program runs but doesn’t exactly correspond to any external measurement of time. It is internally consistent, so a user can record this metric when the program is doing something “interesting” and use it to return directly to that point in the execution history later on during the troubleshooting process.

One simple use of this capability is to leap back to a point early in the execution history. This is useful if the programmer wants to see what the uninitialized value of a given memory location is. This feature can also be used to perform binary searches of execution history for poorly localized events.

If analysis of the later program execution history shows an unexpected data value in a variable that should not change frequently it is often useful to jump “a good distance” back and verify that this variable once held a sensible value. Then a watchpoint can be used, in forward re-execution, to locate the exact point when that variable was corrupted.

2.3 ReplayEngine User Interfaces

TotalView provides both a Graphical User Interface (GUI) and a Command Line Interface (CLI). Most features in TotalView can be driven using either interface. As an add-on ReplayEngine behaviors are accessible via either interface.

Graphical User Interface. To get started with reverse debugging on a specific process, developers enable it by selecting a check-box in the New Program dialog. No further user action is required – the debugger takes care of starting the program in such a way that the instrumentation library is loaded.

The existing TotalView process window contains a series of prominently placed buttons for the stepping commands. The new backward-stepping commands have

been added next to the forward-stepping commands. The fact that the backward-stepping commands are such close analogs of the forward-stepping commands should make these operations very easy to understand. The “run back to” command, like the forward debugging “run to” command requires the user to select a line in the program before it makes itself available.

Whenever the program is moving forward and actually recording new history the first time any reverse command is used the debugger will pause the target and seamlessly transition to Replay mode. In this mode the debugger is showing the user the program’s historical state. GUI feedback distinguishes between Replay and Record modes. The most noticeable feedback is a color change of the current line, which in TotalView is highlighted. The transition between inspecting historical state and recording new history is also seamless. If the user is reviewing execution history and uses any forward command that asks the program to go to a point that has not yet been recorded the debugger will transition back to Record mode as it reaches the end of the recorded history. There is also an explicit Live button which exits execution history and returns the user’s perspective to the live process.

Command Line Interface. The Command Line Interface (CLI) for TotalView is less frequently used for day-to-day interactive debugging, and is best used when developers want to script, extend or programmatically drive the debugger through a precise series of operations. Existing commands such as `dstep` have been extended to have a new `-back` flag that allows them to be used in the reverse direction.

The direct access to history, described above, is only available in the CLI. In that context a query command `dhistory` will allow developers to see the numeric measure of the program’s location in the process execution history. A `dhistory -go_time` command flag allows developers to randomly access the execution history.

3 Challenges Introduced by HPC Cluster Parallelism

When we think about bringing record and replay debugging technology to HPC environments there are a number of challenges that we need to address.

3.1 Cluster Architecture

HPC clusters often contain a very large number of nodes, each one a separate instance of an operating system, connected by a high speed network and often one or more secondary networks for I/O and management access. Jobs are typically launched and managed through a software infrastructure that includes special libraries such as MPI and special resource management software to ensure that the cluster is utilized efficiently. For reverse debugging to be an effective technique in HPC environments tools that provide the capability need to work within this distributed compute environment.

3.2 Multi-core and Multi-threads

HPC systems predominantly use commodity processors, almost all of which now are now multi-core with two or more cores per processor. Nodes typically have one or two sockets. Scientists and developers have a variety of strategies for taking advantage of these processing resources. Users with the highest memory and I/O requirements often run just one user task per node so that single task has full access to the nodes resources. For jobs that are more compute-heavy developers often structure codes to have two or more separate streams of computation per node. This can be done either by scheduling multiple MPI tasks on the node or by using multithreading to split the work within an MPI task over multiple cores.

If multiple MPI tasks are scheduled on a single node then the MPI library often offers ways to accelerate intra-node communication using shared memory. These mechanisms are much faster than using the network stack to move information from one process to another on the same machine. It is therefore important that the record and replay mechanism supports inter-process communication via shared memory. If a single MPI task is written to utilize operating system threads then the record and replay mechanism needs to support the debugging of multithreaded processes. In either case it is important that the mechanism used be deterministic when replaying both correctly- and incorrectly-written programs. Both shared memory and threading introduce the possibility of classical race conditions where two separate execution contexts are reading and writing to or from the same memory location. Any such behavior is likely to be of interest for troubleshooting and needs to be replayed exactly as it happened during recording.

3.3 Limited Memory on Compute Nodes

Perhaps the most significant challenge for applying record and replay debugging in HPC contexts is the limited unused memory that is typically available on the compute nodes of HPC clusters. It is not unusual to find nodes without any local disk storage and even if local disk storage is available many times the cluster is configured without any swap space for performance reasons. So the logical memory that is available is limited by the memory that is physically present in the node. The users program itself may be designed to utilize the majority of this memory, leaving very little on the side for instrumentation and recording.

3.4 Coordinating Time

One of the things that complicates reverse debugging on a distributed system is the very notion of time. Each process is running independently in a separate operating system instance and they are connected with a network which, however fast, still has latencies. How finely can time be measured on the nodes of the cluster? How does one line up the measurements made on separate nodes?

4 Solutions

4.1 Cluster Architecture

ReplayEngine is implemented as an add-on to the TotalView parallel debugger. As a parallel debugger TotalView provides mechanisms for launching MPI and UPC jobs and for attaching to all or a subset of the processes that make up an MPI or UPC job.[6,7] TotalView provides a feature- and function-rich debugging environment including MPI-specific capabilities like viewing data from across multiple processes and interrogating MPI message queues. UPC-specific capabilities include viewing distributed arrays and working with various kinds of UPC pointers to shared arrays.

TotalView is a parallel application that gets constructed alongside the user's parallel application. In particular the front-end `tvmain` process communicates to a distributed set of lightweight `tvdsrv` agent processes to control and query the users parallel program. Typically one `tvdsrv` process is started on each node that hosts a process that is part of the user's job, and that `tvdsrv` performs the low-level process control operations that facilitate debugging.

When ReplayEngine is added to the TotalView product there are changes in both `tvmain` and `tvdsrv`. The main TotalView process represents the additional concept of time to the user and offers the reverse operations. The TotalView agent processes take on the additional job of recording and deterministically replaying the target program through the use of an instrumentation library which gets loaded into the target. MPI communication is recorded by ReplayEngine as input in each MPI task that receives information from a point to point or collective operation.

4.2 Supporting Shared Memory and DMA for Communication

ReplayEngine handles shared memory by treating any reads from shared memory as inputs. This means that if shared memory is used within the MPI library to accelerate intra-node communication, this communication will be captured within MPI processes that are being recorded by ReplayEngine. So when developers want to examine execution history they can easily run back through sections of code that involve either inter- or intra-node communication, and any input into the process will be fed back in from the record of input at the appropriate time.

External Direct Memory Access (DMA) writes into the process are handled in the same way as shared memory; regions of memory that can be written to directly by the network interface card are regarded as potential input and subsequently any reads from those regions are monitored and the values read are logged as input.

In this way ReplayEngine handles a communication mechanism that is important to HPC type programs. There is some incremental overhead, both in the way that the recording is done and because there is additional data regarded as input and therefore logged. Failing to handle shared and DMA memory when

debugging a target that makes use of those techniques isn't an option however, since that would make the replay non-deterministic. If minimal tool overhead is desired users may be able configure their MPI libraries to avoid using shared memory and DMA techniques for communication.

4.3 Supporting Multi-threaded Targets

ReplayEngine handles multiple threads by controlling and recording the execution of threads at the level of individual machine instructions. While the process is being recorded or replayed under ReplayEngine only one thread at a time will execute. ReplayEngine notes where each thread starts and stops during a context switches and is able to ensure that on replay the execution sequence is exactly as it happened during recording.

This, like any instrumentation, can cause the execution order to deviate from what would happen with no instrumentation. While there is some potential that this might prevent defects from manifesting TotalView provides the user with a rich set of thread control features, so that during troubleshooting developers can drive the application through any sequence they desire in order to try to trigger the defect. ReplayEngine frees them up from having to do that over and over again once they have successfully triggered the bug.

The fact that only a single thread is allowed to progress at a time does have the potential to impact the performance of the application during both record and replay in applications that are both multithreaded and largely compute bound.

4.4 Managing Memory Overhead

ReplayEngine can work within user-specified memory limits. On a system with effectively infinite virtual memory resources ReplayEngine is happy to record the entire program execution history. Obviously this is ideal, as at the beginning of troubleshooting developers may not know how deep into history they will have to go to find the root cause of any specific defect. However when resources are more limited ReplayEngine can simply truncate the oldest part of the execution history as needed to make space for more recent history. Users can specify this maximum memory overhead in advance based on their knowledge of their application's memory usage.

There is a minimum memory overhead required for instrumentation so it is possible that some programs simply do not leave enough room for the ReplayEngine infrastructure to function.

5 Conclusion

With support for shared memory communication and the ability to work within defined memory limits ReplayEngine enables reverse debugging within HPC environments. Scientists and engineers working on distributed applications will have the ability to more easily answer the fundamental troubleshooting question "how did this happen?"

One consequence of the transition from restart-cycle debugging techniques to reverse debugging is that debugging can increasingly happen within a single session. This session will probably take longer than the individual sessions in restart-cycle debugging but the entire debugging process should typically take a smaller overall investment of time and occur over a shorter interval. With strictly-managed HPC cluster resources this may have implications for scheduling policy. If the duration of individual jobs tagged as debugging jobs is currently limited, cluster administrators should consider the possibility of easing those limitations to allow for a smaller number of longer but more efficient reverse debugging sessions.

I highlighted the challenge of coordinating time measurements across a cluster in order to establish a “unified history” for a distributed program. This challenge remains and will be the goal of future research and development. Other exciting areas for future development include improved presentations of information across time within the GUI and work-flows which might allow for decoupling of recording and replaying program behavior for offline analysis.

References

1. Bouteiller, A., Bosilca, G., Dongarra, J.: Retrospect: Deterministic replay of mpi applications for interactive distributed debugging. In: Cappello, F., Herault, T., Dongarra, J. (eds.) PVM/MPI 2007. LNCS, vol. 4757, pp. 297–306. Springer, Heidelberg (2007)
2. Xue, R., Liu, X., Wu, M., Guo, Z., Chen, W., Zheng, W., Zhang, Z., Voelker, G.M.: MPI WIZ: Subgroup Reproducible Replay of MPI Applications. In: Principles and Practices of Parallel Programming, Sheridan Printing (2009)
3. TotalView Technologies: TotalView Documentation End User Documentation (2009), <http://www.totalviewtech.com/support/documentation.html>
4. Gropp, W., Lusk, E.: MPICH2 (2006), <http://www.mcs.anl.gov/mpi/mpich2/>
5. Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B.W., Lumsdaine, A., Castain, R.H., Daniel, D.J., Graham, R.L., Woodall, T.S.: Open MPI: Goals, concept, and design of a next generation MPI implementation. In: Kranzmüller, D., Kacsuk, P., Dongarra, J. (eds.) EuroPVM/MPI 2004. LNCS, vol. 3241, pp. 97–104. Springer, Heidelberg (2004)
6. Cownie, J., Gropp, W.: A standard interface for debugger access to message queue information in MPI. In: Margalef, T., Dongarra, J., Luque, E. (eds.) PVM/MPI 1999. LNCS, vol. 1697, pp. 51–58. Springer, Heidelberg (1999)
7. Gottbrath, C., Barrett, B., Gropp, B., Lusk, E.R., Squyres, J.: An interface to support the identification of dynamic MPI 2 processes for scalable parallel debugging. In: Mohr, B., Träff, J.L., Worringen, J., Dongarra, J. (eds.) PVM/MPI 2006. LNCS, vol. 4192, pp. 115–122. Springer, Heidelberg (2006)

8th International Special Session on Current Trends in Numerical Simulation for Parallel Engineering Environments

New Directions and Work-in-Progress

ParSim 2009

Carsten Trinitis¹, Michael Bader¹, and Martin Schulz²

¹ Institut für Informatik, Technische Universität München

Boltzmannstr. 3, 85748 Garching, Germany

{Carsten.Trinitis,Michael.Bader}@in.tum.de

² Center for Applied Scientific Computing

Lawrence Livermore National Laboratory, Livermore CA, USA

schulzm@llnl.gov

In today's world, the use of parallel programming and architectures is essential for simulating practical problems in engineering and related disciplines. Significant progress in CPU architecture (multi- and many-core CPUs, SMT, transactional memory, virtualization support, shared caches etc.) system scalability, and interconnect technology, continues to provide new opportunities, as well as new challenges for both system architects and software developers.

These trends are paralleled by progress in algorithms, simulation techniques, and software integration from multiple disciplines.

In its 8th year, ParSim continues to build a bridge between application disciplines and computer science and to help fostering closer cooperations between these fields. Since its successful introduction in 2002, ParSim has established itself as an integral part of the EuroPVM/MPI conference series. In contrast to traditional conferences, emphasis is put on the presentation of up-to-date results with a short turn-around time. We believe that this offers a unique opportunity to present new aspects in this dynamic field and discuss them with a wide, interdisciplinary audience. The EuroPVM/MPI conference series, as one of the prime events in parallel computation, serves as an ideal surrounding for ParSim. This combination enables participants to present and discuss their work within the scope of both the session and the host conference.

This year, five papers from authors in five countries were submitted to ParSim, and we selected three of them. They cover a range of different application fields including mechanical engineering, material science, and structural engineering simulations. We are confident that this resulted in an attractive special session and that this will be an informal setting for lively discussions as well as for fostering new collaborations.

Several people contributed to this event. Thanks go to Jack Dongarra, the EuroPVM/MPI general chair, and to Jan Westerholm, Juha Fagerholm and

Jussi Heikonen, the PC chairs, for their encouragement and support to continue the ParSim series at EuroPVM/MPI 2009. We would also like to thank the numerous reviewers, who provided us with their reviews in such a short amount of time (in most cases in just a few days) and thereby helped us to maintain the tight schedule. Last, but certainly not least, we would like to thank all those who took the time to submit papers and hence made this event possible in the first place.

