HIGH-PERFORMANCE COMPUTING

Paradigm and Infrastructure

Edited by

LAURENCE T. YANG
MINYI GUO

A JOHN WILEY & SONS, INC., PUBLICATION

HIGH-PERFORMANCE COMPUTING

WILEY SERIES ON PARALLEL AND DISTRIBUTED COMPUTING

Albert Y. Zomaya, Series Editor

Parallel and Distributed Simulation Systems / Richard Fujimoto

Surviving the Design of Microprocessor and Multimicroprocessor Systems: Lessons Learned / Veljko Milutinović

Mobile Processing in Distributed and Open Environments / Peter Sapaty

Introduction to Parallel Algorithms / C. Xavier and S. S. Iyengar

Solutions to Parallel and Distributed Computing Problems: Lessons from Biological Sciences / Albert Y. Zomaya, Fikret Ercal, and Stephan Olariu (*Editors*)

New Parallel Algorithms for Direct Solution of Linear Equations / C. Siva Ram Murthy, K. N. Balasubramanya Murthy, and Srinivas Aluru

Practical PRAM Programming / Joerg Keller, Christoph Kessler, and Jesper Larsson Traeff

Computational Collective Intelligence / Tadeusz M. Szuba

Parallel and Distributed Computing: A Survey of Models, Paradigms, and Approaches / Claudia Leopold

Fundamentals of Distributed Object Systems: A CORBA Perspective / Zahir Tari and Omran Bukhres

Pipelined Processor Farms: Structured Design for Embedded Parallel Systems / Martin Fleury and Andrew Downton

Handbook of Wireless Networks and Mobile Computing / Ivan Stojmenoviić *(Editor)*

Internet-Based Workflow Management: Toward a Semantic Web / Dan C. Marinescu

Parallel Computing on Heterogeneous Networks / Alexey L. Lastovetsky

Tools and Environments for Parallel and Distributed Computing / S. Hariri and M. Parashar (*Editors*)

Distributed Computing: Fundamentals, Simulations and Advanced Topics, *Second Edition* / Hagit Attiya and Jennifer Welch

High-Performance Computing: Paradigm and Infrastructure / Laurence T. Yang and Minyi Guo (*Editors*)

HIGH-PERFORMANCE COMPUTING

Paradigm and Infrastructure

Edited by

LAURENCE T. YANG
MINYI GUO

A JOHN WILEY & SONS, INC., PUBLICATION

Published simultaneously in Canada

Library of Congress Cataloging-in-Publication Data:

High performance computing : paradigm and infrastructure / [edited by] Laurence Tianruo Yang and Minyi Guo.
p. cm.
Includes bibliographical references.
ISBN-13 978-0-471-65471-1 (cloth : alk. paper)
ISBN-10 0-471-65471-X (cloth : alk. paper)
1. High performance computing. 2. Parallel processing (Electronic computers) 3. Electronic data processing—Distributed processing. I. Yang, Laurence Tianruo. II. Guo, Minyi.
QA76.88.H538 2005
0004′.35—dc22
2004028291

Printed in the United States of America.

10 9 8 7 6 5 4 3 2 1

Contents

PART 4 Clusters and Grid Computing

Preface

The field of high-performance computing has obtained prominence through advances in electronic and integrated technologies beginning in the 1940s. With hyperthreading in Intel processors, hypertransport links in next-generation AMD processors, multicore silicon in today's high-end microprocessors from IBM, and emerging cluster and grid computing, parallel (and distributed) computing have moved into the mainstream of computing. To fully exploit these advances, researchers and industrial professionals have started to design parallel or distributed software systems and algorithms to cope with large and complex scientific and engineering problems with very tight timing schedules.

This book reports the recent important advances in aspects of the paradigm and infrastructure of high-performance computing. It has 41 chapters contributed by prominent researchers around the world. We believe all of these chapters and topics will not only provide novel ideas, new results, work-in-progress, and state-of-the-art techniques in the area, but also stimulate the future research activities in the area of high-performance computing with applications.

This book is divided into seven parts: programming model, architectural and system support, scheduling and resource management, clusters and grid computing, peer-to-peer computing, wireless and mobile computing, and high-performance applications. For guidance to readers, we will outline the main contributions of each chapter so that they can better plan their perusal of the material. Extensive external citations can be found in the individual chapters. Supplemental material can be found on the following ftp site: ftp://ftp.wiley.com/public/sci_tech_mea/high-performance_computing.

PART I PROGRAMMING MODEL

It has been noted that the acceptance of parallel computing mainly depends on the quality of a high-level programming model, which should provide powerful abstractions in order to free the programmer from the burden of dealing with low-level issues. In the first chapter, Chan et al. describe a high-level, graph-oriented pro-

gramming model called GOP and a programming environment called ClusterGOP, for building and developing message-passing parallel programs. In the second chapter, Chapman reviews the nature of HPC platforms and programming models that have been designed for use on them. She particularly considers OpenMP, the most recent high-level programming model that successfully uses compiler technology to support the complex task of parallel application development, and she discuss the challenges to its broader deployment in the HPC arena, as well as some possible strategies for overcoming them. In the third chapter, written by Gorlatch, the author describes the SAT (Stages and Transformations) approach to parallel programming, whose main objective is to systematize and simplify the programming process. In Chapter 4, Tikin surveys the state of the art in computation models and programming tools based on the bulk-synchronous parallel (BSP) model, which has now become one of the mainstream research areas in parallel computing, as well as a firm foundation for language and library design.

In Chapter 5, Morris et al. examine the use of two different parallel programming models in heterogeneous systems: MPI, a popular message passing system; and Cilk, an extension to C with dataflow semantics. They consider both the performance of the two systems for a set of simple benchmarks representative of real problems and the ease of programming these systems. The chapter by Gonzalez et al. describes two proposals for the OpenMP programming model arising from the detected lacks in two topics: nested parallelism and pipelined computations. It also presents a proposal introducing the precedence relations in the OpenMP programming model, which allows the programmers to specify explicit point-to-point thread synchronizations, being general enough to manage with any pipelined computation. The emergence of system-on-chip (SOC) design shows the growing popularity of the integration of multiple processors into one chip. In the chapter by Liu et al., the authors propose that a high-level abstraction of parallel programming like OpenMP is suitable for chip multiprocessors. They also present their solutions to extend OpenMP directives to tackle the heterogeneity problem. Several optimization techniques are proposed to utilize advanced architecture features of the target SOC, the Software Scalable System on Chip (3SoC).

PART 2 ARCHITECTURAL AND SYSTEM SUPPORT

The challenge in compiling programs for distributed memory systems is to determine data and computation decompositions so that load balance and low communication overhead can be achieved. In Chapter 8, Lee et al. describe their different compiler techniques for both regular and irregular programs. Hardware prefetching is an effective technique for hiding cache-miss latency and thus improving the overall performance. In Chapter 9, Zhang et al. propose a new prefetching scheme that improves performance without increasing memory traffic or requiring prefetch buffers. For concurrent I/O operations, atomicity defines the results in the overlapped file regions simultaneously read/written by multiple processes. In Chapter 10, Liao et al. investigate the problems arising from the implementation of MPI

atomicity for concurrent overlapping write operations. To avoid serializing the I/O, they examine two alternatives: (1) graph-coloring and (2) process-rank ordering. Compared to file locking, these two approaches are scalable and each presents different degrees of I/O parallelism.

For SOR-like PDE solvers, loop tiling either helps little in improving data locality or hurts performance. Chapter 11 by Xue et al. presents a novel compiler technique called code tiling for generating fast tiled codes for these solvers on uniprocessors with a memory hierarchy. This "one-size-fits-all" scheme makes their approach attractive for designing fast SOR solvers without having to generate a multitude of versions specialized for different problem sizes. To enable the process/thread migration and checkpointing schemes to work in heterogeneous environments, Liu and Chaudhary (Chapter 12) have developed an application-level migration and checkpointing package, MigThread, to abstract computation states at the language level for portability. To save and restore such states across different platforms, their chapter proposes a novel "receiver makes right" (RMR) data conversion method called coarse-grain tagged RMR (CGT-RMR), for efficient data marshalling and unmarshalling. In Chapter 13, Baba et al. address comprehensive techniques in receiving message-prediction and its speculative execution in message passing parallel programs. As communication cost is one of the most crucial issues in parallel computation and it prevents highly parallel processing, the authors focus on reduction of idle time in reception processes in message passing systems. They propose prediction algorithms and evaluate them by using NAS Parallel Benchmarks. In the last chapter of this part, Morrison et al. discuss the experiences and lessons learned from applying reconfigurable computing (FPGAs) to high-performance computing. Some example applications are then outlined. Finally, some speculative thoughts on the future of FPGAs in high-performance computing are offered.

PART 3 SCHEDULING AND RESOURCE MANAGEMENT

In Chapter 15, Hong and Prasanna consider the resource allocation problem for computing a large set of equal-sized independent tasks on heterogeneous computing systems. This problem represents the computation paradigm for a wide range of applications such as SETI@home and Monte Carlo simulations. In the next chapter, Antonopoulos et al. first present experimental results that prove the severity of bus saturation effects for the performance of multiprogrammed workloads on SMPs. The performance penalty is often high enough to nullify the benefits of parallelism. Driven by this observation, they introduce two gang-like, kernel-level scheduling policies that target bus bandwidth as a primary shared-system resource. The new policies attained an average 26% performance improvement over the native Linux scheduler. The most common objective function of task scheduling problems is makespan. However, on a computational grid, the second optimal makespan may be much longer than the optimal makespan because the computing power of a grid varies over time. So, if the performance measure is makespan, there is no approximation algorithm in general for scheduling onto a grid. In Chapter 17, Fujimoto and

Hagihara propose a novel criterion of a schedule. The proposed criterion is called total processor cycle consumption, which is the total number of instructions the grid could compute until the completion time of the schedule. The proposed algorithm does not use any prediction information on the performance of underlying resources.

The genetic algorithm has emerged as a successful tool for optimization problems. Vidyarthi et al. (Chapter 18) propose a task allocation model to maximize the reliability of distributed computing systems (DCSs) using genetic algorithms. Their objective in the chapter is to use the same GA proposed in their earlier work to minimize the turnaround time of the task submitted to the DCS and to compare the resultant allocation with the allocation in which reliability is maximized. Heterogeneous computing systems have gained importance due to their ability to execute parallel program tasks. In many cases, heterogeneous computing systems have been able to produce high performance for lower cost than a single large machine in executing parallel program tasks. However, the performance of parallel program execution on such platforms is highly dependent on the scheduling of the parallel program tasks on to the platform machines. In Chapter 19, Hagas and Janecek present a task scheduling algorithm on a bounded number of machines with different capabilities. The algorithm handles heterogeneity of both machines and communication links. A new model for classifying and extracting the application behavior is presented by Senger et al. in Chapter 20. The extraction and classification of process behavior are conducted through the analysis of data that represent the resource occupation. These data are captured and presented on an artificial self-organizing neural network, which is responsible for extracting the process behavior patterns. This architecture allows the periodic and on-line updating of the resource occupation data, which adapts itself to classify new behaviors of the same process. The proposed model aims at providing process behavior information for the load balancing algorithms that use this information in their selection and transference policies.

PART 4 CLUSTERS AND GRID COMPUTING

Computational grids that couple geographically distributed resources are becoming the de-facto computing platform for solving large-scale problems in science, engineering, and commerce. Chapter 21 by Luther et al. introduces design requirements of enterprise grid systems and discusses various middleware technologies that meet them. It mainly presents a .NET-based grid framework called Alchemi, developed as part of the Gridbus project. In the next chapter, the grid middleware issue is discussed. Grid middleware provide users with seamless computing ability and uniform access to resources in the heterogeneous grid environment. Several software toolkits and systems have been developed, most of which are results of academic research projects all over the world. Asadzadeh et al. focus on four of these middleware technologies: UNICORE, Globus, Legion, and Gridbus. They also present their implementation of a resource broker for UNICORE, as this functionality was not supported in it. A comparison of these systems on the basis of the architecture, implementation model, and several other features is included. A Distributed Java

Virtual Machine (DJVM) is a clusterwide virtual machine that supports parallel execution of a multithreaded Java application on clusters. The DJVM hides the physical boundaries between the cluster nodes and allows executed Java threads in parallel to access all cluster resources through a unified interface. In Chapter 23, Zhu et al. address the realization of a distributed Java virtual machine, named JESSICA2, on clusters, and detail its performance analysis.

The goal of data grids is to provide a large virtual storage framework with unlimited power through collaboration among individuals, institutions, and resources. In Chapter 24, Qin and Jiang first review a number of data grid services such as metadata service, data access service, and performance measurement service. They then investigate various techniques for boosting the performance of data grids; these key techniques fall into three camps: data replication, scheduling, and data movement. Since security issues become immediately apparent and critical in grid computing, they also review two intriguing issues related to security in data grids. Finally and importantly, they have identified five interesting open issues, and pointed out some potential solutions to the open problems. In Chapter 25, Bhardwaj discusses how the effect of the "I/O bottleneck" can be mitigated with parallel I/O operations using ROMIO MPI-IO implementation over parallel virtual file systems (PVFSs) on Linux clusters. As an application example, a seismic imaging algorithm has been shown to perform better with Parallel I/O.

Indiana University's AVIDD (Analysis and Visualization of Instrument-Driven Data) facility is the only geographically distributed cluster on the Top 500 list. It ranked 50th by June of 2003, achieving over 1 TFIops of LINPACK performance. In Chapter 26 by Wang et al., AVIDD's hardware and software setup are introduced, and their experience of performance tuning and benchmarking under the guidance of the existing LINPACK performance model is reported. In the end, the advantages of the distributed cluster-building approach are discussed based on the performance measurements. In Chapter 27, the porting on a Globus equipped platform of a hierarchically distributed, shared-memory parallel version of an application for particle-in-cell (PlC) simulation of plasma turbulence is described by Briguglio et al., based on the hierarchical integration of MPI and Open MP, and originally developed for generic (nongrid) clusters of SMP nodes. The service-oriented architectures generally, and web services in particular, have become important research areas in distributed and grid computing. For well-known reasons, a significant number of users have started to realize the importance of "trust" management for supporting business and scientific interactions electronically in service-oriented architectures. The last chapter of this part, authored by Ali et al., investigates the role of "trust" and "reputation" in the context of service provision and use, and proposes an architecture for utilizing these concepts in scientific applications.

PART 5 PEER-TO-PEER COMPUTING

In Chapter 29, Hsiao and King present two types of peer-to-peer (P2P) resource-discovery systems, namely, unstructured and structured P2P systems. Specially, the

authors discuss how to perform routing and joining operations in a structured P2P overlay. Additionally, they discuss how a structured P2P system can be enhanced to support searches by keyword, attribute-value pairs, and range. Blind flooding is a popular search mechanism used in current commercial P2P systems because of its simplicity. However, blind flooding among peers or superpeers causes large volumes of unnecessary traffic, although the response time is short. In some search mechanisms, not all peers may be reachable, creating the so-called partial coverage problem. Aiming at alleviating the partial coverage problem and reducing the unnecessary traffic, Liu et al. propose an efficient and adaptive search mechanism, Hybrid Periodical Flooding (HPF), in Chapter 30. HPF retains the advantages of statistics-based search mechanisms, alleviates the partial coverage problem, and provides the flexibility to adaptively adjust different parameters to meet different performance requirements. In order to improve the DHT-based P2P system routing performance, a routing algorithm called HIERAS, taking into account of the hierarchical structure, is proposed by Xu et al. in Chapter 31. In the last chapter of this part, Enokido and Takizawa discuss a group protocol that supports applications with group communication service when QoS supported by networks or required by applications is changed. An autonomic group protocol is realized by cooperation of multiple autonomous agents. Each agent autonomously takes a class of each protocol function. They make clear what combination of classes can be autonomously used by agents in a view. They also present how to autonomously change the manner of retransmission.

PART 6 WIRELESS AND MOBILE COMPUTING

Mobility management is important for both the 2G personal communications services (PCS) networks and the 3G cellular networks. The research presented by Patury et al. in Chapter 33 aims at implementing the gateway location register (GLR) concept in the 3G cellular networks by employing a caching location strategy at the location registers for certain classes of users meeting certain call and mobility criteria. Results obtained without any cache are compared with those from the three cases when a cache is applied only at the GLR, only at the visitor location register (VLR), and at both. Their study indicates when a cache is preferred in 3G cellular networks. Due to the limitation of available energy in the battery-driven devices in wireless ad hoc networks (WANETs), the longevity of the network is of prime concern. Chapter 34, written by Deng and Gupta, reviews the research on maximizing the lifetime of tree-based multicasting in WANETs. The problem can be divided into two categories. When a single tree is used throughout the multicast session, maximum multicast time can be found in polynomial time. In the other situation, where more than one multicast tree is used alternately, a quantized-time version of this problem has been proven to be NP-hard. In the next chapter, authored by Kim et al., first proposes two QoS-aware scatternet scheduling algorithms: a perfect algorithm for bipartite scatternets and a distributed local algorithm for general scatternets. Then, they analyze the QoS performance of the proposed algorithms

and evaluate the quantitative performance by simulation. For the objective performance evaluation, they devise a random scheduling algorithm and compare the performance results of both algorithms.

PART 7 HIGH-PERFORMANCE APPLICATIONS

In the first high-performance applications chapter, Harvey et al. present MinEX, a novel latency-tolerant partitioner that dynamically balances processor workloads while minimizing data movement and runtime communication for applications that are executed in a parallel distributed-grid environment. The chapter also presents comparisons between the performance of MinEX and that of METIS, a popular multilevel family of partitioners. These comparisons were obtained using simulated heterogeneous grid configurations. A solver for the classical N-body problem is implemented to provide a framework for the comparisons. In Chapter 37, by Cirne et al., the authors discuss how to run bag-of-tasks applications (those parallel applications whose tasks are independent) on computational grids. Bag-of-tasks applications are both relevant and amenable for execution on grids. However, few users currently execute their bag-of-tasks applications on grids. They investigate the reason for this state of affairs and introduce MyGrid, a system designed to overcome the identified difficulties. MyGrid also provides a simple, complete, and secure way for a user to run bag-of-tasks applications on all resources she or he has access to. In the next chapter, Tabirca et al. present an efficient method to calculate in parallel the values of the Smarandache function. The computation has an important constraint, which is to have consecutive values computed by the same processor. This makes the dynamic scheduling methods inapplicable. The proposed solution in the chapter is based on a balanced workload block scheduling method. Experiments show that the method is efficient and generates a good load balance.

In Chapter 39, Orendt et al. present a case study on the design, implementation, and deployment of a low-cost cluster for periodic comparisons of gene sequences using the BLAST algorithms. This work demonstrates that it is possible to build a scalable system that can accommodate the high demands imposed by the need for continuous updating of the existing comparisons due to newly available genetic sequences. Branch & bound applications represent a typical example of irregularly structured problems whose parallelization using hierarchical computational architectures (e.g., clusters of SMPs) involves several issues. In Chapter 40, Aversa et al. show how the combined use of PVM and OpenMP libraries allow us to develop hybrid code in order to introduce an additional dynamic distribution among the shared memory nodes of the system. Both coarse-grain and fine-grain parallelization are based on the coordinator/workers paradigm. In the last of the whole book, Bonifacio and Spinillo describe an advanced telecommunication service, based on a convergent architecture between TDM networks and IP networks. The service makes use of advanced protocols and a layered architecture, for the fast delivery of new features and the best leaveraging of existing legacy solutions. The chapter gives a

short overview of advanced elephony architectures and protocols, focusing then on describing the detailed implemented solutions.