We are confident that this session will fulfill its purpose to provide new insights from both the engineering and the computer science side and encourages interdisciplinary exchange of ideas and cooperations, and that this will continue ParSim's tradition at EuroPVM/MPI.

Carsten Trinitis

Lehrstuhl für Rechnertechnik und Rechnerorganisation (LRR)

Institut für Informatik

Technische Universität München, Germany

Carsten.Trinitis@in.tum.de

Michael Bader

Lehrstuhl für wissenschaftliches Rechnen

Institut für Informatik

Technische Universität München, Germany

Michael.Bader@in.tum.de

Martin Schulz¹

Center for Applied Scientific Computing

Lawrence Livermore National Laboratory

Livermore, CA

schulzm@llnl.gov

¹ Part of this work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-413824).

A Parallel Simulator for Mercury (Hg) Porosimetry

C.H. Moreno-Montiel¹, F. Rojas-González², G. Román-Alonso¹,
S. Cordero-Sánchez², M.A. Castro-García¹, and M. Aguilar-Cornejo¹

¹ Departamento de Ingeniería Eléctrica

² Departamento de Química

Universidad Autónoma Metropolitana - Iztapalapa, P.O. Box 55-534 México D.F., México
`{chmm, fernando, grac, sscs, mcas, mac}@xanum.uam.mx`

Abstract. A parallel simulator, based on the Dual Site-Bond Model of complex media, is developed to study Hg intrusion and extrusion processes in the myriad of voids contained in a porous network. In order to reduce the requirements in RAM and computing resources, the porous network is partitioned into several sub-networks distributed in different cluster processors. The simulator uses shared memory to process (with OpenMP) each sub-network and applies a message passing protocol (with MPI) to allow communication among different processors. We show experimental results that reflect a good performance of our proposal when using different sizes of porous networks in a cluster with 32 nodes, each one having 4 processors.

1 Introduction

Parallel computing is a powerful alternative in the study of many physical phenomena due to the great reduction of computation times and the attained good efficiency that results through its application [1,5]. This is especially true in models where the number of entities (data) is very large and the involved computational effort is intrinsically expensive. An important area of application is the simulation of porous media and the physicochemical phenomena occurring therein. The number of pores that exist in a typical porous sample is usually of the order of millions, billions or even trillions of them per unit mass of solid. These pores are generally interconnected to each other by way of a sinuous 3-D pathway. In lattice models [3], the porous space is distributed between two types of elements, the sites (cavities) and the bonds (necks). These two entities are interconnected alternatively throughout the space and imitate the void phase of the solid substrata. Hg porosimetry comprises two stages. The first stage, intrusion, starts with the immersion of a porous sample in Hg. As the pressure of Hg is increased, the pore entities are penetrated sequentially, i.e. from the largest to the smaller ones according to the current value of the external pressure. The second stage, retraction, consists in the withdrawal of Hg from the pores. Since this last process involves the gradual decrease of the external pressure, the succession by which pores are emptied goes from the smallest to the largest ones. In this work, we develop a parallel simulator related to the Hg intrusion and retraction processes in lattice substrates. Contrastingly, a previous simulator was built based on an iterative (non-parallel) algorithm having serious memory and computing shortcomings[2]. The developed simulator works on a cluster

of multiprocessor nodes by exploiting the shared memory architecture inside a node and the message passing communication among different nodes. This simulator allows an efficient study of Hg porosimetry experiments when working with lattices displaying several memory requirements. The simulation is done by partitioning a lattice into several parts or sub-lattices; each part is allocated into a different node of the cluster. After performing a step of either intrusion or retraction on a sub-lattice, a node should exchange information with its neighboring nodes in order to know how Hg is being allocated inside the network. We used both OpenMP and MPI tools [4] for the implementation of the parallel simulator and used a cluster of 32 nodes, each one having 4 processors.

In Section 2, we describe the physical models that are employed to represent both the lattice and the Hg porosimetry processes. In Section 3, the Hg porosimetry algorithm is described. The parallel simulator is presented in Section 4. Section 5 shows and discusses the results. In Section 6, incumbent conclusions are formulated.

2 On Modelling Porous Networks from the Dual Site-Bond Model (DSBM)

Computationally, a porous network can be generated by considering two types of pore entities distributed throughout a solid matrix: the sites (cavities) and the bonds (throats). A given site is connected to C neighboring sites (as shown in Figure 1.a) through the same number of bonds; C being the connectivity of the network. The size of the throat cannot be larger than the size of the cavity to which it is connected. This restriction is called the Construction Principle (CP). A simple way to visualize a porous network under the CP perception is to represent the sites as hollow spheres that are connected to each other through C hollow cylinders (Figure 1.b). The modeling just described is known as the DSBM model[2].

In this work, we consider a cubic network with a connectivity C= 6. Each network has LxLxL sites and associated bonds; L being an indicator of the network size. In Figure 1.c, we schematize the modelling of a network of size L=3.

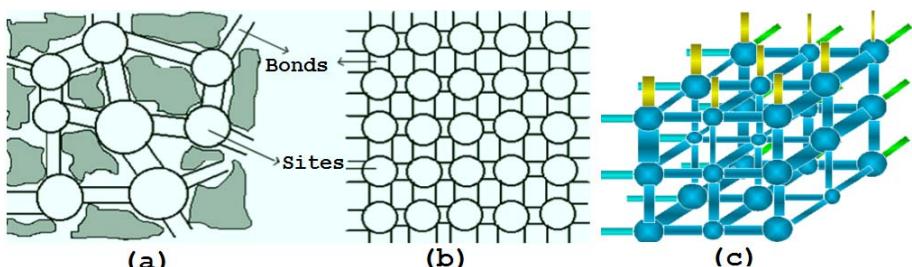


Fig. 1. (a) DSBM representation of the porous medium as an interconnected collection of sites and bonds. (b) Lattice model with C=4. (c) A DSBM network with C= 6 and L = 3.

3 Hg Porosimetry

When a fluid invades a porous medium there arises a diversity of physicochemical phenomena that erupt during the entrance or exit of fluids to or from the pores. A capillary process is defined as that in which two or more fluids compete for the possession of the empty space within a porous network when either adhesion or capillary (i.e. cohesion or disintegration) forces are dominant. Among capillary processes, Hg intrusion and extrusion are useful tools to determine several important textural properties of porous media such as mean pore size, porosity, specific surface area, and mean connectivity [6]. The intrusion and extrusion processes consist in either forcing Hg into the voids of the porous substrate by increasing the pressure of a second fluid (oil) in contact with the Hg source or by releasing this pressure to recover the Hg phase from the pores. Applications of Hg porosimetry include studies in the cement industry with regard to the effects that the pore size distribution has on curing time, stress resistance, and crack development. Hg porosimetry has also been employed to understand the fragility of bones in archaeological studies due to the porosity change caused by the loss of proteomes and minerals. Besides, the textural properties of gels, aerogels, and pharmaceutical products can be determined by this characterization technique.

3.1 The Intrusion Process

The intrusion of a pore with Hg can be simulated from an equation that considers the conditions for spherical (sites) or cylindrical (bonds) pores to be intruded with Hg or dispossessed of it. The pressure P^l required to intrude a cylindrical bond of radius R_B is given by the Washburn equation:

$$P^l = \frac{2\sigma^{lv} \cos\theta}{R_B} \quad (1)$$

Where σ^{lv} represents the $l-v$ interfacial tension, and θ is the contact angle at the three-phase line of contact.

The *Hg* intrusion of a site requires at least one of its surrounding bonds to be permeated by the liquid phase and to fulfill some other conditions as explained below. The intrusion algorithm employed in this work is as follows.

- Invasion of peripheral bonds. *Hg* intrusion starts when this liquid is forced to enter through the bonds positioned at the outer layer of the substrate. In concordance with equation 1, a bond is intruded if its size R_B fulfills the condition $R_B > R_C$, R_C is the radius of the smallest bond that can be intruded with Hg at pressure P^l .
- Invasion of sites. To penetrate their adjoining sites, a canthotaxis effect needs to be overcome; this consists in the temporary anchoring of the advancing interface at the site-bond edge until the advancing meniscus steadily swells for raising its curvature and subsequently intrudes the connected site. For a site to be invaded by Hg, a continuous liquid path must exist from the site in question to the Hg source.
- Invasion of internal bonds. The bonds lying inside the porous network are invaded by Hg when a meniscus proceeding from a neighboring site has sufficient curvature (i.e. pressure) as to surmount the energy necessary to occupy the capillary in

question. Besides of having the right size, a continuous liquid path is required for this bond invasion to take place.

The sequential program for Hg intrusion is designed through a recursive algorithm that consists in the Hg invasion of bordering bonds of the cubic lattice. If a given one of these bonds is penetrated by Hg, the connected site is analyzed and in case of fulfilling the conditions of: (i) being empty, (ii) being of the right size, and (iii) having a continuous path to the Hg source, the site is invaded, otherwise it remains empty. Once a site is filled it remains to check if any other one of the remaining C neighboring sites is filled with liquid; if sites are being invaded by Hg, it is necessary to check in a recursive manner for more sites or bonds of the network that are being filled with liquid. The recursive process ends when no more pore entities are invaded with Hg when repeating the recursive filling during each pressure step. The intrusion process continues toward the inner sites of the network until most, if not all, sites are occupied by liquid.

3.2 The Retraction Process

This process occurs when the pressure exerted on the Hg source is progressively released and starts when most pores are filled with Hg. For Hg to abandon a given bond, several conditions must be simultaneously fulfilled: (1) the bond size should be smaller than a critical radius, (ii) there should exist a continuous liquid path from the pore to be drained of Hg toward the outer liquid source. For this last condition to occur, the bond has to be simultaneously connected to two sites: one of them empty of liquid (i.e. where the liquid-vapor meniscus rests) and the other one full with Hg (for the liquid phase to recede through a continuous path to its bulk source). Besides of having the right size, for a site to be emptied of Hg during a retraction operation, the cavity must be connected to an empty bond (this fact secures the existence of a liquid-vapor interface), concurrently another one of the delimiting bonds must be filled of condensate in order to ensure a continuous liquid path to the outer Hg source.

Nevertheless, it is important to mention that during the retraction process, there occurs the so-called *snap-off* phenomenon. This effect consists in the breakage of the liquid phase located inside a bond into two residual vapor bags at opposite sides of the bond under attention. If these bags coalesce with each other when the pressure is further reduced, there occurs a disconnection of the liquid phase at the bond undergoing snap-off. This disconnection can bring about the apparition of vapor islands of diverse sizes distributed at different regions of the porous network. In summary, the retraction algorithm is as follows:

- A critical snap-off bond size, $R_{C_{snap-off}}$, is imposed thus meaning that bonds smaller than this value can undergo this phenomenon when the pressure is lowered
- At a given pressure any bond of size $R_B < R_C$ (consistent with a critical radius R_C) is going to be freed of Hg through a *piston-like* mechanism, i.e. the interface recedes inside the bond, if it is also linked to a continuous liquid path that leads to the outer liquid bulk source
- The retraction of the liquid-vapor interface from a site of size R_S can happen via a piston-like mechanism when $R_S < R_C$ together with the fact of being connected to the bulk Hg source by a continuous path.

We first start with the implementation of a sequential version of the simulator in which only one processor visits all the lattice bonds and sites during both Hg intrusion and retraction processes. This sequential version of the simulator spends a lot of RAM resources; therefore problems related with large size lattices cannot be studied in a personal computer.

4 The Proposed Parallel Simulator

We first propose one version in which a complete lattice is processed in parallel by a set of threads in a processor, sharing memory to communicate with each other. The second version of our simulator uses more than one processor; here the complete lattice is partitioned into several sub-lattices that are allocated in different processors. At each processor, we continue applying a processing equivalent to the parallel behavior of our first version, but now we add a message passing protocol among processors to exchange information about the current Hg pore occupation.

4.1 Shared Memory Version

The first parallel version was designed to be executed in shared memory systems like multi-core or multi-processor nodes (using OpenMP). Here, we have several execution threads sharing the complete lattice space. The intrusion or retraction process are performed by several threads working simultaneously while starting from different external lattice bonds. Likely, a good number of threads should be 6 (considering the cubic lattice geometry) one thread per external face of the cubic lattice; however, it is also possible to handle multiple threads if we have nodes with more processing units. If the number of threads is larger than six, for example N, then $\frac{N}{6}$ threads will start from different external bonds of the same face. In Figure 2, we show how the external lattice faces are distributed among 6 threads (a) and 12 threads (b) to start the processing.

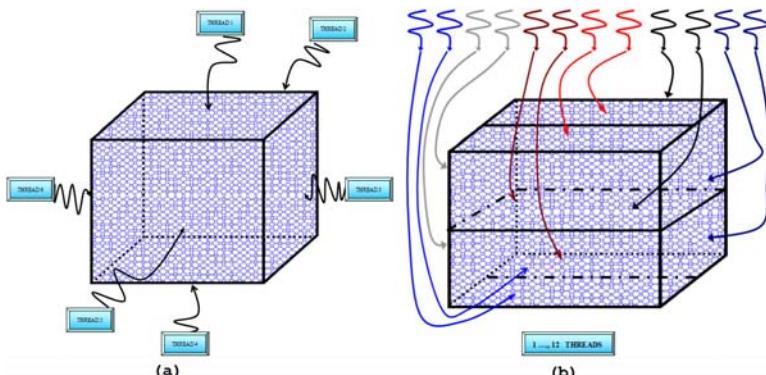


Fig. 2. Parallel shared memory processing, using 6 (a) and 12 (b) threads

Once a thread starts a Hg porosimetry process, it continues as in the sequential recursive algorithm until the termination conditions are reached.

4.2 Addition of a Message Passing Protocol

The previous simulator version was improved by adding the use of all the multiprocessor nodes of a cluster. Here, we continue exploiting the shared memory resources at each node, but to exchange information among processors a message passing protocol is added. The complete cubic lattice is divided in several parts, and each sub-lattice is distributed among the processors (using MPI), thus diminishing the RAM requirements per node. To partition a lattice, we begin by dividing it at the middle of the Z axis thus partitioning the lattice into two sub-lattices. For having 4 partitions, we divide the lattice at the middle of the X axis, thus resulting 4 sub-lattices that have two internal and four external faces. Eight partitions are generated when dividing the lattice at the middle of the Y axis, as shown in Figure 3.a. For more than 8 partitions, subsequent divisions can only be made at the Y axis; in this way, a generated sub-lattice has at least two external faces. In Figure 3.b, a lattice partitioned into 16 parts is shown. After the partitioning, each sub-lattice is allocated into a different processor.

Partitions are made at the bonds that are shared by two sites. After the partitioning, each processor creates a bond interface that is integrated with six bond faces of its corresponding sub-lattice, as shown in Figure 3.c. A face contains external or internal sub-lattice bonds. Adjacent sub-lattices share the same bond values in one of their interfacial faces. The Hg intrusion/retraction simulation in each sub-lattice is very similar to the parallel process made in the shared memory version, but here at the beginning the threads start working only on the external faces. A processor now transfers a report about Hg occupation arising inside its own sub-lattice, to the processors of its adjacent sub-lattices (called neighbors). After a local intrusion/retraction in a sub-lattice, the bond interface is exchanged between each processor and its neighbors.

When a process receives the bond faces from its neighbors, it then compares them with their own (with side correspondence). If the own interface have the same information than the received one, the algorithm stops. When some corresponding bonds are

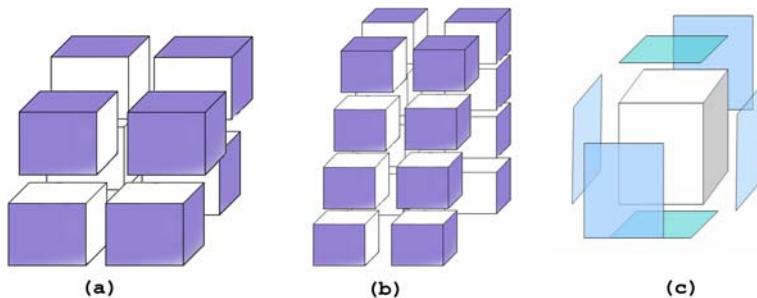


Fig. 3. Partitioning of a lattice into:(a) 8 and (b)16 sub-lattices. (c) Interface of a sub-lattice integrated by 6 bond faces.

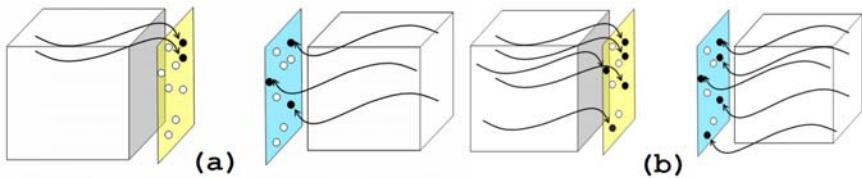


Fig. 4. Two corresponding bond faces with (a) different, and (b) equal values in all bonds

different, that is if in one face a bond is empty and in the correspondent face the same bond is filled with Hg, the intrusion or retraction process continues now starting from those bonds. In the example of Figure 4.a, two sub-lattices generate a bond interface and after suffering a local intrusion several interface bonds are invaded by Hg through one of the sub-lattices but not at the other one, therefore the intrusion processing continues in both sub-lattices from the bonds that were invaded. In Figure 4.b, we have the same example but now all the interfacial bonds are in the same state, so that the intrusion process stops at a certain pressure. Once there are no more interfacial exchanges, at a certain pressure, the processing is repeated by increasing or decreasing the pressure until the lattice reaches the maximum level of Hg intrusion or retraction.

5 Results

First, we show the results when using only one node (One-node version). Then, we show the advantages obtained when employing a cluster of 32 4-processor nodes (Cluster version) that combine shared memory and message passing models.

The performance of our simulator executed in One-node depends on the number of threads created to process a lattice. To determine which configuration attains the best results, several experiments were performed by using lattices of different sizes ($L=25$, 50 , 75 and 100) and by increasing the number of threads; we used 2 , 4 , 5 , 6 , 8 , 10 , 12 and 36 threads working with shared memory on a node with 4 processors. In Figure 5.a and 5.b, the execution times obtained for the Hg intrusion and retraction process are plotted; in both cases, the behavior is similar.

A Hg typical porosimetry process begins from the 6 external faces of the network, these can be divided by the number of threads at the node. If the node has 4 processors and only 2 threads are used, the parallelism is not adequately exploited. If we use 4 threads, in a first step the threads then work simultaneously on 4 different external surfaces, thus remaining 2 faces to be intruded/retracted. Only after the conclusion of the Hg porosimetry process is reached, in a second step two of the 4 threads continue working with the remaining 2 faces. Here, a dependence restriction makes two threads heavier than the other ones.

When using 6 threads on a 4 -processors node, $h1$ to $h6$, each thread works with a different external face. Despite the fact that at the beginning 2 pairs of threads share the same processor, eg $h1-h5$ and $h2-h6$, there is now no longer a dependence restriction. Even better, when $h3$ and $h4$ finish their work, threads $h5$ or $h6$ can be executed by

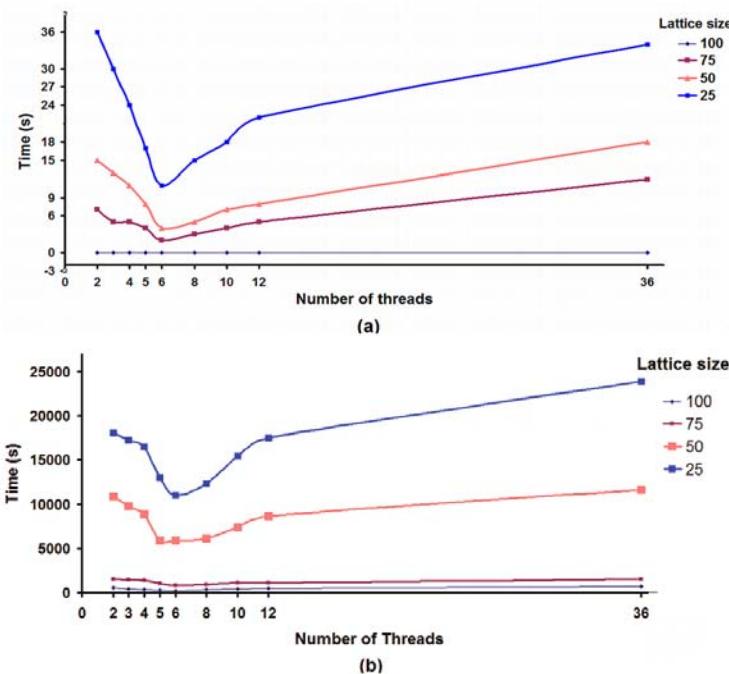


Fig. 5. Execution times for the (a) intrusion and (b) retraction processes in lattices of several sizes, using One-node version and augmenting the number of threads

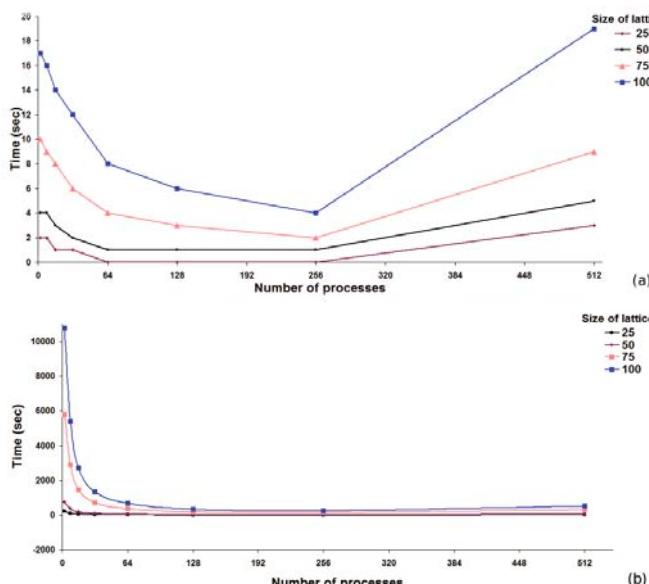


Fig. 6. Execution times for the (a) intrusion and (b) retraction processes simulation in lattices of several sizes, using the cluster version, and augmenting the number of processes

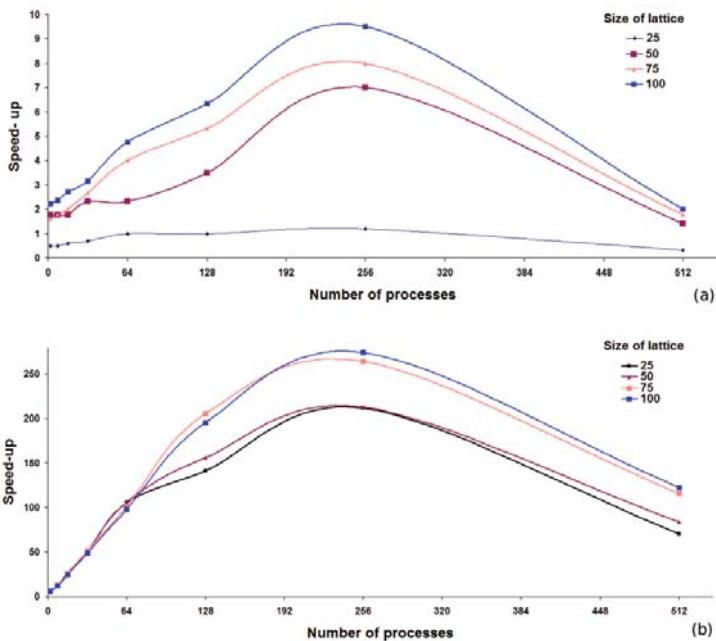


Fig. 7. Speed-up plots for (a) intrusion and (b) retraction simulation

an idle processor, thus supporting a good load balancing and attaining the best execution time. When using more than 6 threads an external face can be shared by two or more threads, this increases the execution time due to the cost of context changes of threads in each processor.

In the Cluster version it is needed to determine which processes number is ideal for obtaining good performance. The number of processes is related to the number of lattice partitions, the more partitions, the more processes are needed. The partitions defined in our tests required the creation of 2, 8, 16, 32, 64, 128, 256, and 512 processes, executed in a cluster with 32 nodes (Technical University of Munich cluster), each one having 4 processors (64 bits, 8-16GB RAM, 2.4GHz). Figures 6.a and 6.b show the execution times obtained for Hg intrusion and retraction processes.

A static partitioning is applied so that once a sub network or partition is allocated to a processor, it must be completely processed therein. The expected best distribution was that running 1 process per processor; however, the configuration that provided the best results employed 256 processes (2 processes per processor). In the 128 processes version some processors became idle while others were working at 100%. A justification of this is that sub-lattices have different processing times. Maybe in a given sub-lattice, the condition of its pores makes the intrusion/retraction processes more difficult to perform while in other sub-lattice of the same size it could happen a faster processing. When using 256 processes, allocating 2 processes (sub networks) per processor, fewer processors are out of work. If a network is difficult to process,

the partitioning into two sub networks assigned to different processors enhances the load distribution. An underloaded processor will be able to cooperate in processing a second sub network. The partitioning means that the amount of processing for each network decreases, but it also increases the number of communications. There should be a compromise between the amount of partitioning and the number of generated communications, so that the gain in processing time will not be affected by the increase in communications. In our application, when using 4 processes per processor the performance deteriorated.

To estimate the Speed-up, we considered the sequential and parallel times obtained with the Cluster version. When comparing Cluster vs. sequential procedures, the execution time of the Cluster version decreases more than 100 times for large size lattices thus allowing an efficient study of Hg porosimetry. For lattices of large sizes, the execution time of the Sequential version behaves in an exponential way, due to the expensive frequent swap accesses generated when RAM memory is being exhausted. We can see in Figures 7.a and 7.b how Speed-up changes when the number of processes increases, for Hg intrusion and retraction, respectively. The best Speed-up is obtained when using 256 processes and all processors, showing good scalability.

6 Conclusions

A parallel simulator is proposed for the study of Hg porosimetry. The simulator exploits the resources of a cluster of 32 4-processors nodes using shared memory (with OpenMP) and distributed memory protocols (with MPI). An initial version was implemented where a lattice is processed by following a shared memory parallel model, in one multi-processor node. The intrusion and retraction processes got the best execution times when using 6 threads. Based on these results, a second simulator version was created by partitioning the 3D-lattice into several sub-lattices assigned in different processors while distributing the whole RAM requirements and computing resources among the whole set of processors. The shared memory resources at each node were exploited, but a message passing protocol was added to allow the communication among the processors. A processor needed to exchange information about Hg movement through the sub-lattice borders with the adjacent processors. The best Speed-up was reached when 256 processes were employed, showing good scalability.