ACKNOWLEDGMENTS

Of course, our division of this book into parts is arbitrary. The represented areas, as well, are not an exhaustive representation of the world of current high-performance computing. Nonetheless, they represent a rich and many-faceted body of knowledge that we have the pleasure of sharing with our readers. We would like to thank the authors for their excellent contributions and patience in assisting us. Last but not the least, we thank Professor Albert Y. Zomaya very much for helping and encouraging us to finish this work. All the great help and patience from Val Moliere and Emily Simmons of Wiley throughout this project are also very warmly acknowledged.

LAURENCE T. YANG
MINYI GUO

Antigonish, Nova Scotia, Canada
Fukushima, Japan
September 2005

Contributors

Rashid J. Al-Ali, School of Computer Science and Welsh eScience Centre, Cardiff University, Cardiff, United Kingdom

Ali Shaikh Ali, School of Computer Science and Welsh eScience Centre, Cardiff University, Cardiff, United Kingdom

Alejandro Sanchez Alvarado, Department of Neurobiology and Anatomy, University of Utah, Salt Lake City, Utah

Christos D. Antonopoulos, High Performance Information Systems Lab, Computer Engineering and Informatics Department, University of Patras, Patras, Greece

Anish Arora, Department of Computer and Information Science, The Ohio Statue University, Columbus, Ohio

Parvin Asadzadeh, Grid Computing and Distributed Systems Lab, Department of Computer Science and Software Engineering, University of Melbourne, Melbourne, Australia

R. Aversa, Department of Engineering Information, Second University of Naples, Aversa, Italy

E. Ayguade, Computer Architecture Department, Technical University of Catalonia, Barcelona, Spain

Takanobu Baba, Department of Information Science, Faculty of Engineering, Utsunomiya University, Utsunomiya, Tochigi, Japan

Carla Osthoff Barros, Department of Computer Systems, Federal University of Campina Grande, Campina Grande, Brazil

Dheeraj Bhardwaj, Department of Computer Science and Engineering, Indian Institute of Technology, Delhi, India

Laxmi Bhuyan, Department of Computer Science and Engineering, University of California, Riverside, California

Rupak Biswas, NAS Division, NASA Ames Research Center, Moffett Field, California

Anna Bonifacio, Covansys, Rome, Italy

Francisco Brasileiro, Department of Computer Systems, Federal University of Campina Grande, Campina Grande, Brazil

Sergio Briguglio, Association EURATOMENEA, Frascati, Rome, Italy

Rajkumar Buyya, Grid Computing and Distributed Systems Lab, Department of Computer Science and Software Engineering, University of Melbourne, Melbourne, Australia

Jiannong Cao, Department of Computing, Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong, China

Regina Helena Carlucci Santana, Department of Computer Science and Statistics, Mathematics and Computer Science Institute, University of Sao Paulo, Sao Carlos, Brazil

Fan Chan, Department of Computing, Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong, China

Barbara Chapman, Department of Computer Science, University of Houston, Houston, Texas

Vipin Chaudhary, Institute for Scientific Computing, Wayne State University, Detroit, Michigan

Alok Choudhary, Electrical and Computer Engineering Department, Northwestern University, Evanston, Illinois

Walfredo Cirne, Department of Computer Systems, Federal University of Campina Grande, Campina Grande, Brazil

Kenin Coloma, Electrical and Computer Engineering Department, Northwestern University, Evanston, Illinois

Lauro Costa, Department of Computer Systems, Federal University of Campina Grande, Campina Grande, Brazil

Sajal K. Das, Department of Computer Science, University of Texas at Arlington, Arlington, Texas

Guofeng Deng, Department of Computer Science and Engineering, Arizona State University, Tempe, Arizona

Beniamino Di Martino, Department of Engineering Information, Second University of Naples, Aversa, Italy

Tomoya Enokido, Department of Computers and Systems Engineering, Tokyo Denki University, Ishizaka, Hatoyama, Hiki, Saitama, Japan

Julio C. Facelli, Center for High Performance Computing, University of Utah, Salt Lake City, Utah

Weijian Fang, Department of Computer Science, University of Hong Kong, Pokfulam Road, Hong Kong, China

Rodrigo Fernandes de Mello, Department of Computer Science and Statistics, Mathematics and Computer Science Institute, University of Sao Paulo, Sao Carlos, Brazil

Giulana Fogaccia, Association EURATOMENEA, Frascati, Rome, Italy

Noriyuki Fujimoto, Graduate School of Information Science and Technology, Osaka University, Toyonaka, Osaka, Japan

Fumihito Furukawa, Venture Business Laboratory, Utsunomiya University, Tochigi, Japan

Marc Gonzalez, Computer Architecture Department, Technical University of Catalonia, Barcelona, Spain

Sergei Gorlatch, Institute for Information, University of Munster, Munster, Germany

Minyi Guo, Department of Computer Software, University of Aizu, Aizu-Wakamatsu-Shi, Japan

Rajiv Gupta, Department of Computer Science, University of Arizona, Tucson, Arizona

Sandeep K. S. Gupta, Department of Computer Science and Engineering, Arizona State University, Tempe, Arizona

Kenichi Hagihara, Graduate School of Information Science and Technology, Osaka University, Toyonaka, Osaka, Japan

Tarek Hagras, Department of Computer Science and Engineering, Czech Technical University, Praha, Czech Republic

Dave Hart, University Information Technology Services, Indiana University, Bloomington, Indiana

Daniel J. Harvey, Department of Computer Science, Southern Oregon University, Ashland, Oregon

Brian Haymore, Center for High Performance Computing, University of Utah, Salt Lake City, Utah

Philip D. Healy, Department of Computer Science, The Kane Building, The University of College Cork, Cork, Ireland

Bo Hong, Department of Electrical Engineering—Systems, University of California, Los Angeles, California

Hung-Chang Hsiao, Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan

Yiming Hu, Electrical and Computer Engineering, and Computer Science, University of Cincinnati, Cincinnati, Ohio

Qingguang Huang, School of Computer Science and Engineering, University of New South Wales, Sydney, Australia

Yohsiyuki Iwamoto, Nasu-Seiho High Schoo, Tochigi, Japan

Jan Janeček, Department of Computer Science and Engineering, Czech Technical University, Praha, Czech Republic

Hai Jiang, Institute for Scientific Computing, Wayne State University, Detroit, Michigan

Hong Jiang, Department of Computer Science and Engineering, University of Nebraska, Lincoln, Nebraska

Chun Ling Kei, Grid Computing and Distributed Systems Lab, Department of Computer Science and Software Engineering, University of Melbourne, Melbourne, Australia

JunSeong Kim, School of Electrical and Electronics Engineering, Chung-Ang University, Seoul, Korea

Young Man Kim, School of Computer Science, Kookmin University, Seoul, South Korea

Chung-Ta King, Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan

J. Labarta, Computer Architecture Department, Technical University of Catalonia, Barcelona, Spain

Ten H. Lai, Department of Computer and Information Science, The Ohio Statue University, Columbus, Ohio

Francis C. M. Lau, Department of Computer Science, University of Hong Kong, Pokfulam Road, Hong Kong, China

KyuHo Lee, School of Electrical and Electronics Engineering, Chung-Ang University, Seoul, Korea

PeiZong Lee, Institute of Information Science, Academia Sinica, Taipei, Taiwan

Jie Li, Department of Computer Science, Graduate School of Systems and Information Engineering, University of Tsukuba, Tsukuba Science City, Ibaraki, Japan

Wei-Keng Liao, Electrical and Computer Engineering Department, Northwestern University, Evanston, Illinois

Xiaola Lin, Department of Computer Science, City University of Hong Kong, Kowlon, Hong Kong, China

Feng Liu, Institute for Scientific Computing, Wayne State University, Detroit, Michigan

Yunhao Liu, Department of Computer Science and Engineering, Michigan State University, East Lansing, Michigan

Akshay Luther, Grid Computing and Distributed Systems Lab, Department of Computer Science and Software Engineering, University of Melbourne, Melbourne, Australia

X. Martorell, Computer Architecture Department, Technical University of Catalonia, Barcelona, Spain

N. Mazzocca, Department of Engineering Information, Second University of Naples, Aversa, Italy

John Morris, Department of Computer Science, The University of Auckland, Auckland, New Zealand

John P. Morrison, Department of Computer Science, The Kane Building, The University of College Cork, Cork, Ireland

Deepa Nayar, Grid Computing and Distributed Systems Lab, Department of Computer Science and Software Engineering, University of Melbourne, Melbourne, Australia

Lionel M. Ni, Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong, China

Dimitrios S. Nikolopoulos, Department of Computer Science, The College of William and Mary, Williamsburg, Virginia

Padraig O'Dowd, Department of Computer Science, The Kane Building, The University of College Cork, Cork, Ireland

Kanemitsu Ootsu, Department of Information Science, Faculty of Engineering, Utsunomiya University, Utsunomiya, Tochigi, Japan

Anita M. Orendt, Center for High Performance Computing, University of Utah, Salt Lake City, Utah

Yi Pan, Department of Computer Science, Georgia State University, Atlanta, Georgia

Mary Papakhian, University Information Technology Services, Indiana University, Bloomington, Indiana

Theodore S. Papatheodorou, Department of Computer Science, The College of William and Mary, Williamsburg, Virginia

Daniel Paranhos, Department of Computer Systems, Federal University of Campina Grande, Campina Grande, Brazil