References

1. Kalé, L., Skeel, R., Bhandarkar, M., Brunner, R., Gursoy, A., Phillips, J., Krawetz, N., Shinozaki, A., Varadarajan, K., Schulten, K.: Namd2: Greater scalability for parallel molecular dynamics. *Computational Physics* 151, 283–312 (1999)
2. Kornhauser-Straus, I., Cordero-Sánchez, S., Felipe-Mendoza, C., Rojas-González, F., Ramírez-Cuesta, A.J., Riccardo., J.L.: On comparing pore characterization results from sorption and intrusion processes. In: *Fundamentals of Adsorption 7*, Proceedings of the 7th International Conference on Fundamentals of Adsorption, pp. 1030–1037 (2001)

3. Mayagoitia, V., Rojas, F., Kornhauser, I., Zgrablich, G., Riccardo, J.: Fluid-phase morphologies induced by capillary processes in porous media. In: Characterization of Porous Solids III. Studies in Surface Science and Catalysis, vol. 87, pp. 141–150. Elsevier, Amsterdam (1994)
4. Quinn, M.J.: Parallel Programming in C with MPI and OpenMP, 1st edn. McGraw-Hill, New York (2003)
5. Wiley, R.L.: Parallel processing and numerical weather prediction. In: Second International Specialist Seminar on the Design and Application of Parallel Digital Processors, April 1991, pp. 33–37 (1991)
6. Zgrablich, G., Mendioroz, S., Daza, L., Pajares, J., Mayagoitia, V., Rojas, F., Conner, W.C.: Effect of porous structure on the determination of pore size distribution by mercury porosimetry and nitrogen sorption. Langmuir 7(4), 779–785 (1991)

Dynamic Load Balancing Strategies for Hierarchical p -FEM Solvers

Ralf-Peter Mundani¹, Alexander Düster¹, Jovana Knežević², Andreas Niggl³,
and Ernst Rank¹

¹ TU München, Chair for Computation in Engineering, 80333 München, Germany

² University of Belgrade, Department of Mathematics, 11000 Belgrade, Serbia

³ SOFiSTiK AG, 85764 Oberschleissheim, Germany

Abstract. Equation systems resulting from a p -version FEM discretisation typically require a special treatment as iterative solvers are not very efficient here. Applying hierarchical concepts based on a nested dissection approach allow for both the design of sophisticated solvers as well as for advanced parallelisation strategies. To fully exploit the underlying computing power of parallel systems, dynamic load balancing strategies become an essential component.

1 Introduction

Within computational engineering, structure dynamics are one main source for challenging numerical computations. Here, the p -version of finite element methods is a very prominent technique allowing to increase accuracy without increasing the amount of elements, too. Nevertheless, the resulting equation systems fail to be efficiently solved with iterative methods such as CG or multigrid and, hence, need to be processed via expensive direct solvers (Gauss and relatives) that also entail limited potential for the parallelisation. Applying hierarchical methods based on the nested dissection concept opens the door to sophisticated solvers but also to new problems concerning the scalability of parallelisation strategies related to tree structures. Hence, an optimisation of both run time and parallel efficiency arises the necessity of dynamic load balancing strategies that are able to exploit the underlying hierarchy and, thus, to leverage parallelisation on all levels ranging from multithreading to distributed computing.

In this paper, we will show the hierarchical organisation of the p -version using octrees and their advantages for the solution of equation systems and parallelisation. Furthermore, we will show a dynamic load balancing strategy that allows to tackle the scalability problem which is essential — for instance — within interactive computational steering applications in order to achieve small run times and, thus, high update frequencies. As this paper describes work in progress, we will mainly highlight concepts and their benefits for parallelisation instead of concrete benchmark results which are subject to future research.

2 Structural Analysis of Thin-Walled Structures

The development of accurate and efficient element formulations for thin-walled structures has been in the focus of research in Computational Mechanics since the advent of the finite element method. At a very early stage it seemed to be clear that an investigation of plate or shell problems with tetrahedral or hexahedral elements is not feasible for practical problems, as a sufficient accuracy could only be obtained by a prohibitively large amount of degrees of freedom and computational effort. The major reasons for this observation are the mapping requirements of isoparametric, low order elements. Accurate solutions can only be obtained if the ratio aspect of an element is close to one, resulting in an enormous amount of elements, even if only one or a few layers are used over the thickness of the structure. A natural consequence of this observation was to use dimensionally reduced models like Reissner-Mindlin plates or Naghdi shells, and to build element formulations based on these theories. However, it turned out that pure displacement type elements for these models lead to notorious numerical problems like locking, giving rise to the development of numerous improvements, like mixed elements, for example.

The approach presented in this paper is different from concepts usually applied when low order elements are chosen. The idea is to construct a hierachic family of high order elements for both thin and thick-walled structures to make it possible to control the model error inherent in every plate or shell theory by simply increasing the polynomial degree of the trial or Ansatz functions in thickness direction. The high order finite element approach for three-dimensional thin and thick-walled structures is based on a hexahedral element, applying hierachic shape functions [8,1,9,2]. The present implementation not only allows the polynomial degree to be varied for the three different local directions but also a different degree to be chosen for each primary variable, reducing the numerical effort significantly.

Figure 1 depicts a hexahedral element, discretising a part of a thin-walled structure. Since the blending function method is used (see [8,1,9], e. g.) the geometry of the discretised domain may be quite complex. The shell-like solid may, for instance, be doubly curved with a non-constant thickness. When thin-walled structures like the one depicted in Figure 1 are to be discretised, it is important

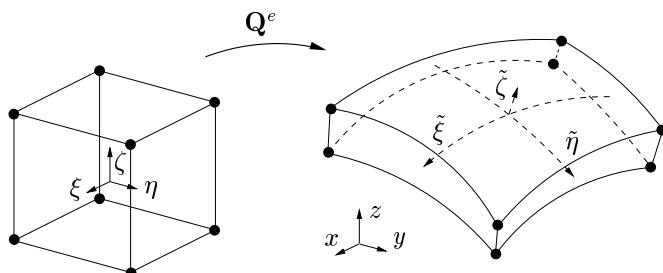


Fig. 1. Curved shell-like solid element of high order

to treat the in-plane direction (ξ, η) and the thickness direction (ζ) differently. This can be accounted for by using anisotropic Ansatz functions for the three-dimensional displacement field $\mathbf{u} = [u_x, u_y, u_z]^T$. In some situations it may be sufficient to restrict the polynomial degree of the Ansatz in thickness direction (ζ) to a certain degree, for example, $q = 3$ whereas the Ansatz chosen for the in-plane direction (ξ, η) is to be of order $p \geq 3$.

Since the p -version is less prone to locking effects [8,9], a pure, strictly three-dimensional displacement formulation can be applied. The numerical effort related to the computation of thin as well as thick-walled structures based on this formulation has two major sources: the computation of the element stiffness matrices and the solution of the resulting linear equation system. Comparing classical dimensionally reduced low-order finite elements for plates or shells with the proposed high-order formulation it turned out in many benchmark computations [1,9] that the high-order approach needs much less degrees of freedoms for the same accuracy and shifts the computational work from the global level (solution of equation system) to the local level, i. e. the computation of element matrices which is due to the numerical integration of the element matrices numerically quite demanding. Considering the parallelisation of this approach this can be regarded as an important advantage since the computation of the element matrices does not necessitate any communication and can, thus, be parallelised very efficiently, see [7]. Whereas the parallelisation of the computation of the element stiffness matrices is straightforward, the solution of the resulting linear equation system is more involved. This is due to the fact that the parallel solution of the equation system can not be carried out without any communication between the parallel processes. Furthermore, we are restricted in the choice of the solver since we apply a strict three-dimensional element formulation which results in a equation system with a poor condition number when discretising thin-walled structures. Therefore, iterative procedures, such as the preconditioned conjugate gradient method, for example, turn out to be not efficient. This drawback is even more pronounced when nonlinear problems of structural mechanics (hyperelasticity, elastoplasticity, etc.) are considered, worsening the condition of the equation system. We therefore prefer to apply a direct solver which will be described in the next section.

3 Hierarchical Organisation of the p -Version

As described in the previous section, iterative solvers are not very efficient due to the poor condition number of the equation system. Even direct solvers are advantageous here, they nevertheless suffer from a high computational complexity and they typically entail extensive parallelisation strategies in order to exploit the underlying computing power. Hence, different approaches are necessary to cover the aforementioned problems. Well-known from the field of domain decomposition is the nested dissection method that was first introduced by J.A. George [3]. The basic idea of nested dissection (ND) is to recursively subdivide the computational domain and to set up a local equation system on each subdomain

to be solved in a bottom-up step by successively eliminating local influences (computing the so-called Schur complement).

For our p -version this means to start from the computed element stiffness matrices and to organise those in a hierarchical way in order to apply ND. Therefore, octrees are used as their underlying principle of spatial partitioning is advantageous for the hierarchical organisation of the p -version. Element stiffness matrices are stored to the octree's leaf nodes — each leaf node stores at most one element or stays empty — while degrees of freedom (DOF) are stored to leaf nodes and inner nodes. The corresponding node for storing a DOF can easily be determined by finding the common parent node of all leaf nodes, i. e. element stiffness matrices, that share this DOF. It's obvious that the closer some DOF is stored to the tree root node the later it will be eliminated and the more processing time has to be invested [6]. The position of a DOF in the tree highly depends on the spatial partitioning of the domain. As octrees always halve each spatial dimension in every step, structures with huge dimensions in length and small dimensions in width and height, for instance, suffer from too many DOFs being concentrated in the root node. Here, a two-stage approach will help that halves only in one spatial dimension as long as the size of the subdomain is larger than some threshold value [10].

The results achieved with ND so far are very promising, especially ND allows to vastly reduce the computational complexity in case the underlying structure changes. As only those tree branches that contain a modification (material parameters, e. g.) have to be re-computed, all others stay untouched and the Schur complements that have been computed in a previous run can be re-used [5]. Nevertheless, the computational effort for complex scenarios is too high for retrieving results in real time — as necessary within interactive computational steering applications, for instance — thus, parallelisation is inevitable. Efficient strategies for the dynamic parallelisation of ND are subject of the next section.

4 Parallelisation Strategies

For the parallelisation of our ND approach we want to address both multi-threading and distributed computing. This allows us to easily incorporate latest developments in hardware such as multi- or manycore CPUs as well as to provide the necessary flexibility towards a unique workload distribution which — as we will see — is quite difficult to achieve and moreover plays a dominant role concerning fair speed-up and scalability values. Therefore, starting from a classical tree parallelisation we will then advance to a sophisticated dynamic load balancing strategy.

4.1 Problem Analysis

One main advantage due to the hierarchical organisation of ND via octrees is the pure vertical communication between parent and children nodes (upwards for sending Schur complements and downwards for receiving the solution). Hence,

cutting the tree at some level L — defining $L = 0$ for the root level — leads to 8^L independent sub-trees which could be processed in parallel. Assuming now a full and balanced tree with N leaf nodes, i. e. N element stiffness matrices, one would achieve the highest parallelism for cutting this tree at level $L = \lceil \log_8 N \rceil - 1$. Nevertheless, speed-up and efficiency are very limited, as this approach is similar to the problem that Minsky et al. posed in [4] concerning the parallel summation of $2N$ numbers on N processors. As the amount of active processes decreases in our case by a factor of 8 in each ND level, the possible values for speed-up and efficiency are slightly better, nevertheless far away from being a satisfying result due to the huge amount of inactive processes and the bad scalability inherent to this approach.

In order to achieve good results for both, i. e. speed-up and scalability, some efficient load balancing strategy is inevitable. A master-slave approach will serve as starting point here, nevertheless arising the necessity for being adopted to the underlying problem. First, when dealing with a large amount of slaves one single master might become the bottleneck due to a huge communication advent, hence, a multi-level concept is required. Furthermore, independent tasks have to be identified and according to their dependencies on the results of other tasks then administrated by those masters. Tasks per se are processed by the slaves in parallel and should incorporate (simultaneous) multithreading to speed-up local computations. As this strategy also covers distributed computing, topics such as distributed storage are of high relevance but not part of our work in the current stage.

4.2 Task Management

Before single tasks might be administrated by some master, an equal distribution of tasks among all masters has to be initiated. Therefore, we choose a 2-level hierarchy with one master on the first level and several masters on the second. To distinguish between those masters, the one on the first level is called main master and the rest are called traders. In case of more than two levels, only the masters on the lowest level are traders, the rest are main master, second masters, third masters and so on. The difference between masters and traders is that only the masters are serving requests from the slaves, delegating these requests to the corresponding traders which then take care about the real data exchange.

To initiate a work load distribution, the main master starts to analyse the octree structure and estimates the amount of work load, i. e. the amount of necessary elimination steps for computing the Schur complement, in each node. Based on these values he is able to predict the total amount of work and, thus, to decide how many traders and how many slaves should be spawned. Furthermore, the estimated work load in each node allows the master to distribute the octree among the traders more equally even in case of very imbalanced trees. This is not the case when just cutting an imbalanced tree at some level L and assigning the resulting sub-trees to the traders. Nevertheless, to achieve an equal distribution the tree might be cut into much more parts than traders, thus, one trader has to administrate several sub-trees which do not always preserve neighbourhood

relations. This might entail further complexity due to more communication but could not be observed so far for the current implementation.

Once sub-trees have been assigned to a trader, the trader himself analyses the respective tree structures in order to determine all tasks, i. e. the equation systems in all nodes, together with their dependencies on child-nodes concerning the input data (Schur complements). The tasks that have been identified together with their dependencies are then stored to a priority queue which is updated by the trader each time some slave returns the results of its computation. That means, tasks in the queue are checked if any of their dependencies are fulfilled. Tasks without further dependencies are ready to be processed by a slave and therefore get a higher priority while tasks that cannot be processed yet have a priority $p = \infty$. The trader picks one task among those ready for processing and sends the corresponding task ID to the master for being advertised to the slaves. The master stores tuples consisting of task ID and trader ID — one tuple per trader — in order to serve requests from the slaves. Hence, the master is not involved into heavy communications as he just has to tell the slaves which trader to contact for which task.

4.3 Processing Tasks

Slaves always contact the master to request new tasks. If tasks are available, the master picks one and sends the tuple task ID and trader ID as answer back to the slave before he continues serving the next request. Receiving such a tuple, a slave can now contact the trader and request the corresponding task for a local processing. Depending on the type of task the trader initiates a data transfer, consisting of an element stiffness matrix or several Schur complements. Again, within the current implementation we do not cover distributed storage. In this case, the trader would send a mixture of locally stored data and keys to remote repositories where the slave can retrieve the rest of the information. When the data transfer has been finished, the slave holds all subsequent data to start its computation without any further communication to the master or the trader.

If a slave receives an element stiffness matrix it can immediately start with the static condensation of local DOFs in order to compute the Schur complement. In case it received several Schur complements (as result from static condensation in the respective child-levels) it first has to assemble its local equation system $K \cdot u = d$. Therefore, all Schur complements K_i are build together as $K = \sum_i K_i$ by summing up corresponding matrix entries. Using several threads for the assembly step might entail heavy synchronisation in case parallelisation takes places over the K_i as no two threads are allowed to update the same matrix element k_{ij} in parallel. This can easily be solved deploying several critical sections (one per column, row or some block of K) or by using all threads for processing just one K_i instead. While the latter one comes without the need for synchronisation it entails lower parallelism due to the serial processing of all K_i . Concerning the static condensation of local DOFs, a partial Gaussian elimination is performed. Here, a multithreaded approach concerning the middle of the three nested loops from Gaussian elimination is advantageous, as threads profit

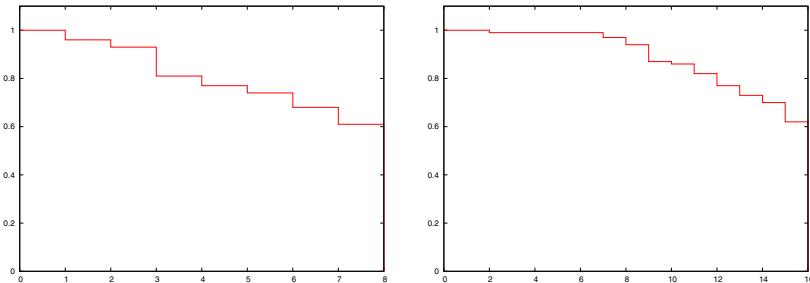


Fig. 2. Normalised working index for 8 and 16 slaves processing a problem with 4171 initial independent tasks on leaf level

both from the independent loop iterations and the shared memory. Obviously, this results in perfect, i. e. linear speed-up values on (SMP) architectures with 2, 4, and 8 cores as been tested. When finished, the slave contacts the same trader again to return its Schur complement for the next level tasks.

In the current stage, slaves are implemented as memoryless processes, deleting all subsequent information when the task has been finished. Keeping the regarding data even after the static condensation saves communication time in the final solution step, as these parts do not have to be transmitted again. Nevertheless, it implies a rigid order which slave has to process which task what might lead to bottlenecks in some cases.

A much more interesting questions is related to the working index of slaves. Slaves are either active, i. e. processing a task, or idle, i. e. waiting to be served. Summing up all phases a slave i is active and dividing it by the slave's total processing time results in a normalised working index $\omega_i \in [0, 1]$. Plotting all ω_i in decreasing order provides a step function that allows to estimate about the mean time slaves are busy. In the ideal case, all ω_i are close to 1, but due to the problem of decreasing activities (i. e. independent tasks) in each higher ND level, this will be the rare case. Figure 2 shows the working index for 8 and 16 slaves processing a problem with 4171 elements, i. e. initial independent tasks, served by 4 and 8 traders, resp. It can be observed, that the working indices vary between 0.6 and 1 which is a promising result especially when compared to the theoretical values according to Minsky et al. Nevertheless, only about half of the processes are active more than 90% of the total processing time, which is somewhat clear as the task size increases on each ND level when approaching the tree root and, thus, inactive processes need to wait longer before being served a new task. Here, further optimisation is possible if tasks larger than some threshold size are distributed among several processes in order to be processed in parallel on both process and block (via threads) level. This approach is closely coupled with the question at which point distributed parallel processing is more efficient than multithreading—a question that cannot be answered easily regarding latest trends in multi- and manycore architectures and which is still part of our researches.

5 Conclusion

In this paper, we have proposed a dynamic load balancing strategy for linear equation solvers based on the nested dissection approach in order to tackle known problems related to tree parallelisations. Applied to the p -version of finite element methods, first results sound very promising, bringing us one step closer to the long-term objective of structure dynamics in real time as needed for interactive computational steering scenarios.

Acknowledgements

Parts of this work have been carried out with the financial support of the International Graduate School of Science and Engineering (IGSSE) at Technische Universität München.

References

1. Düster, A., Bröker, H., Rank, E.: The p -version of the finite element method for three-dimensional curved thin walled structures. *Int. J. for Numer. Meth. in Eng.* 52, 673–703 (2001)
2. Düster, A., Scholz, D., Rank, E.: pq -Adaptive solid finite elements for three-dimensional plates and shells. *Comp. Meth. Appl. Mech. Eng.* 197, 243–254 (2007)
3. George, J.A.: Nested dissection of a regular finite element mesh. *SIAM J. on Numer. Analysis* 10, 345–363 (1973)
4. Minsky, M., Papert, S.: On some associative, parallel and analog computations. *Associative Information Technologies*. Elsevier/North Holland, Amsterdam (1971)
5. Mundani, R.-P.: Hierarchische Geometriemodelle zur Einbettung verteilter Simulationsaufgaben. Shaker Verlag (2006)
6. Mundani, R.-P., Bungartz, H.-J., Rank, E., Niggl, A., Romberg, R.: Extending the p -version of finite elements by an octree-based hierarchy. In: Proc. of the 16th Int. Conf. on Domain Decomp. Methods. LNCSE, vol. 55, pp. 699–706. Springer, Heidelberg (2006)
7. Rank, E., Rücker, M., Düster, A., Bröker, H.: The efficiency of the p -version finite element method in a distributed computing environment. *Int. J. for Numer. Meth. in Eng.* 52, 589–604 (2001)
8. Szabó, B.A., Babuška, I.: Finite Element Analysis. John Wiley & Sons, Chichester (1991)
9. Szabó, B.A., Düster, A., Rank, E.: The p -version of the Finite Element Method, ch. 5. *Encyclopedia of Computational Mechanics*, vol. 1, pp. 119–139. John Wiley & Sons, Chichester (2004)
10. Trummer, T.: Implementierung eines hochperformanten Lösungskonzeptes einer Simulations- und Steering-Umgebung im Bereich der Medizintechnik. Bachelor's thesis, Institut für Informatik, TU München (2008)

Simulation of Primary Breakup for Diesel Spray with Phase Transition

Peng Zeng¹, Samuel Sarholz², Christian Iwainsky², Bernd Binninger³,
Norbert Peters³, and Marcus Herrmann⁴

¹ Institut for Combustion Technology

RWTH Aachen University

Templergraben 64, 52056 Aachen, Germany

Tel.: +49-241 80-94622

Fax: +49-241 80-92923

p.zeng@itv.rwth-aachen.de

² Center for Computing and Communication

RWTH Aachen University

sarholz@rz.rwth-aachen.de

³ Institut for Combustion Technology

RWTH Aachen University

⁴ Department of Mechanical and Aerospace Engineering

Arizona State University

marcus.herrmann@asu.edu

Abstract. We perform direct numerical simulation on large distributed memory parallel computers in order to investigate the primary breakup process of diesel spray direct injection. Local refinement algorithm–Refined level-set method has been used to reduce the memory requirement, and we analyze the performance by experiments on a 1024-processor parallel computer.

Keywords: Two-phase flow, Liquid atomization, DNS.

1 Introduction

Numerical simulation of diesel engine combustion has become an important tool in engine development. One major issue in the modeling of turbulent reactive flows is the turbulent spray that accompanies fuel injection. The atomization of a liquid jet can be considered as two subsequent processes: Primary breakup, also called primary atomization, is the very first fragmentation process when the liquid column rushes out of a nozzle, forming ligaments and breaking up into primary droplets. Then, the spray will continue to break up into smaller droplets, this is called the secondary atomization process. The small droplets will evaporate, forming fuel-air mixture, ultimately, ignition will start combustion. Among all the physical models of the spray combustion process, the primary breakup of liquid jets in an initial dense region is mostly poorly understood due to its complex nature [7], as it involves a sudden jump in the density across the

gas-liquid interface, surface tension force on the interface, topological changes of the interface, and phase transition.

Modeling primary atomization has so far relied on analytical and empirical approaches. They predict mean drop sizes and global primary breakup rates, but mostly fail to capture the interaction of the phase interface with the turbulent flow field on the numerically resolved scale.

Different from traditional approaches, we consider using direct numerical simulation(DNS) combined with level-set method to resolve all time and length scales of primary atomization close to the injector [1]. By solving the governing equations in all details and capturing the gas-liquid interface accurately, we may provide new understanding towards the physics of primary atomization. The detailed information shall be used to derive a statistical model to simulate primary breakup for engineering application.

In the following sections, firstly, in section 2, the governing equations and the refined level-set grid method will be described. Then, the computational aspects concerning the performance of the numerical method will be discussed in section 4. In section 3, we will cover the direct numerical simulation of primary breakup with some interesting result.

2 Two-Phase Flow Using Level-Set Method

The two-phase flow is described in one-fluid formulation, liquid and vapor phases have their own fluid properties, i.e., density, viscosity, surface tension, etc. The flow is governed by the unsteady Navier-Stokes equations in the variable density incompressible limit[1,7],

$$\nabla \cdot \mathbf{u} = 0 \quad (1)$$

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} = -\frac{1}{\rho} \nabla p + \frac{1}{\rho} \nabla \cdot (\mu(\nabla \mathbf{u} + \nabla^T \mathbf{u})) + \mathbf{g} + \frac{1}{\rho} \mathbf{T}_\sigma \quad (2)$$

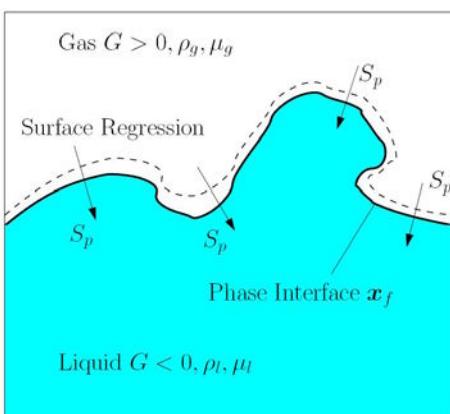


Fig. 1. G field

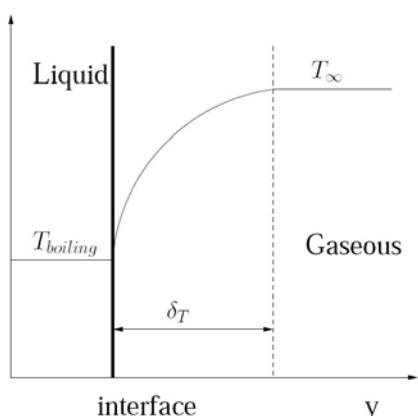


Fig. 2. Temperature Boundary Layer

Surface tension force \mathbf{T}_σ is non-zero only at the location of the phase interface \mathbf{x}_f

$$\mathbf{T}_\sigma(\mathbf{x}) = \sigma \kappa \delta(\mathbf{x} - \mathbf{x}_f) \mathbf{n} . \quad (3)$$

The phase interface location \mathbf{x}_f is described by a level-set scalar $G(\mathbf{x}_f, t) = 0$. In the gas, $G(\mathbf{x}_f, t) < 0$; in the liquid, $G(\mathbf{x}_f, t) > 0$. The surface regression velocity S_p shown in Fig. 1 due to evaporation, leads to the interface evolution equation as

$$\frac{\partial G}{\partial t} + \mathbf{u} \cdot \nabla G + S_p |\nabla G| = 0 . \quad (4)$$

Starting from the balance of energy, we assume all the conducted heat is consumed by evaporation,

$$\frac{\rho_g \nu_g}{Pr} \frac{\partial T}{\partial y} = \frac{\dot{m} h_L}{C_p} , \quad (5)$$

where $\dot{m} = \rho_l S_p$ is the mass flow rate per unit area, h_L is the latent heat of phase transition, C_p is the heat capacity of liquid phase, and Pr is the Prandtl number. Fig. 2 shows the temperature boundary layer, where δ_T is the boundary layer thickness which includes the length scale. In laminar cases, the surface regression velocity can therefore be modeled as

$$S_p = \frac{1}{Pr} \frac{\rho_g}{\rho_l} \frac{C_p(T_\infty - T_{Boiling})}{h_L} \frac{\nu_g}{\delta_T} . \quad (6)$$

3 Numerical Methods

The primary breakup process contains a large range of physical length scales, ranging from the size of the jet or sheet down to the size of the tiny drops that can be initially ripped out of the phase interface. Adaptive mesh refinement techniques are essential to ensure correct DNS results, using available computing resources. The interface evolution equation (4) is solved by using Refined Level-Set Grid(RLSG) method on an auxiliary, high-resolution equidistant Cartesian grid(see Fig. 3), while the Navier-Stokes equations (1)(2) are solved on their own

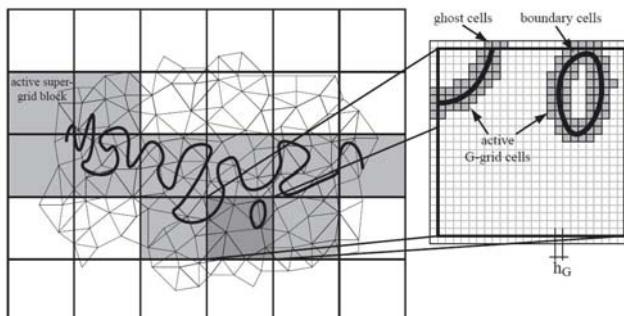


Fig. 3. Refined Level Set Grid method

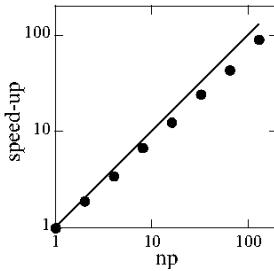


Fig. 4. Speedup of RLSG method [2]

computational grid. The remaining variables are expressed in terms of function based on the instantaneous position of the liquid-vapor interface. The RLSG solver LIT (Level set Interface Tracker) uses 5th order WENO scheme for space and 3rd order Runge-Kutta scheme for time discretization. The Navier-Stokes equations are spatially discretized using low-dissipation, finite-volume operators [3]. The flow solver CDP uses a fully unstructured computational grid. A low-dissipation, finite-volume operators [3] spatially discretized the Navier-Stokes equations. CDP uses a second order Crank-Nicolson scheme for implicit time integration, and the fractional step method will remove the implicit pressure dependence in the momentum equations. Communication between the level-set solver and the flow solver is handled by the coupling software CHIMPS [4]. Domain decomposition parallelization of RLSG method has been achieved by the two-level narrow band methodology. On the first level, the super-grid cells, or blocks, that contain part of the interface are activated and stored in linked lists distributed among the distributed processors (see Fig. 3). A lookup table stores the i, j, k coordinates of the super-grid coordinates enable fast access of them. On the second level, each active block contains an equidistant Cartesian grid of local refined grid size h_G (see Fig. 3). Again, only those grid cells that contain part of the interface are activated and stored. This approach effectively reduces the number of cells that are stored and on which the level-set equations have to be solved from $O(N^3)$ to $O(N^2)$. Moreover, the dynamic load balancing and direct access to random nodes are straightforward, so high level of parallel efficiency and scalability can be achieved (see Fig.4).