Krishna Priya Patury, Department of Computer Science, Georgia State University, Atlanta, Georgia

Viktor K. Prasanna, Department of Electrical Engineering—Systems, University of California, Los Angeles, California

Neil Pundit, Scalable Computing Systems Department, Sandia National Laboratories, Albuquerque, New Mexico

Xiao Quin, Department of Computer Science, New Mexico Institute of Mining and Technology, Socorro, New Mexico

Kirti Rani, Department of Computer Science, Rani Durgavati University, Jabalpur, India

Omer F. Rana, School of Computer Science, Cardiff University, Cardiff, United Kingdom

Rajiv Ranjan, Grid Computing and Distributed Systems Lab, Department of Computer Science and Software Engineering, University of Melbourne, Melbourne, Australia

Kieran Reynolds, Department of Computer Science, The Kane Building, The University of College Cork, Cork, Ireland

David Richardson, Center for High Performance Computing, University of Utah, Salt Lake City, Utah

Sofia Robb, Department of Neurobiology and Anatomy, University of Utah, Salt Lake City, Utah

Eric Russell, Scalable Computing Systems Department, Sandia National Laboratories, Albuquerque, New Mexico

Marcos José Santana, Department of Computer Science and Statistics, Mathematics and Computer Science Institute, University of Sao Paulo, Sao Carlos, Brazil

Elizeu Santos-Neto, Department of Computer Systems, Federal University of Campina Grande, Campina Grande, Brazil

Biplab Kume Sarker, Department of Computer and Engineering, Kobe University, Kobe, Japan

Luciano José Senger, Department of Informatics, State University of Ponta Grossa, Ponta Grossa, Brazil

Steven Simms, University Information Technology Services, Indiana University, Bloomington, Indiana

G. Spinillo, Covansys, Rome, Italy

Craig Stewart, University Information Technology Services, Indiana University, Bloomington, Indiana

Sabin Tabirca, Department of Computer Science, The Kane Building, The University of College Cork, Cork, Ireland
Tatiana Tabirca, Department of Computer Science, The Kane Building, The University of College Cork, Cork, Ireland
Makoto Takizawa, Department of Computers and Systems Engineering, Tokyo Denki University, Ishizaka, Hatoyama, Hiki, Saitama, Japan
Alexander Tiskin, Department of Computer Science, University of Warwick, Conventry, United Kingdom
Anil Kumar Tripathi, Department of Computer Science, Faculty of Science, Banaras Hindu University, Varanasi, India
George Turner, University Information Technology Services, Indiana University, Bloomington, Indiana
Salvatore Venticinque, Department of Engineering Information, Second University of Naples, Aversa, Italy
Srikumar Venugopal, Grid Computing and Distributed Systems Lab, Department of Computer Science and Software Engineering, University of Melbourne, Melbourne, Australia
Deo Prakash Vidyarhi, Department of Computer Science, Faculty of Science, Banaras Hindu University, Varanasi, India
Gregorio Vlad, Association EURATOMENEA, Frascati, Rome, Italy
John Paul Walters, Institute for Scientific Computing, Wayne State University, Detroit, Michigan
Chien-Min Wang, Institute of Information Science, Academia Sinica, Taipei, Taiwan
Cho-Li Wang, Department of Computer Science, University of Hong Kong, Pokfulam Road, Hong Kong, China
Peng Wang, University Information Technology Services, Indiana University, Bloomington, Indiana
Lee Ward, Scalable Computing Systems Department, Sandia National Laboratories, Albuquerque, New Mexico
Jan-Jan Wu, Institute of Information Science, Academia Sinica, Taipei, Taiwan
Li Xiao, Department of Computer Science and Engineering, Michigan State University, East Lansing, Michigan
Yang Xiao, Computer Science Division, The University of Memphis, Memphis, Tennessee
Zhiyong Xu, Department of Computer Science and Engineering, University of California, Riverside, California
Jingling Xue, School of Computer Science and Engineering, University of New South Wales, Sydney, Australia
Laurence T. Yang, Department of Computer Science, St. Francis Xavier University, Antigonish, Nova Scotia, Canada
Takashi Yokota, Department of Information Science, Faculty of Engineering, Utsunomiya University, Utsunomiya, Tochigi, Japan
Youtao Zhang, Department of Computer Science, University of Texas at Dallas, Richardson, Texas

Wenzhang Zhu, Department of Computer Science, University of Hong Kong, Pokfulam Road, Hong Kong, China

Zhenyun Zhuang, Department of Computer Science and Engineering, Michigan State University, East Lansing, Michigan

CHAPTER 1

ClusterGOP: A High-Level Programming Environment for Clusters

FAN CHAN, JIANNONG CAO, and MINYI GUO

1.1 INTRODUCTION

Traditionally, parallel programs are designed using low-level message-passing libraries, such as PVM [18] or MPI [29]. Message-passing (MP) provides the two key aspects of parallel programming: (1) synchronization of processes and (2) read/write access for each processor to the memory of all other processors [23]. However, programmers still encountered difficulties when using MP facilities to develop parallel programs, especially for large-scale applications. First, MP facilities such as MPI and PVM provide low-level programming tools. Their interfaces are simple but force the programmer to deal with low-level details, and their functions are too complicated to use for a nonprofessional programmer. The low-level parallel primitives make the writing of real-world parallel applications tedious and error-prone. Second, often, the main implementation complexity arises from the process management of MP facilities. Applications require the programming of process management, which is not an easy task. Third, MPMD (multiple program multiple data) programming is difficult in many situations without good support. For example, MPI is mainly for SPMD (single program multiple data) programming. Even with some new and enhanced features in MPI-2 [20], the support is not sufficient for programmers to take advantage of the MPMD paradigm. Consequently, programming parallel applications requires special techniques and skills, which are different for each MP facility, and programmers have to rely on their experience, quite often in an ad-hoc way.

It has been agreed that the acceptance of parallel computing mainly depends on the quality of a high-level programming model, which should provide powerful abstractions in order to free the programmer from the burden of dealing with low-level issues [3]. Several high-level models have been developed for message-pass-

High-Performance Computing: Paradigm and Infrastructure. Edited by L. T. Yang and M. Guo

ing programming. Ensemble [12, 13, 14] supports the design and implementation of message-passing applications (running on MPI and PVM), particularly MPMD and those demanding irregular or partially regular process topologies. In Ensemble, the applications are built by composition of modular message-passing components. There are some other high-level languages (e.g., Mentat [19], C++ [9], and Fortran-M [16]), and runtime systems (e.g., Nexus [17]), which support combinations of dynamic task creation, load balancing, global name space, concurrency, and heterogeneity. Programming platforms based on an SPMD model (e.g., Split-C [15] and CRL [22]) usually have better performance than MPMD-based ones. MPRC [11, 10] is an MPMD system that uses RPC as the primary communication abstraction and can produce good results compared with the SPMD model. Some proramming tools integrate message passing with other parallel programming paradigms to enhance the programming support. The Nanothreads Programming Model (NPM) [21] is such a programming model; it integrates shared memory with MPI and offers the capability to combine their advantages. Another example is Global Arrays (GA) [26], which allows programmers to easily express data parallelism in a single, global address space. GA provides an efficient and portable shared-memory programming interface for parallel computers. The use of GA allows the programmer to program as if all the processors have access to the same data in shared memory.

Graphical programming environments have also been developed to ease parallel programming using visual and interactive approaches. CODE [2, 24] provides a visual programming environment with a graphical parallel programming language. The programmer can create a parallel program by drawing a dataflow graph that shows the communication structure of the program. HeNCE [1] is a graphical environment for creating and running parallel programs in graphs over a heterogeneous collection of computers. Differing from CODE, the graph in HeNCE shows the control flow of a program [28]. VPE [25] is a visual parallel programming environment. It provides a simple GUI for creating message-passing programs and supports automatic compilation, execution, and animation of the programs.

In this chapter, we describe a high-level, graph-oriented programming model called GOP and a programming environment called ClusterGOP for building and developing message-passing parallel programs. GOP [5, 4] supports high-level design and programming of parallel and distributed systems by providing built-in support for a language-level construct and various operations on a high-level abstraction called the *logical graph.* With GOP, the configuration of the interacting processes of a parallel/distributed program can be represented as a user-specified logical graph that is mapped onto the physical network topology. The programming of interprocess communication and synchronization between local processors is supported by built-in primitives for graph-based operations. ClusterGOP is a programming environment built on top of the GOP model, providing a high-level programming abstraction through the ClusterGOP library for building parallel applications in clusters, It contains tools for development of GOP-based parallel programs with intelligent, visual support and a run-time system for deployment, execution, and management of the programs. ClusterGOP supports both SPMD and MPMD

parallel computing paradigms [7, 8]. Also, with ClusterGOP, developing large parallel programs can be simplified with the predefined graph types and scalability support. The ClusterGOP system is portable as its implementation is based almost exclusively on calls to MPI, a portable message-passing standard.

The rest of chapter is organized as follows. Section 1.2 presents an overview of the ClusterGOP framework, including the GOP model and the ClusterGOP architecture. Section 1.3 describes the main features of the VisualGOP tool for supporting visual program development, from program construction to process mapping and compilation. In Section 1.4, we describe the ClusterGOP library of programming primitives. Support for MPMD programming in ClusterGOP is discussed in Section 1.5. Section 1.6 illustrates how parallel program development is done in ClusterGOP by using an example. Finally Section 1.7 concludes the chapter with a discussion of our future work.