4 Computational Aspects

The simulation has been computed on an Intel Xeon-based Linux cluster at the Center for Computing and Communication of the RWTH Aachen University. The 128 nodes are equipped with two 3 Ghz Xeon E5450 quad-core processors and 16 GB of memory and are connected with 4x DDR Infiniband.

The dataset uses approximately 100 GB of memory, which is evenly split between the MPI processes. Overall 6000 time steps were computed with 512 processes on 128 nodes using a wall-clock time of 51 hours. Lacking a serial run for the large data set, we approximate a speedup of 350.

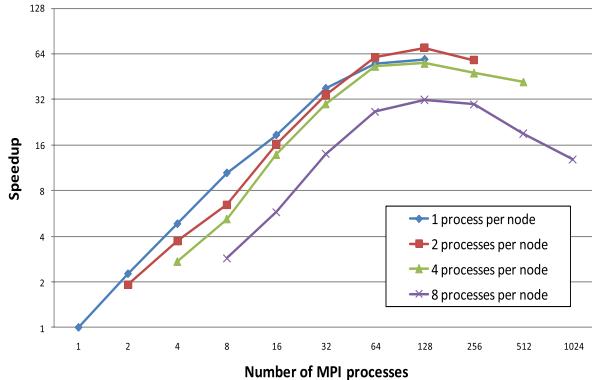


Fig. 5. Scalability graph, reduced data set

We concluded that the original dataset was too large to be usable for speedup measurements due to memory restrictions. Therefore we scaled the data set down by a factor of 8 and computed only two iterations.

With measurement we determined that the optimal performance (70.2 speedup) was reached with 128 MPI processes, see figure 5. For small MPI process numbers, in this case 32, we observed superlinear speedup as a benefit of accumulated cache size. Evaluation of the communication behavior with Vampir [8] showed, that the increasing complexity of the mainly used allreduce dominates the execution time for more than 128 processes.

Variation of the number of processes per node did not change the behavior of the scalability. The optimal number of processes per node is about 2 or 4, as the memory bandwidth of a single node reaches saturation for this.

We can only partially measure the scalability of the large data set. Therefore we compared samples of the speedup curve for the large dataset starting at 32 MPI processes through 512 with the smaller one. We observe that the sampled area shows same characteristics as the scaled version shifted by a factor of 8. Extrapolating from the available speedup curves we predict that the large data set will reach its lowest execution time with 1024 MPI processes running on 512 nodes.

5 DNS Results of Primary Breakup

The simulations use $256 \times 256 \times 512$ grid points in radial, azimuthal and axial directions for the flow solver, and the mesh is stretched in order to cluster grid points near the spray center, spacing the finest grid $\Delta x \simeq 3\eta \sim 4\eta$, where $\eta \simeq 1\mu m$ is the Kolmogorov length coherent to the Reynolds number given before. The refined level-set grid has a half billion active cells. This combination was shown to yield promising results for primary breakup [5] [4].

Fig. 6 shows snapshots of the turbulent liquid jet and droplets generated by primary breakup. The Lagrangian spray model, which removes the droplets from

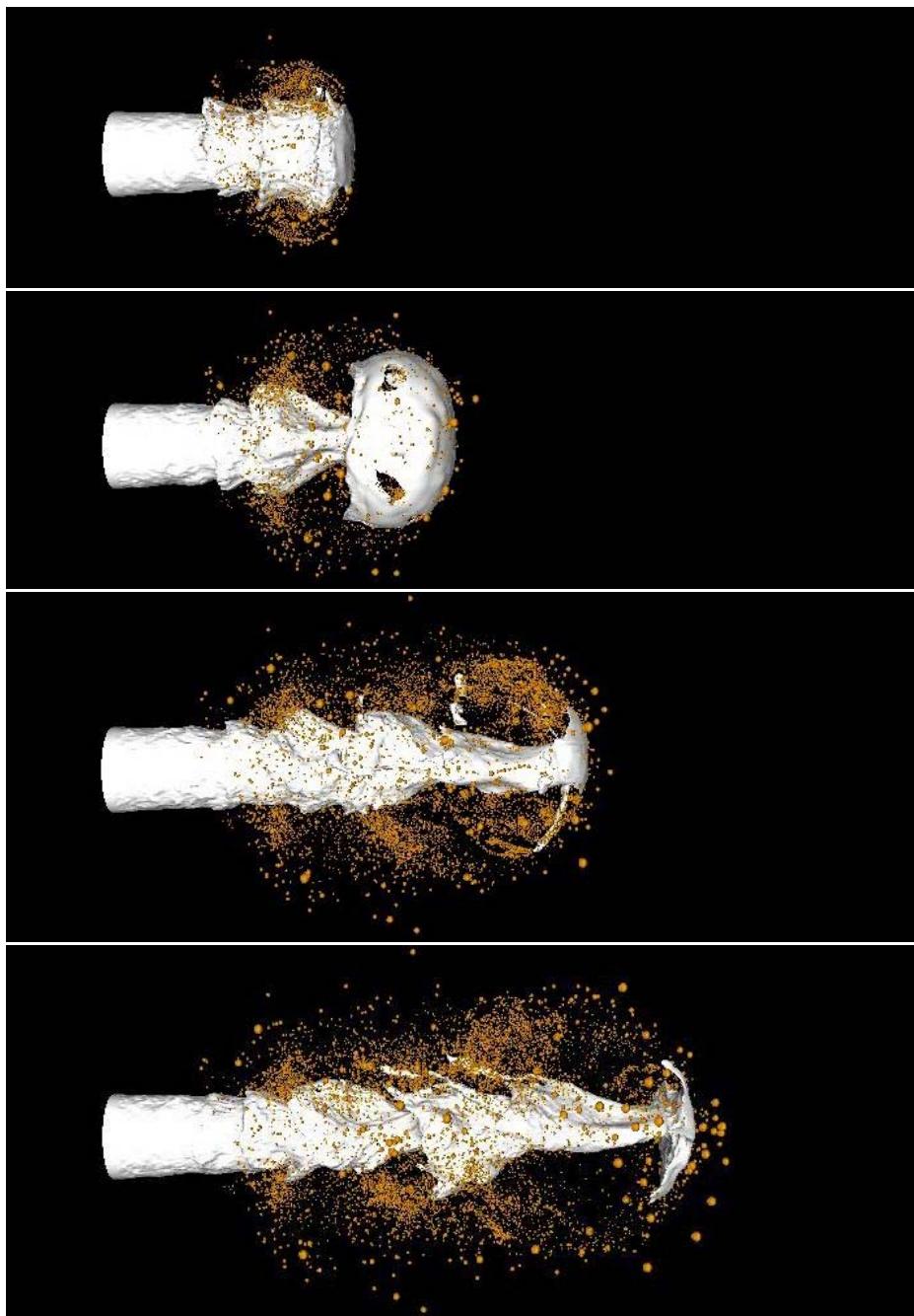


Fig. 6. Four successive snapshots of primary atomization, from top to bottom, $t = 4\mu s$, $t = 6\mu s$, $t = 8\mu s$, $t = 10\mu s$

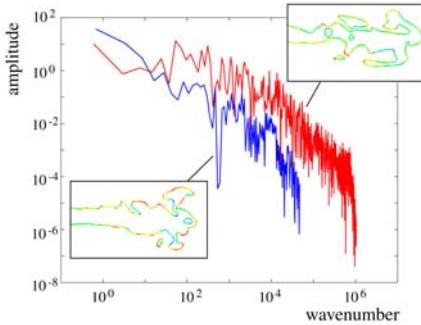


Fig. 7. Curvature Spectrum, with evaporation(red), without(blue)

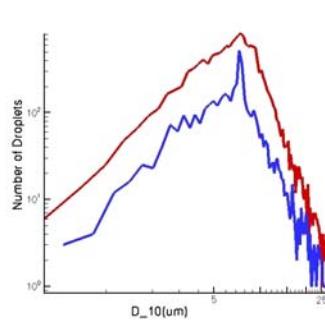


Fig. 8. Droplet size distribution, with evaporation(red), without(blue)

the ligaments and transfers into Lagrangian particles, can be found in [5]. Most of the droplets come from the mushroom tip at the jet head, complex topology and elongated ligaments have been observed. Compared with the atomization process without evaporation, the breakup of ligaments and droplet generation are much faster and more intensive. This can be explained in Fig. 7, which shows the curvature spectrum made from Fourier transformation of local curvature values along the ligaments with- and without evaporation. In the evaporation case, the large wave number of curvature fluctuations will promote the breakup processes.

Fig. 8 shows the droplet size distribution, ranging from the cut-off length scale that accompanies with the numerical grid size to large liquid blocks. Different from the atomization process without evaporation, more small droplets can be observed. Mesh convergence has not been performed yet, this is a first step in a series of calculations, where the focus is on the evaporation effect on spray primary breakup.

6 Summary and Outlook

The refined level-set grid method for primary breakup with phase transition has been presented. The speedup performance analysis shows that for a constant number of processors best efficiencies are reached when the problem size is as large as possible. However, for every problem with a specific size, there is an optimal number of processors to use. The scalability of our code for the current case was found satisfactory, as we can compute in 4 days for a typical run. The two phase flow solver has been applied on direct numerical simulation of a turbulent diesel injection, although there are many numerical uncertainties, preliminary results show promising direction towards further understanding of the physical process of atomization with evaporation effect. The mathematical model and the DNS solution presented here will provide the frame for a statistical simulation of the primary breakup, within the large eddy simulation (LES) will be done in the future. Investigation is necessary to reduce the MPI overhead

and improve performance in order to be able to couple our code with secondary breakup models, and ultimately towards spray combustion simulation.

Acknowledgments

This work is financed by the German Research Foundation in the framework of DFG-CNRS research unit 563: Micro-Macro Modelling and Simulation of Liquid-Vapour Flows, (DFG reference No. Pe241/35-1).

References

1. Gorokhovski, M., Herrmann, M.: Modeling Primary Atomization. *Annual Review of Fluid Mechanics* 40, 343–366 (2008)
2. Herrmann, M.: A balanced force Refined Level Set Grid method for two- phase flows on unstructured flow solver grids. *J. Comput. Phys.* 227, 2674–2706 (2008)
3. Ham, F., Mattsson, K., Iaccarino, G.: Accurate and stable finite volume operators for unstructured flow solvers. *Center for Turbulence Research Annual Research Briefs* (2006)
4. Kim, D., Desjardins, O., Herrmann, M., Moin, P.: The Primary Breakup of a Round Liquid Jet by a Coaxial Flow of Gas. In: *Proceedings of the 20th Annual Conference of ILASS Americas* (2007)
5. Herrmann, M.: Detailed Numerical Simulations of the Primary Breakup of Turbulent Liquid Jets. In: *Proceedings of the 21st Annual Conference of ILASS Americas* (2008)
6. Spiekermann, P., Jerzembeck, S., Felsch, C., Vogel, S., Gauding, M., Peters, N.: Experimental Data and Numerical Simulation of Common-Rail Ethanol Sprays at Diesel Engine-Like Conditions. *Atomization and Sprays* 19, 357–387 (2009)
7. Baumgarten, C.: *Mixture Formation in Internal Combustion Engines*. Springer, Heidelberg (2006)
8. Knüpfer, A., Brunst, H., Doleschal, J., Jurenz, M., Lieber, M., Mickler, H., Müller, M.S., Nagel, W.E.: The Vampir Performance Analysis Tool-Set. In: *Proceedings of the 2nd HLRS Parallel Tools Workshop* (2008)

Implementing Reliable Data Structures for MPI Services in High Component Count Systems

Justin M. Wozniak, Bryan Jacobs, Robert Latham,
Sam Lang, Seung Woo Son, and Robert Ross

Mathematics and Computer Science Division,
Argonne National Laboratory, Argonne, IL 60439, USA

{wozniak,robl,slang,sson,rross}@mcs.anl.gov, bryan.jacobs@rochester.edu

High performance computing systems continue to grow: currently deployed systems exceed 160,000 cores and systems exceeding 1,000,000 cores are planned. Without significant improvements in component reliability, partial system failure modes could become an unacceptably regular occurrence, limiting the usability of advanced computing infrastructures. In this work, we intend to ease the development of survivable systems and applications through the implementation of a reliable key/value data store based on a distributed hash table (DHT). Borrowing from techniques developed for unreliable wide-area systems, we implemented a distributed data service built with MPI [1] that enables user data structures to survive partial system failure. The service is based on a new implementation of the Kademia [2] distributed hash table.

Small faults should have small impacts on applications, resulting in proportionate data movement costs to restore redundancy schemes to a nominal state and normalize the distributed data structures that enable the efficient management of the system. In a DHT-based system, a fault of unit size impacts a small number of cooperating processors, which provides the primary motivation for this work. Using such an implementation effectively in a high performance setting faces many challenges, including maintaining good performance, offering wide compatibility with diverse architectures, and handling multiple fault modes. The implementation described here employs a layered software design built on MPI functionality and offers a scalable data store that is fault tolerant to the extent of the capability of the MPI implementation.

DHT implementations offer a simple equivalent to the well-known hash table: *put(key, value)* to store a key/value pair and *get(key) → value* to retrieve a value. Peer-to-peer DHTs decentralize the organization of the system, requiring that each peer node be capable of participating in the self-organization of the system, enabling robust reassembly in the event of node failure or departure from the system. Additionally, each node is limited to an asymptotically small amount of information about the rest of the system.

Our implementation enables the storage of critical application data in a distributed key/value database, achieving non-volatility through redundancy. We demonstrate that the data structure offers good asymptotic performance at large scale in terms of latency and memory usage and can endure multiple faults. The system can be used in various ways, including as a temporary in-core cache on a compute cluster or as a stand-alone distributed database.

Background

The fault handling characteristics of MPI have been extensively considered in previous work [3, 4, 5, 6]. In this work, we are interested in properly using the fault handling capabilities offered by the underlying infrastructure - the MPI implementation - and in providing reliability to a higher level application.

A reliable numerical programming use case with FT-MPI [7] demonstrates how scientific data structures can achieve non-volatility through in-core parity-based checkpointing, reducing the need for checkpoint operations to disk. Other reliable MPI systems offer implementations that help the user checkpoint computational and messaging state. A semi-automatic system, the Cornell Checkpoint (pre-)Compiler (C^3) [8], enables the use of preprocessor directives that direct checkpointing. MPICH-V [9] provides multiple techniques to log messages, operating at a low level of the MPI implementation stack, allowing for comparison among multiple fault tolerance protocols.

Our project differs from these systems in multiple ways. First, we intend to provide a fault tolerant user data store, and not preserve the state of a computation or in-flight messages. Second, our implementation takes the form of a data service and associated library, not an MPI implementation. Since we use an MPI implementation to provide these services, we could potentially use a checkpoint-based fault-tolerant MPI implementation; however, the DHT provides its own redundancy and other fault tolerant characteristics. Hence, we require significantly less from the MPI implementation than is provided by the solutions above.

References

1. The MPI Forum: MPI-2: Extensions to the message-passing interface (1997)
2. Maymounkov, P., Mazières, D.: Kademia: A peer-to-peer information system based on the XOR metric. In: Druschel, P., Kaashoek, M.F., Rowstron, A. (eds.) IPTPS 2002. LNCS, vol. 2429, pp. 53–65. Springer, Heidelberg (2002)
3. Lu, C.-d., Reed, D.A.: Assessing fault sensitivity in MPI applications. In: Proc. SC 2004 (2004)
4. Gropp, W., Lusk, E.: Fault tolerance in MPI programs. J. High Performance Computing Applications 18(3) (2004)
5. Latham, R., Ross, R., Thakur, R.: Can MPI be used for persistent parallel services? In: Mohr, B., Träff, J.L., Worringer, J., Dongarra, J. (eds.) PVM/MPI 2006. LNCS, vol. 4192, pp. 275–284. Springer, Heidelberg (2006)
6. Thakur, R., Gropp, W.: Open issues in MPI implementation. In: Proc. Asia-Pacific Computer Systems Architecture Conference (2007)
7. Chen, Z., Fagg, G.E., Gabriel, E., Langou, J., Angskun, T., Bosilca, G., Dongarra, J.: Fault tolerant high performance computing by a coding approach. In: Proc. Symposium on Principles and Practice of Parallel Programming (2005)
8. Schulz, M., Bronevetsky, G., Fernandes, R., Marques, D., Pingali, K., Stodghill, P.: Implementation and evaluation of a scalable application-level checkpoint-recovery scheme for MPI programs. In: Proc. SC 2004 (2004)
9. Bouteiller, A., Herault, T., Krawezik, G., Lemarinier, P., Cappello, F.: MPICH-V Project: A multiprotocol automatic fault-tolerant MPI. J. High Performance Computing Applications 20(3) (2006)

Parallel Dynamic Data Driven Genetic Algorithm for Forest Fire Prediction*

Mónica Denham, Ana Cortés, and Tomás Margalef

Departament d' Arquitectura de Computadors i Sistemes Operatius, E.T.S.E.,
Universitat Autònoma de Barcelona, 08193 - Bellaterra (Barcelona) Spain
`monica@caos.uab.es, {ana.cortes,tomas.margalef}@uab.es`

1 Forest Fire Spread Prediction

Forest fire simulators are a very useful tool for predicting fire behavior. A forest fire simulator needs to be fed with data related to the environment where fire occurs: terrain main features, weather conditions, fuel type, fuel load and fuel moistures, wind conditions, etc. However, it is very difficult to obtain the real values of these parameters during a disaster [1]. The lack of accuracy of the input parameter values adds uncertainty to any prediction method and it usually provokes low quality simulations.

In order to achieve high quality simulations, we use a two stages prediction method, which is depicted in figure 1 (a) [1]. During the calibration stage we use a Dynamic Data Driven Genetic Algorithm (DDDGA) in order to improve simulator input values. The best set of input values found during the calibration stage is used in the prediction stage in order to better predict the fire progress.

Thus, through calibration stage a search in a very big search space (formed by all possible input values combinations) is performed. This search is solved by a parallel implementation of the Genetic Algorithm. During the calibration stage several independent simulations are performed so, we apply a master/worker programming paradigm as a parallel scheme (see figure 1 (b)). Furthermore, the observed real fire propagation is studied and the knowledge obtained is injected during this stage in order to steer the DDDGA. In particular, two steering methods were proposed and applied [2]. We have also analyzed in detail the task execution time [4] and the communication pattern of the whole application to determine workload, message passing patterns [3], application granularity, etc., avoiding parallel computing possible penalties.

2 Experimental Results and Conclusions

Application scalability. Figure 2 shows that the proposed parallel Dynamic Data Driven Genetic Algorithm exhibits a good scalability as the number of worker processes increases. Moreover, computational steering method and non guided GA have a similar behavior, concluding that the inclusion of a steering strategy will not affect the scalability of the whole system.

* This work has been supported by the MEC-Spain under contracts TIN 2007-64974.

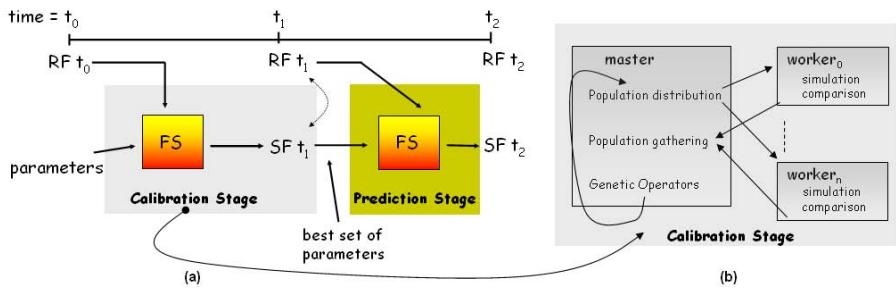


Fig. 1. (a) DDDAS for forest fire prediction (b) Master/worker application for calibration stage

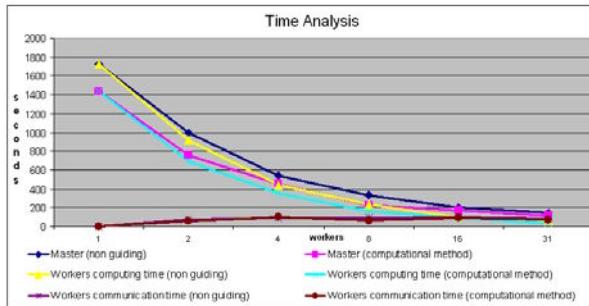


Fig. 2. Computing and communication time comparison for different number of workers

Workload balance analysis. Analyzing in more detail the execution time for each individual worker, we observed that there were no significant time differences among them, what confirms that the proposed task scheduling strategy works properly for this kind of application. Although the experiment was performed using a homogeneous cluster, our workload distribution method will be also useful when heterogeneous nodes are used.

Communication Time. As we can observe in figure 2, the worker's communication times keep lower bounded, for different steering methods and executions, which mainly means that master/worker communication does not represent a bottleneck for this kind of application.

References

1. Bianchini, G.: Wildland Fire Prediction based on Statistical Analysis of Multiple Solutions. Ph. D Thesis. Universitat Autònoma de Barcelona (Spain) (July 2006)
2. Denham, M., Cortés, A., Margalef, T.: Computational Steering Strategy to Calibrate Input Variables in a Dynamic Data Driven Genetic Algorithm for Forest Fire Spread Prediction. In: Allen, G., et al. (eds.) ICCS 2009, Part II. LNCS, vol. 5545, pp. 479–488. Springer, Heidelberg (2009)
3. Foster, I.: Designing and Building Parallel Programs (Online). In: Message Passing Interface, ch. 8. Addison-Wesley, Reading (1995), <http://www-unix.mcs.anl.gov/dbpp/text/node94.html>
4. MPE_Open_Graphics, <http://www-unix.mcs.anl.gov> (Accessed on September 2008)

Hierarchical Collectives in MPICH2

Hao Zhu¹, David Goodell², William Gropp¹, and Rajeev Thakur²

¹ Department of Computer Science,

University of Illinois, Urbana, IL, 61801, USA

² Mathematics and Computer Science Division,

Argonne National Laboratory, Argonne, IL 60439, USA

Abstract. Most parallel systems on which MPI is used are now hierarchical, such as systems with SMP nodes. Many papers have shown algorithms that exploit shared memory to optimize collective operations to good effect. But how much of the performance benefit comes from tailoring the algorithm to the hierarchical topology of the system? We describe an implementation of many of the MPI collectives based entirely on message-passing primitives that exploits the two-level hierarchy. Our results show that exploiting shared memory directly usually gives small additional benefit and suggests design approaches for where the benefit is large.

Keywords: MPI, Collective Communication.

Most modern parallel computing systems have a hierarchical structure, often with several levels of hierarchy. For years, these systems have been composed of a (possibly hierarchical) high-speed network connecting many symmetric multi-processor (SMP) nodes with a handful of processor cores on each node. While node counts will likely increase in many HPC systems, core counts per node are expected to increase dramatically [4] in the coming years.

Much work has been done in the areas of both hierarchical and shared memory collective algorithms. Hierarchical algorithms are discussed in the context of wide-area network (WAN) MPI implementations in [1,2,3]. The WAN/LAN collectives scenario is analogous to the network/shared memory scenario discussed in this paper. Of particular note, [2] provides a flexible parameterized LogP model for analyzing collective communication in systems with more than one level of network hierarchy.

We implement hierarchical algorithms for several collectives. Our hierarchical algorithms for broadcast, reduce, allreduce and barrier have similar structures:

1. If necessary, perform local node operation and collect data in the master process of each node, such as in broadcast and barrier. Here master process means the representative of the node, which is the only process participating inter-node communication. We select the master processes as the lowest ranked process on each node in the communicator passed to the collective.
2. Perform inter-node operation among all the master processes. Send/collect data from/to the root in rooted collectives, or send/collect data in all the master processes.

3. If necessary, perform local node operation such as broadcast data received in step 2 from master process to other processes in the node.

We can easily implement hierarchical algorithms for many collectives, because non-hierarchical collectives can be used in for both intra-node and inter-node phases of communication. A few operations, such as scan, are more difficult. We have a special algorithm for scan, which will not be explained here.

For all of the collective algorithms discussed here, the inter-node communication time dominates the overall communication time. By measuring the time spent in the inter-node phase of the collective and assuming an ideal (zero time) intra-node phase we can use Amdahl's Law to obtain a lower-bound for total collective time when using a shared-memory collective algorithm.

Our results show that significant performance benefits can be realized by exploiting the hierarchical nature of the interconnect topology. For many of the collective routines, particularly those that do not make many copies of the same data (which Beast does), our performance models suggest that for the long-message case, directly using shared memory will not significantly improve performance beyond what exploiting the topology achieves.

For short messages, where overheads dominate, using shared memory may offer a relatively greater advantage. This suggests an implementation strategy that uses small amounts of shared memory for short data transfers within an SMP node for collectives, switching to message-passing for longer data transfers.

References

1. Karonis, N.T., de Supinski, B.R., Foster, I., Gropp, W., Lusk, E., Bresnahan, J.: Exploiting hierarchy in parallel computer networks to optimize collective operation performance. In: Fourteenth International Parallel and Distributed Processing Symposium, May 2000, pp. 377–384 (2000)
2. Kielmann, T., Bal, H.E., Gorlatch, S., Verstoep, K., Hofman, R.F.H.: Network performance-aware collective communication for clustered wide-area systems. *Parallel Computing* 27(11), 1431–1456 (2001)
3. Kielmann, T., Hofman, R.F.H., Bal, H.E., Plaat, A., Bhoedjang, R.A.F.: MagPIe: MPI's collective communication operations for clustered wide area systems. *SIGPLAN Not.* 34(8), 131–140 (1999)
4. Vangal, S.R., Howard, J., Ruhl, G., Dighe, S., Wilson, H., Tschanz, J., Finan, D., Singh, A., Jacob, T., Jain, S., Erraguntla, V., Roberts, C., Hoskote, Y., Borkar, N., Borkar, S.: An 80-tile sub-100-w TeraFLOPS processor in 65-nm CMOS. *IEEE Journal of Solid-State Circuits* 43(1), 29–41 (2008)

An MPI-1 Compliant Thread-Based Implementation

J.C. Díaz Martín, J.A. Rico Gallego, J.M. Álvarez Llorente,
and J.F. Perogil Duque

Escuela Politécnica, University of Extremadura
Avda. de la Universidad s/n, 10071, Cáceres, Spain
`{juancarl,jarico,llorente,fperduq}@unex.es`
<http://gim.unex.es>

Abstract. This work presents AzequiaMPI, the first full compliant implementation of the MPI-1 standard where the MPI node is a thread. Performance comparisons with MPICH2-Nemesis show that thread-based implementations exploit adequately the multicore architectures under oversubscription, what could make MPI competitive with OpenMP-like solutions.