1.2 GOP MODEL AND ClusterGOP ARCHITECTURE

ClusterGOP is based on the GOP model. In developing GOP for parallel programming, it was our observation that many parallel programs can be modeled as a group of tasks performing local operations and coordinating with each other over a logical graph, which depicts the program configuration and intertask communication pattern of the application. Most of the graphs are regular ones such as tree and mesh. Using a message-passing library, such as PVM and MPI, the programmer needs to manually translate the design-level graph model into its implementation using low-level primitives. With the GOP model, such a graph metaphor is made explicit at the programming level by directly supporting the graph construct in constructing the program.

The key elements of GOP are a logical graph construct to be associated with the local programs (LPs) of a parallel program and their relationships, and a collection of functions defined in terms of the graph and invoked by messages traversing the graph. In GOP, a parallel program is defined as a collection of LPs that may execute on several processors [4, 5]. Parallelism is expressed through explicit creation of LPs and communication between LPs is solely via message passing. GOP allows programmers to configure the LPs into a logical graph, which can serve the purpose of naming, grouping, and configuring LPs. It can also be used as the underlying structure for implementing uniform message passing and LP coordination mechanisms. The code of the LPs can be implemented using a set of high-level operations defined over the graph. As shown in Figure 1.1, the GOP model consists of the following parts:

- A logical graph (directed or undirected) whose nodes are associated with LPs, and whose edges define the relationships between the LPs.
- An LPs-to-nodes mapping, which allows the programmer to bind LPs to specific nodes.

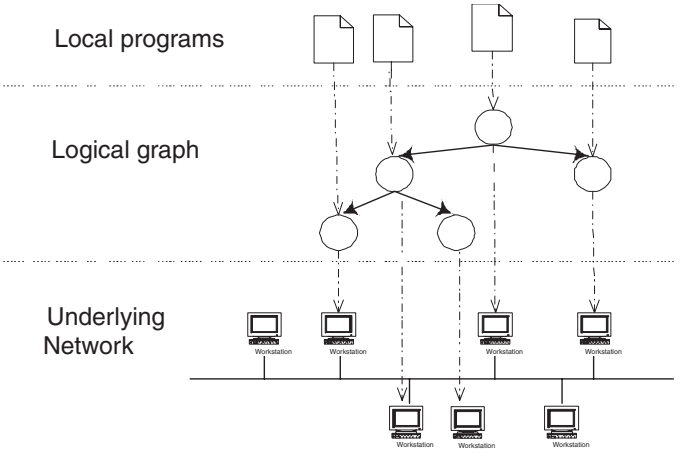

Figure 1.1 The GOP conceptual model.

- An optional nodes-to-processors mapping, which allows the programmer to explicitly specify the mapping of the logical graph to the underlying network of processors. When the mapping specification is omitted, a default mapping will be performed.
- A library of language-level, graph-oriented programming primitives.

The GOP model provides high-level abstractions for programming parallel applications, easing the expression of parallelism, configuration, communication, and coordination by directly supporting logical graph operations (GOP primitives). GOP programs are conceptually sequential but augmented with primitives for binding LPs to nodes in a graph, with the implementation of graph-oriented, inter-node communications completely hidden from the programmer. The programmer first defines variables of the graph construct in a program and then creates an instance of the construct. Once the local context for the graph instance is set up, communication and coordination of LPs can be implemented by invoking operations defined on the specified graph. The sequential code of LPs can be written using any programming language such as C, C++, or Java.

1.2.1 The ClusterGOP Architecture

The framework of the ClusterGOP programming environment is illustrated in Figure 1.2. The top layer is VisualGOP, the visual programming tool [6]. The ClusterGOP Application Programming Interface (API) is provided for the programmer to build parallel applications based on the high-level GOP model. The ClusterGOP library provides a collection of routines implementing the ClusterGOP API, with a very simple functionality to minimize the package overhead [7].

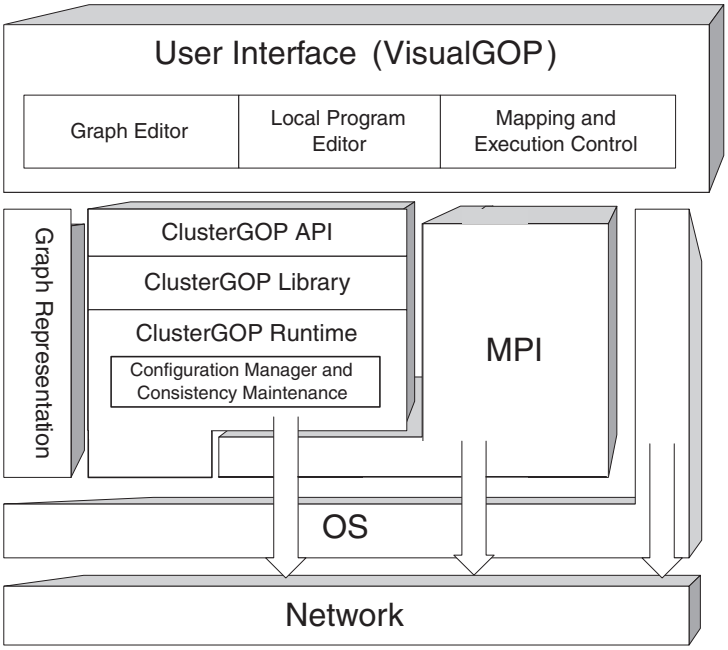

Figure 1.2 The ClusterGOP framework.

The ClusterGOP Runtime is responsible for compiling the application, maintaining the logical graph structure, and executing the application. On each machine, there exist two runtimes. The first one is the ClusterGOP runtime, a background process that provides graph deployment, update, querying, and synchronization. The second is the MPI runtime, which is used by ClusterGOP as the low-level parallel programming facility for the ClusterGOP implementation.

As shown in Figure 1.3, the architecture of ClusterGOP is divided into three layers: the programming layer, the compilation layer, and the execution layer. In the programming layer, the programmer develops a ClusterGOP program using GOP's high-level abstraction. ClusterGOP exports a set of APIs with supporting library routines that provides the implementation of the parallel applications with traditional programming languages, e.g., the C language.

In the compilation layer, LPs will be transformed into an executable program in the target's execution environment. The LPs are compiled with both the MPI and the ClusterGOP libraries.

The bottom execution layer is realized through the services of two important runtimes, the MPI runtime and the ClusterGOP runtime. The MPI runtime is used by ClusterGOP for implementing process communication through the network. The ClusterGOP runtime is developed as a daemon process on top of the operating system on each node. It helps the ClusterGOP processes to dynamically resolve the

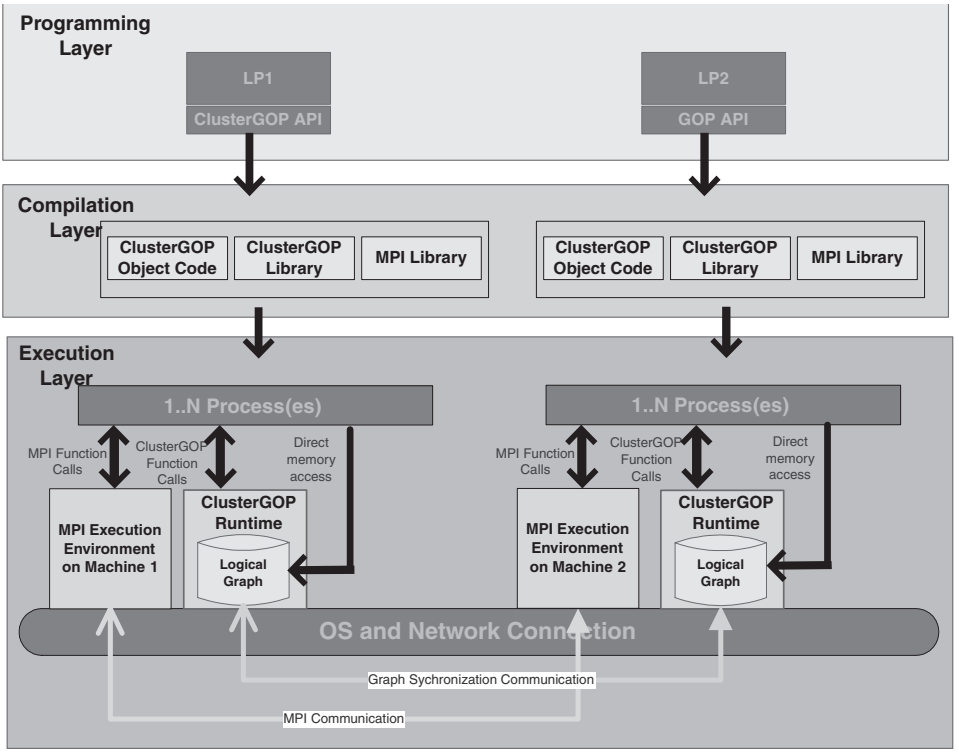

Figure 1.3 The ClusterGOP program communication implementation.

node names into process IDs and to prepare for MPI communication. A graph representation is used for the operations defined by ClusterGOP primitives. The ClusterGOP runtime provides the synchronization among all the nodes, so that each node can maintain the most updated logical graph for running ClusterGOP primitives. The ClusterGOP runtime system is implemented by the C language with socket communication and synchronization functions. Nodes in the same machine use the shared memory to access the graph. Nodes on different machines use a memory coherence protocol called the Sequential Consistency Model [27] to synchronize graph updating.

1.3 VISUALGOP

VisualGOP is a visual tool that supports the design, construction, and deployment of ClusterGOP programs. It helps the programmer further eliminate many coding details and simplify many of the program development tasks.

The organization of ClusterGOP components is shown in Figure 1.4. There are two levels: the visual level and the nonvisual level. Figure 1.5 shows a screen of

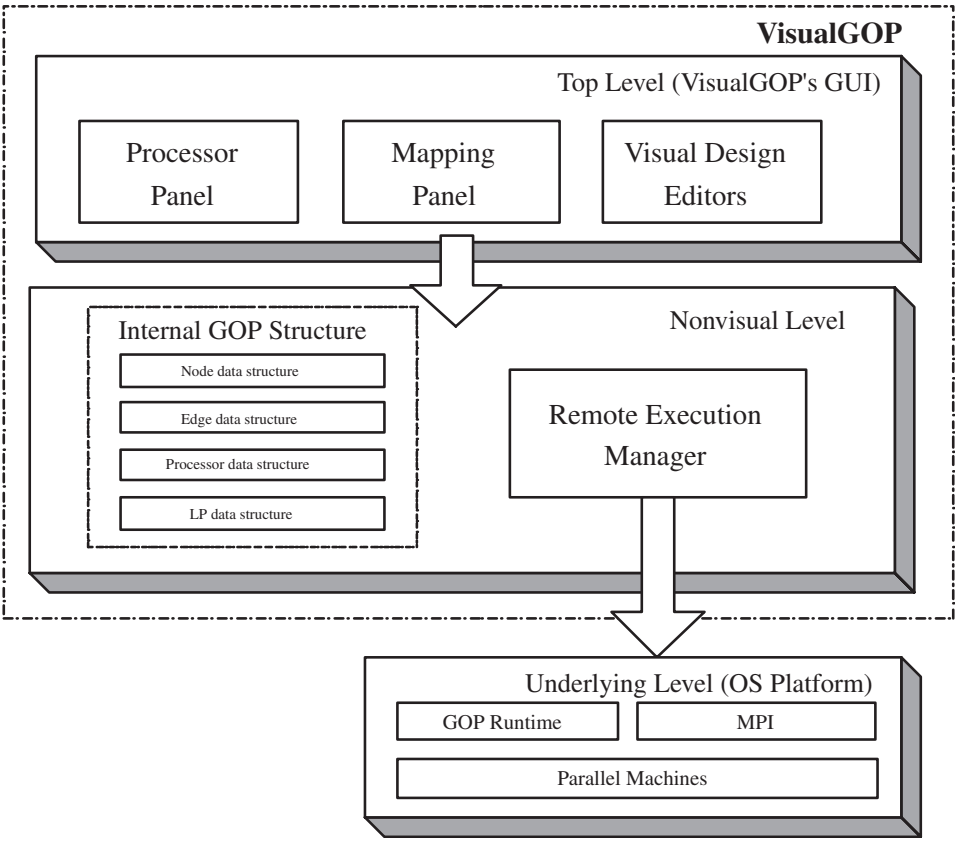

Figure 1.4 The VisualGOP architecture.

the main visual interface of VisualGOP. The components for visual program construction provide graphical aids (visual design editors) for constructing a ClusterGOP program. The graph editor is used to build the logical graph, representing the program structure from a visual perspective. The LP editor is used for editing the source code of LPs in a ClusterGOP program. The components for mapping and resource management provide the programmer the control over the mapping of LPs to graph nodes and the mapping of graph nodes to processors (through the mapping panel). They also allow the programmer to access the information about the status of the machine and network elements (through the processor panel).

The nonvisual level contains components responsible for maintaining a representation of GOP program design and deployment information. This representation is kept in memory when program design takes place, and is later stored in a file. Components in this level are also responsible for compilation and execution of the constructed programs. Compilation transforms the diagrammatic representations

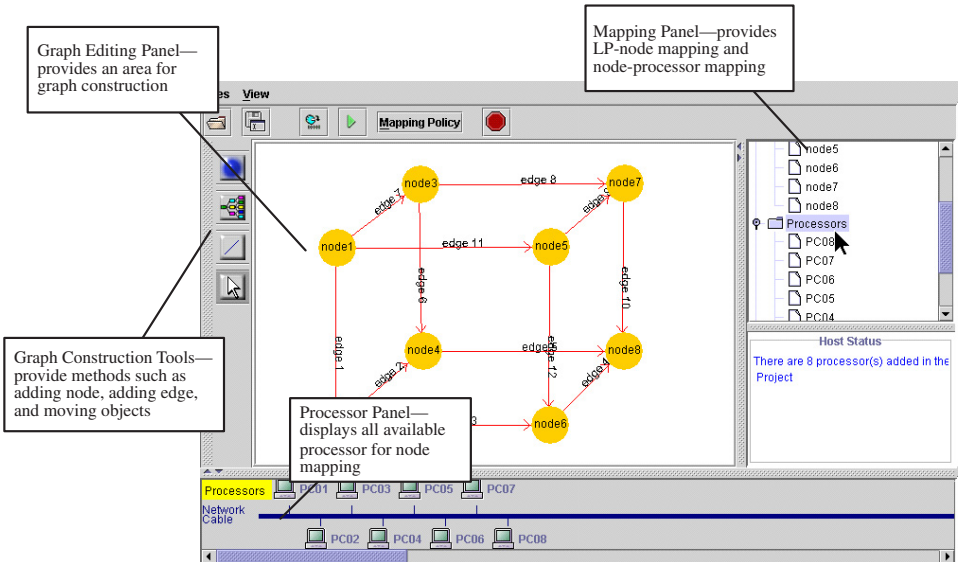

Figure 1.5 The main screen of VisualGOP.

and GOP primitives in the LPs into the target machines code. The remote execution manager makes use of the stored information to execute the constructed GOP program on a given platform.

With VisualGOP, the programmer starts program development by building a highly abstracted design and then transforms it successively into an increasingly more detailed solution. The facilities provided help the programmer to separate program design and configuration (i.e., the definition of the logical graph) from implementing the program components (i.e., the coding of the LPs). VisualGOP also supports the automation of generating a large portion of GOP code such as declaration and initialization of graph constructs and generation of message templates. The visual programming process under VisualGOP consists of the following iterative stages:

1. Visual programming. The programmer interacts with VisualGOP through a visual representation of the design of a parallel program. The design is presented as a logical graph that consists of a set of nodes representing LPs and a set of edges representing the interactions between the LPs in the program. The graph editor supports the creation and modification of a logical graph and the LP editor is used to visually manipulate the textual LP source code.
2. Binding of LPs to graph nodes. The programmer specifies the binding of LPs of a parallel program to the nodes of the logical graph. The programmer has two ways of doing this. The first way is to create the LPs and then bind them to the graph nodes. Another way is to combine the two steps into one: click

on a node of the graph to open the program editor by which the code of the LP mapped to the node can be entered.

3. Mapping of graph nodes to processors. The programmer specifies the mapping of the nodes in the graph to the processors in the underlying network. The mapping panel of VisualGOP displays the GOP program elements (nodes, processors, LPs) in a hierarchical tree structure. The programmer can use drag-and-drop to bind and unbind the LPs to graph nodes and the graph nodes to processors. Detailed binding information concerning a node can be viewed by selecting and clicking the node. The processor panel displays processors and their connections, and also provides the drag-and-drop function to bind and unbind graph nodes to one of the processors.
4. Compile the LPs. The programmer uses this function to distribute the source files to the specified processors for compilation.
5. Execute the program. The programmer uses this function to execute the constructed GOP program on the specified processors. Outputs will be displayed on the VisualGOP.

During the program design process, VisualGOP provides the user with automated, intelligent assistance. Facilities provided include the generation of ClusterGOP primitives in editing LP code, checking the consistency of LP code against the logical graph specified by the programmer, and support for scalability [6]. All of these are achieved through visial interactions with the tools in VisualGOP.

1.4 THE ClusterGOP LIBRARY

The ClusterGOP runtime has been implemented on top of MPI. Graph-oriented primitives for communications, synchronization, and configuration are perceived at the programming level and their implementation hides the programmer from the underlying programming activities associated with accessing services through MPI. The programmer can thus concentrate on the logical design of an application, ignoring unnecessary low-level details.

The implementation of ClusterGOP applications is accomplished through wrapping functions (ClusterGOP library) to native MPI software. In general, ClusterGOP API adheres closely to MPI standards. However, the ClusterGOP library simplifies the API by providing operations that automatically perform some MPI routines. It also allows the argument list to be simplified.

ClusterGOP provides a set of routines to enable graph-oriented point-to-point and collective communications (in both blocking and nonblocking modes). The API follows the MPI standard, but it is simpler and the implementation is specifically designed for the graph-oriented framework. For example, graph node ID is used instead of process ID to represent different processes, so the LP bound to a node can be replaced without affecting other LPs. ClusterGOP API also hides the complexity of low-level addressing and communication, as well as initializing processes from

the programmer. Three types of communication and synchronization primitives are provided [7]:

1. *Point-to-point communication* consists of a process that sends a message and another process that receives the message—a send/receive pair. These two operations are very important as well as being the most frequently used operations. To implement optimal parallel applications, it is essential to have a model that accurately reflects the characteristics of these basic operations. ClusterGOP provides primitives for a process to send a message to and receive a message from a specific graph node, as well as primitives for sending messages to nodes with relationships defined in terms of the graph, such as parents, children, neighbors, and so on.
2. *Collective Communication* refers to message passing involving a group (collection) of processes. Sometimes, one wishes to gather data from one or more processes and share this data among all participating processes. At other times, one may wish to distribute data from one or more processes to a specific group of processes. In ClusterGOP API, another type of collection communication primitives using the parent and child relationships is provided. It is used for finding the parent or child nodes, and then broadcasts the data to all the corresponding nodes.
3. *Synchronization operations* are provided to support the synchronization of processes.

ClusterGOP also provides a set of operations to query the information about nodes and edges in the graph. The information can be generated during the running of the application. Programmers can use the query information in communication. For example, when programmers want to find the neighbor node connected to the current node, they can use an API function to retrieve the node name of a connected edge. Programming in this way helps programmers dynamically assign the node names in the communication procedures, without specifying static node names in the LP code. Therefore, it helps programmers design the LP structure freely, and produces a more readable code for software maintenance.