Keywords: Thread-based MPI implementation, multicore architectures.

1 Introduction and Design

AzequiaMPI is a thread-based full conformant implementation of the MPI-1.3 standard. The source code is available with open source license ([1]). Azequia is a layer that provides point-to-point communication and application deployment on the cluster by lock-based communications upon Pthreads. AzequiaMPI implements the MPI interface and semantics upon Azequia. Targeted to embedded systems, AzequiaMPI runs on heterogeneous distributed embedded signal processing platforms of DSP and FPGAs ([4]). Linux clusters of SMPs are also supported.

2 Performance

AzequiaMPI consumes about a quarter of the MPICH2-Nemesis memory, an important feature for embedded systems. We conducted a ping-pong microbenchmark (Figure 1) to compare thread-based implementations versus MPICH2-Nemesis on a Intel Clovertown chip (eight 2.66 GHz processors) on shared memory (results are virtually the same in a Gigabit Ethernet distributed setup). Two processors are involved, one holding the sender and another the receiver. MPICH2-Nemesis gives by large the best performance for short messages in this dedicated environment, mainly due to the latency introduced by synchronisation in the thread-based case. AzequiaMPI behaviour is quite similar to TOMPI

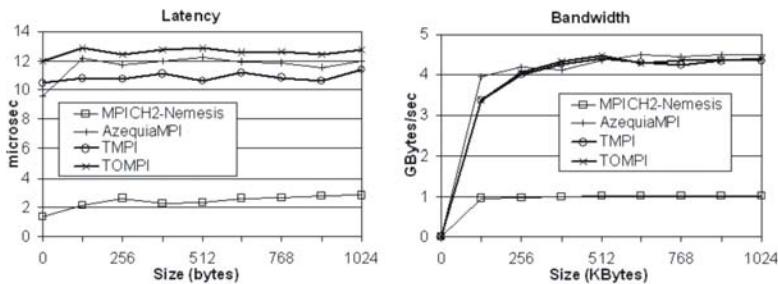


Fig. 1. Performance of thread-based versus process-based MPI on shared memory

([2]) and TMPI ([3]), in spite of the former supports the whole MPI-1 standard. MPICH2-Nemesis performance, however, degrades for big messages and, although not shown here, degrades much more under oversubscribing. It is true that a process-based implementation needs two copies for sending a message and a thread-based implementations just one. This argument, however, does not justify by itself the cute fall of MPICH2-Nemesis.

3 Conclusions and Further Work

We have presented the first thread-based full implementation of the MPI-1.3 standard. Tests against MPICH2-Nemesis demonstrate that thread-based implementations exploit adequately the multicore architectures under oversubscription, throwing a communication bandwidth that may make MPI competitive with OpenMP-like solutions. This fact takes to a single programming paradigm for the whole application, MPI, what would represent an enormous advantage from the software engineering point of view. We plan to make performance comparisons with OpenMP and improving the performance for short messages, as well as to initiate the MPI-2 extension.

Acknowledgments. This work has been supported by Spanish Government CDTI under national research program “Ingenio 2010, subprogram CENIT-2005”.

References

1. <http://gim.unex.es/azequia>, <http://gim.unex.es/azequiampi>
2. Demaine, E.D.: A threads-only MPI implementation for the development of parallel programs. In: Proc. of the 11th Int. Symposium on High Performance Computing Systems, July 1997, pp. 153–163 (1997)
3. Tang, H., Shen, K., Yang, T.: Program Transformation and Runtime Support for Threaded MPI Execution on Shared Memory Machines. ACM Transactions on Programming Languages and Systems 22(4), 673–700 (2000)
4. <http://www.sundance.com>

Static-Analysis Assisted Dynamic Verification of MPI Waitany Programs (Poster Abstract)^{*}

Sarvani Vakkalanka¹, Grzegorz Szwedzka¹, Anh Vo¹,
Ganesh Gopalakrishnan¹, Robert M. Kirby¹, and Rajeev Thakur²

¹ School of Computing, Univ. of Utah, Salt Lake City, UT 84112, USA

² Math. and Comp. Sci. Div., Argonne Nat. Lab., Argonne, IL 60439, USA

Overview: It is well known that the number of schedules (interleavings) of a concurrent program grows exponentially with the number of processes. Our previous work has demonstrated the advantages of an MPI-specific dynamic partial order reduction (DPOR, [5]) algorithm called POE in a tool called ISP [1,2,3] in dramatically reducing the number of interleavings during formal dynamic verification. Higher degrees of interleaving reduction were achieved when the programs were *deterministic*. In this work, we consider the problem of verifying MPI using **MPI_Waitany** (and related operations wait/test some/all). Such programs potentially have a higher degree of non-determinism. For such programs, POE can become ineffective, as shown momentarily. To solve this problem, we employ static analysis (supported by ROSE [4]) in a supporting role to POE to determine the extent to which the *out* parameters of **MPI_Waitany** can affect subsequent control flow statements. This informs ISP's scheduler to exert even more intelligent backtrack/replay control.

Illustration: Consider the MPI example shown in Figure 1(a). The MPI code for rank 0 (lines 1 – 12) issues two **MPI_Irecv**s (lines 2 – 3) and later invokes a **MPI_Waitany** (line 5) on the request handles returned by the **MPI_Irecv**s. The code in lines 6 – 10 will execute along code paths determined by the value of the **done** variable (recall that this variable represents the index into the request array returned by **MPI_Waitany**). For sound verification, ISP must ensure that a sufficient number of values of the **done** variable are returned. In this example, **done = 0** and **done = 1** are both feasible, since both the posted **Irecv** have matching sends available. Thus, the previous POE algorithm and the proposed one will both execute two interleavings, one for each value of **done**.

Now consider the MPI program in Figure 1(b) – a slight modification of that in Figure 1(a). The MPI program issues two **MPI_Waitany**s in a for loop (lines 5–6). Our previous version of the POE algorithm would examine four interleavings (two interleavings for the first loop iteration, and two more interleavings under that for the second loop iteration). Under the new algorithm, this program would result in only two interleavings to accommodate the use of the received data. Since the **done** variable itself is not used, we will not need to create separate interleavings to account for possible control-flow changes based on it. Now, if

* Supported in part by Microsoft, NSF CNS-0509379, CCF-0811429, and the Mathematical, Information, and Computational Science Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

```

1: if (rank == 0) {
2:   Irecv (from 1, &req[0]);
3:   Irecv (from 2, &req[1]);
4:
5:   Waitany (2, req, &done,
6:             &status);
7:   if (done == 0) {
8:     ... // More MPI code
9:   } else if (done == 1) {
10:    ... // More MPI code
11:  }
12:  if (rank == 1) {
13:    Isend( to 0, &req[0])
14:    Wait(&req[0]);
15:  }
16:  if (rank == 2) {
17:    Isend( to 0, &req[0])
18:    Wait(&req[0]);
}

```

(a)

```

1: if (rank == 0) {
2:   Irecv (from 1, &req[0]);
3:   Irecv (from 2, &req[1]);
4:   ... use the received data ...
5:   for (i = 0; i < 2 ; i++)
6:     Waitany (2, req, &done,
7:               &status);
8:   }
9:   if (rank == 1) {
10:    Isend( to 0, &req[0])
11:    Wait(&req[0]);
12:   }
13:   if (rank == 2) {
14:    Isend( to 0, &req[0])
15:    Wait(&req[0]);
}

```

(b)

Fig. 1. (a) Waitany example with conditionals (b) Example without conditionals

rank 0 posted yet another `Irecv` with argument ‘`from 3`’ and the `for` loop is iterated three times, this would still result in three total interleavings for the entire loop, whereas the previous POE algorithm would have examined 27 interleavings. With respect to the previous POE, MPI_Waitsome is worse, as each invocation of Waitsome results in 2^{count} interleavings where $count$ is the number of requests passed to `MPI_Waitsome`. However, observe that the MPI program in Figure 1(b) has no conditional paths based on the value returned in the `done` variable; therefore even for this example, the new ISP tool would examine just one interleaving per iteration.

While we did not encounter such interleaving explosions in practice yet, the very purpose of a tool such as ISP is to be prepared in case these cases do arise. The new algorithm does not introduce any overheads barring static analysis time, which is negligible. A limitation is in dealing with MPI programs where the sources are unavailable.

References

1. http://www.cs.utah.edu/formal_verification/ISP-release/
2. Vakkalanka, S., Gopalakrishnan, G.C., Kirby, R.M.: Dynamic verification of MPI programs. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 66–79. Springer, Heidelberg (2008)
3. Vo, A., Vakkalanka, S., DeLisi, M., Gopalakrishnan, G., Kirby, R.M., Thakur, R.: Formal verification of practical MPI programs. In: PPoPP 2009, pp. 261–270 (2009)
4. <http://rosecompiler.org>
5. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: POPL, pp. 110–121. ACM, New York (2005)

Author Index

- Aananthakrishnan, Sriram 261
Afsahi, Ahmad 250
Aguilar-Cornejo, M. 294
Almeida, Francisco 185
Álvarez Llorente, J.M. 327
Anand, Rakhi 124

Bader, Michael 292
Balaji, Pavan 20
Binninger, Bernd 313
Blas, Javier García 164
Buettner, David 134
Buntinas, Darius 20
Burtscher, Martin 1

Calderón, Alejandro 207
Carretero, Jesús 164, 207
Carriault, Patrick 94
Castro-García, M.A. 294
Clet-Ortega, Jérôme 104
Cordero-Sánchez, S. 294
Cortés, Ana 323

DeLisi, Michael 261
Denham, Mónica 323
Díaz Martín, J.C. 327
Doallo, Ramón 174
Dongarra, Jack 240
Dorta, Antonio J. 185
Duarte, Angelo 73
Düster, Alexander 305

Fahringer, Thomas 196
Fialho, Leonardo 73
Filgueira, Rosa 207
Fraguela, Basilio B. 174

Gabriel, Edgar 124
Gebser, Martin 64
Geimer, Markus 31
Goodell, David 20, 325
Gómez, Andrés 174
Gopalakrishnan, Ganesh 8, 261, 271, 329
Gottbrath, Chris 282

Graham, Richard L. 2, 116
Gropp, William 3, 20, 42, 325

Hermanns, Marc-André 31
Herrmann, Marcus 313
Hoefler, Torsten 240
Hofmann, Michael 54
Hori, Atsushi 9

Isailă, Florin 164
Ishikawa, Yutaka 9
Iwainsky, Christian 313

Jacobs, Bryan 321
Jourdren, Hervé 94

Kaminski, Roland 64
Kamoshida, Yoshikazu 9
Kaufmann, Benjamin 64
Keller, Rainer 116
Kirby, Robert M. 8, 261, 271, 329
Knežević, Jovana 305
Kumar, Sameer 20
Kumon, Kouichi 9
Kunkel, Julian 134

Lang, Sam 321
Larsson, Jesper 20
Lastovetsky, Alexey 4
Latham, Robert 42, 164, 321
LeBlanc, Troy 124
Li, Ang 84
Liu, Xiaoguang 84
Ludwig, Thomas 134
Lumsdaine, Andrew 240
Luque, Emilio 73
Lusk, Ewing 6, 20, 42

Mallón, Damián A. 174
Margalef, Tomás 323
Matsuba, Hiroya 9
Mercier, Guillaume 104
Mir, Faisal Ghias 154
Mohr, Bernd 31
Moreno-Montiel, C.H. 294

- Moritsch, Hans 196
Mouriño, J. Carlos 174
Mundani, Ralf-Peter 305

Nakashima, Kohta 9
Naruse, Akira 9
Niggli, Andreas 305

Panda, Dhabaleswar K. 230
Pellegrini, Simone 196
Pérache, Marc 94
Perogil Duque, J.F. 327
Peters, Norbert 313

Qian, Ying 250

Rank, Ernst 305
Rexachs, Dolores 73
Reyes, Ruymán 185
Rico Gallego, J.A. 327
Rojas-González, F. 294
Román-Alonso, G. 294
Ross, Robert 42, 164, 321
Rünger, Gudula 54

Sande, Francisco de 185
Santos, Guna 73
Sarholz, Samuel 313
Schaub, Torsten 64
Schneidenbach, Lars 64
Schnor, Bettina 64
Schulz, Martin 292
Sehrish, Saba 143

Siegel, Andrew R. 219
Siegel, Stephen F. 7, 219
Singh, David E. 207
Son, Seung Woo 321
Sridhar, Jaidev K. 230
Subhlok, Jaspal 124
Sumimoto, Shinji 9
Szubzda, Grzegorz 329

Taboada, Guillermo L. 174
Teijeiro, Carlos 174
Thakur, Rajeev 20, 42, 143,
 261, 271, 325, 329
Touriño, Juan 174
Träff, Jesper Larsson 20, 154
Trinitis, Carsten 292

Vakkalanka, Sarvani 261, 271, 329
Vo, Anh 261, 271, 329

Wang, Gang 84
Wang, Jie 196
Wang, Jun 143
Williams, Jason 271
Wolf, Felix 31
Wozniak, Justin M. 321

Yasui, Takashi 9

Zeng, Peng 313
Zhang, Fan 84
Zhu, Hao 325