1.5 MPMD PROGRAMMING SUPPORT

MPMD is advantageous when the application becomes large and complex with heterogeneous computations requiring irregular or unknown communication patterns. MPMD separates the application into different loosely coupled functional modules with different program sources for concurrent tasks, thus promoting code reuse and the ability to compose programs.

With the MPMD programming model under ClusterGOP, each LP is associated with a separate source code. Data can be distributed and exchanged among the LPs. ClusterGOP also has a better node-group management than MPI so that the processes can form groups easily and efficiently. The main features in ClusterGOP

to support high-level MPMD programming include forming process groups, data distribution among the processes, and deployment of processes for execution. With these features, programmers can program group communication by using NodeGroup, manage distributed tasks and processors through visual interfaces, map resources to tasks, and compile and execute programs automatically. The underlying implementation using MPI is hidden from the programmer.

In ClusterGOP, we use a NodeGroup to represent a group of processes, providing the same functionality as the MPI communicator. NodeGroup helps the programmer to write code for group communications and is implemented using MPI's communicator. A NodeGroup consists of a set of member processes. Each process in the group can invoke group communication operations such as collective communications (gather, scatter, etc.). In the design phase, VisualGOP provides a visual way for representing the NodeGroup involved in a MPMD program. The programmer can highlight the NodeGroup in the logical graph to specify the nodes that belong to the NodeGroup. When the programmer wants to use the collective operations, VisualGOP provides the valid NodeGroup for programmer to select. It also hides the programming details that are used for constructing the NodeGroup in the MPMD programs so that the programmer can concentrate on programming the nodes' tasks. As a result, the program is easier to understand.

In the MPMD programming model, tasks have different programs to execute but usually need to exchange or share some data. MPI provides API functions for distributing data to different processes, but the programmer still has to write the code for the data distribution procedure. In ClusterGOP, tasks share data by keeping a portion of the global memory in each node that is involved in the communication. The node can update the memory without having to communicate with other nodes. Using VisualGOP, data distribution can be performed by the programmer through the visual interface. The global memory can be created by selecting an option in VisualGOP: vertical, horizontal, or square memory distribution. By default, the memory is distributed to the nodes in a balanced way such that each node will almost share the same amount of the distributed data object. VisualGOP also provides a visual interface to allow the programmer to manually specify the memory distribution on each node. ClusterGOP translates the data distribution specified by the programmer into MPI code.

In many exisiting systems, the distributed memory object is a built-in function and most of the objects are distributed in the whole environment. However, due to its complex design nature, overheads occur frequently that reduce its efficiency. ClusterGOP implements the DSM in a different way in that distributed objects are used only if programmer explicitly demands them. ClusterGOP implements the global memory functions by using the GA toolkit [26], which is based on MPI for communications. GA provides an efficient and portable shared-memory programming interface. In ClusterGOP, before compiling the application, all distributed objects are converted into Global Array (GA) codes.

In MPMD programming, managing the mapping of nodes (tasks) to processors can be a complicated task. VisualGOP provides the support for visually mapping LPs to nodes and nodes to processors, and for automatic compilation and execution

of the applications. This facilitates the development process and simplifies the running of the large-scale MPMD application.

1.6 PROGRAMMING USING ClusterGOP

1.6.1 Support for Program Development

In VisualGOP, programmers are provided with a variety of abstractions that are useful in developing parallel programs. When designing a parallel program, the programmer starts by constructing a graph that shows the program structure. A graph is constructed in VisualGOP as a structure containing data types for describing a conceptual graph. The programmer can use the graph editing panel to draw the graph, which will be translated to the textual form. The graph editing panel displays the graphical construction view of VisualGOP, and this is where all of the editing of the program's logical graph structure takes place. Controls are provided for drawing the graph components within this panel. These controls include functions for adding nodes, subgraph nodes, and edges, as well as for moving nodes. Also, edges can be added between two nodes by joining the source and the destination nodes.

The programmer then can use the LP editor to define source code for LPs attached to the graph nodes (see Figure 1.6). The programmer can type in the code or

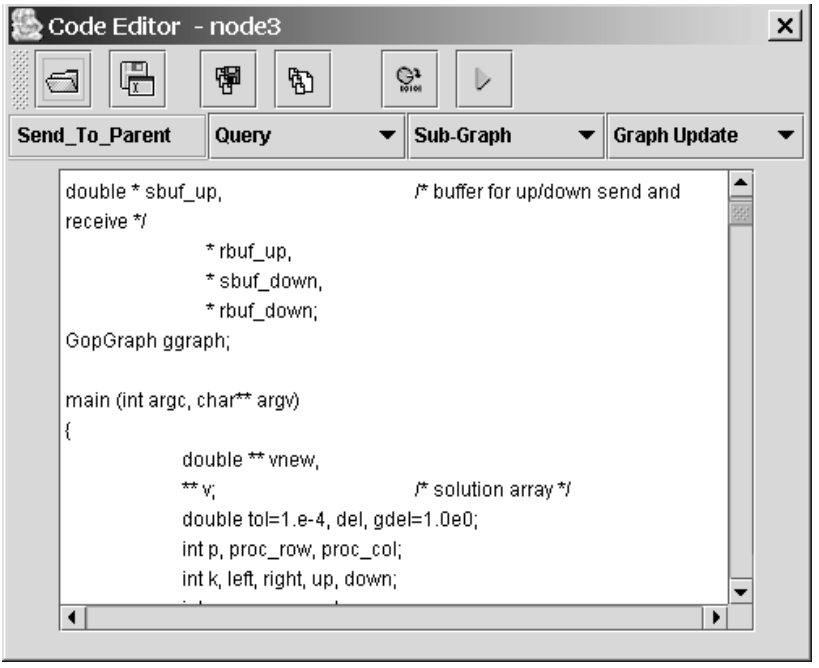

Figure 1.6 Editing the LP.

load the existing program source into the editor. After that, the new LP will be bound to the node automatically. ClusterGOP primitives can be automatically generated from visual icons. Also, the LP editor provides consistency checking to prevent the creation of graph semantic errors. For example, the programmer is not allowed to make incorrect connections, such as connecting a node to a nonexistent node. To assist the programmer with graph-oriented programming, the editor has features to ensure that newly created or modified node or edge names are correct within the logical graph structure.

The LPs of the parallel program need to be bound to the nodes of a logical graph for naming, configuration, and communication purposes. They also need to be distributed to processors in the underlying network for execution through the ClusterGOP runtime system. Therefore, after the logical graph and LPs are created, the programmer needs to bind the LPs to the nodes, and to set up a processor list for binding the nodes to the processors. These are achieved by using VisualGOP's LP-to-node mapping and node-to-processor mapping functions. When the programmer selects the status of a specific node, an LP can be chosen from the node property's pull-down menu. After the selection, the LP is mapped to that node. When using the node-to-processor function, the programmer can explicitly specify the mapping by dragging and dropping the node into one of the processors in the processor panel, or let the GOP system automatically perform task allocation. Once mapped, the graph node has all this required information such as IP address/hostname, compiler path, and so on.

The mapping panel presents the graph component properties in a simple tree structure. The components are classified into one of three categories: node, processor, or local program. Each component shows its properties in the program, and the mapping relationship that it belongs to. It is updated automatically whenever the program structure is modified in the graph editing panel.

Let us look at an example (see Figure 1.7). We first define a mapping, M1, for a MPMD program, which defines the relationships between the graph nodes and the LPs for the program. In the map, there are several types of LPs: the `distributor`, which receives and distributes the data, and the calculator, which calculates and submits its own partial data to the `distributor` and receives data from the neighbor nodes and from the `distributor`. Our definition of M1 is (given in the C language, where LV-MAP is the corresponding map data type):

```
LV-MAP M1 =
{ {0, "distributor"}, {1, "calculatorA"}, {2, "calculatorA"}, {3, "calculatorB"},
{4, "calculatorA"}, {5, "calculatorA"}, {6, "calculatorB"}, {7, "calculatorC"},
{8, "calculatorC"}, {9, "calculatorD"} }
```

Then, we can specify a mapping, M2, of the graph nodes onto the processors. With the aid of the processor panel, the node is mapped onto the target processor. It is assumed that each node will be executed on a different processor. The processor panel displays information about processor availability. If it is currently assigned, the relevant node is indicated.

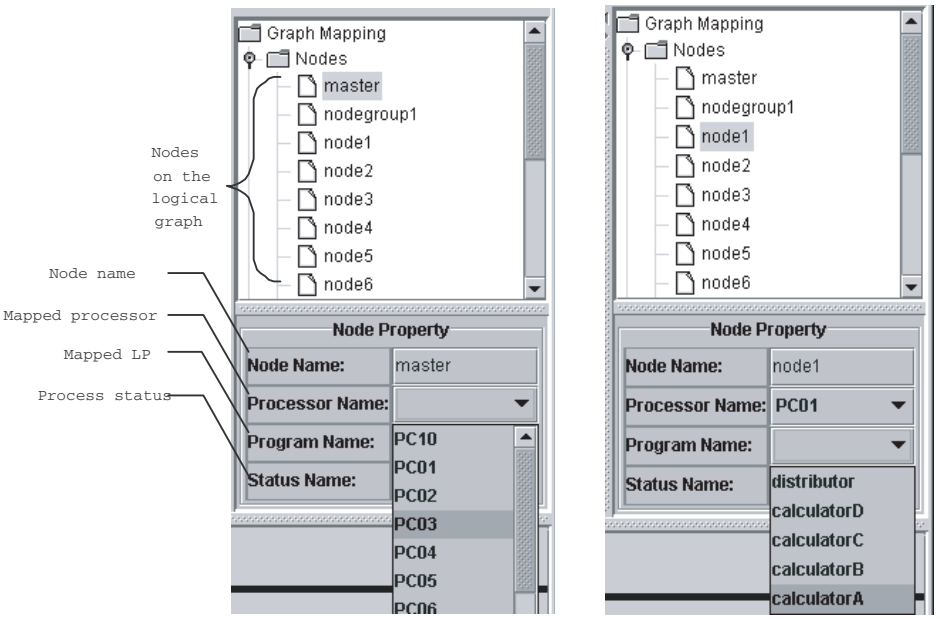

Figure 1.7 LP-to-node and node-to-processor mapping.

The final step in building an application is to compile and execute the LPs on the target processors. VisualGOP automatically generates the compiled machine code and starts the program execution remotely. It recognizes the LP's implementation language type and invokes the corresponding remote compilation and execution procedures. Remote ClusterGOP runtimes installed on the target machines are listening for incoming requests.

When the programmer selects the remote compilation option for a node, he/she can specify the processor to be used as the target for compiling the program source for execution. The machine's name is the same as that of the processor that the programmer specified in the graph panel. After a processor is selected, the program source is made ready for compilation. The programmer is only required to input the command argument that is used for compiling or executing the LP on the target machine. After that, the remote compilation and execution will be done automatically. The execution results can be viewed through VisualGOP.

1.6.2 Performance of ClusterGOP

Serveral example applications have been implemented on ClusterGOP, including the finite difference method, parallel matrix multiplication, and two-dimensional fast Fourier transform [8]. The performance of the applications are evaluated using MPI and ClusterGOP. We use Parallel Matrix Multiplication to show how ClusterGOP supports the construction and running of parallel programs on clusters.

In this example, we consider the problem of computing $C = A \times B$, where A, B, and C are dense matrices of size $N \times N$ (see Equation 1.1).

$$C_{ij} = \sum_{k=0}^{N-1} A_{ik} \times B_{kj} \tag{1.1}$$

As shown in Figure 1.8, a 3×3 mesh is defined in this example. Besides $N \times N$ nodes in the mesh, there is an additional node, named "master," which is connected to all the nodes of the mesh. There are two types of programs in this example: "distributor" and "calculator." There is only one instance of "distributor," which is associated with the "master" node. Each mesh node is associated with an instance of "calculator." The "distributor" program first decomposes the matrices A and B into blocks, whose sizes are determined by the mesh's dimension. It then distributes the blocks to the nodes on the left-most column and the nodes on the bottom rows, respectively. Each "calculator" receives a block of matrix A and matrix B from its left edge and bottom edge, and also propagates the block along its right edge and top edge. After data distribution, each "calculator" calculates the partial product and sends it back to the "distributor." The "distributor" assembles all the partial products and displays the final result.

The experiments used a cluster of twenty-five Linux workstations; each workstation running on a Pentium-4 2GHz. The workstations are set up with MPICH 1.2 and all the testing programs are written in C. Execution times were measured in sec-

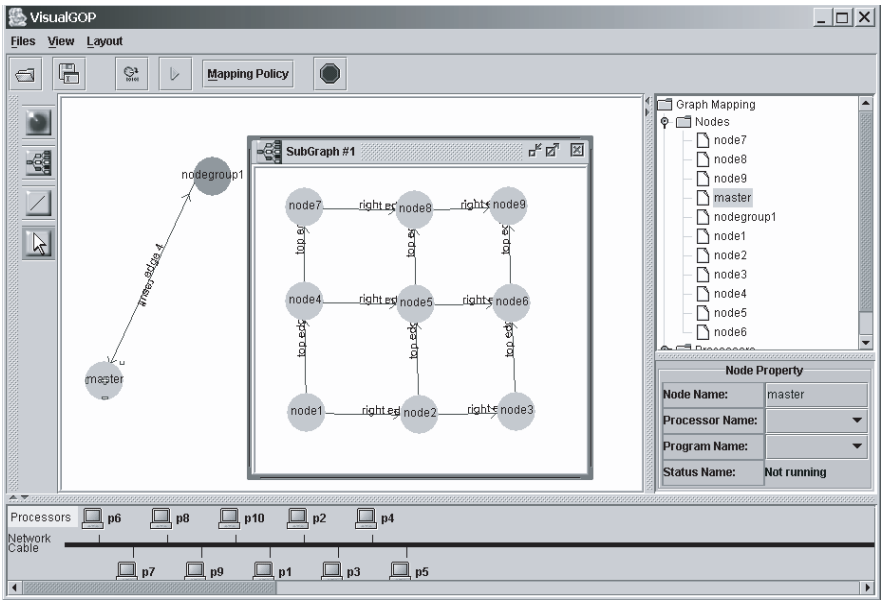

Figure 1.8 VisualGOP for parallel matrix multiplication.

onds using the function MPI Wtime(). Measurements were made by inserting instructions to start and stop the timers in the program code. The execution time of a parallel operation is the greatest amount of time required by all processes to complete the execution. We choose to use the minimum value from ten measurements.

Figure 1.9 shows the performance result in execution time. We can see that the MPI program runs slightly faster than the ClusterGOP program. This may be the result of conversion overheads (nodes to the rank ID) in the ClusterGOP library. However, there are no significant differences between MPI and ClusterGOP when the problem size and processor number are getting larger.

1.7 SUMMARY

In this chapter, we first described the GOP approach to providing high-level abstraction in message-passing parallel programming. The GOP model has the desirable features of expressiveness and simple semantics. It provides high-level abstractions for programming parallel programs, easing the expression of parallelism, configuration, communication, and coordination by directly supporting logical graph operations. Furthermore, sequential programming constructs blend smoothly and easily with parallel programming constructs in GOP. We then presented a programming environment called ClusterGOP that supports the GOP model. ClusterGOP supports both SPMD and MPMD programming models with various tools for the programmer to develop and deploy parallel applications. We showed the steps of programming parallel applications using ClusterGOP and reported the results of the evaluation on how ClusterGOP performs compared with the MPI. The results showed that ClusterGOP is as efficient as MPI in parallel programming.

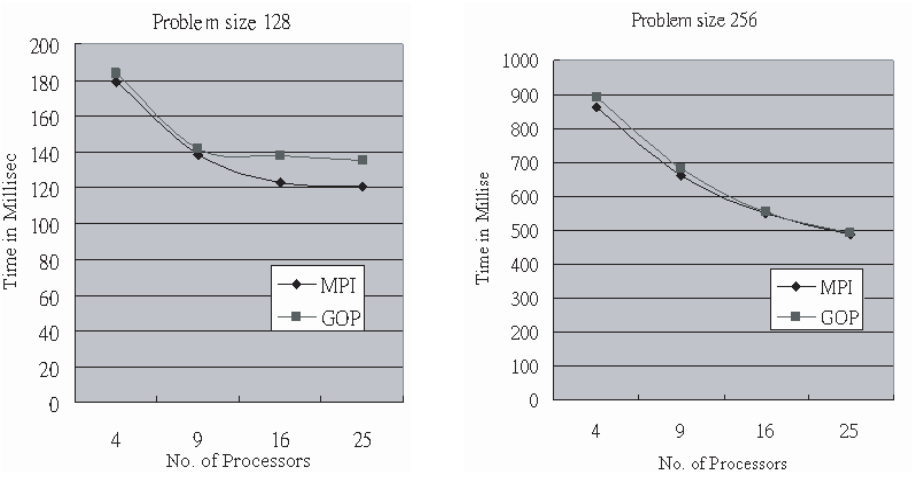

Figure 1.9 Execution time per input array for the parallel matrix multiplication application.

For our future work, we will enhance the current implementation with more programming primitives, such as update and subgraph generation. We will also define commonly used graph types as builtin patterns for popular programming schemes. Finally, we will develop real-world scientific and engineering computing applications using ClusterGOP.

ACKNOWLEDGMENTS

This work was partially supported by Hong Kong Polytechnic University under HK PolyU research grants G-H-ZJ80 and G-YY41.

REFERENCES

1. C. Anglano, R. Wolski, J. Schopf, and F. Berman. Developing heterogeneous applications using zoom and HeNCE. In *Proceedings of the 4th Heterogeneous Computing Workshop,* Santa Barbara, 1995.
2. D. Banerjee and J. C. Browne. Complete parallelization of computations: Integration of data partitioning and functional parallelism for dynamic data structures. In *Proceedings of 10th International Parallel Processing Symposium (IPPS '96),* pp. 354–360, Honolulu, Hawaii, 1996.
3. M. Besch, H. Bi, P. Enskonatus, G. Heber, and M.Wilhelmi. High-level data parallel programming in PROMOTER. In *Proceedings of 2nd International Workshop on High-level Parallel Programming Models and Supportive Environments,* pp. 47–54, Geneva, Switzerland, 1997.
4. J. Cao, L. Fernando, and K. Zhang. DIG: A graph-based construct for programming distributed systems. In *Proceedings of 2nd International Conference on High Performance Computing,* New Delhi, India, 1995.
5. J. Cao, L. Fernando, and K. Zhang. Programming distributed systems based on graphs. In M.A. Orgun and E.A. Ashcroft, (Eds.), *Intensional Programming I,* pp. 83–95. World Scientific, 1996.
6. F. Chan, J. Cao, A. T.S. Chan, and K. Zhang. Visual programming support for graph-oriented parallel/distributed processing. Accepted by *Software—Practice and Experiences, 2004,* 2002.
7. F. Chan, J. Cao, and Y. Sun. High-level abstractions for message-passing parallel programming. *Parallel Computing, 29*(11–12), 1589–1621, 2003.
8. F. Chan, A. T.S. Chan J. Cao, and M. Guo. Programming support forMPMDparallel computing in ClusterGOP. *IEICE Transactions on Information and Systems,* E87-D(7), 2004.
9. K. Chandy and C. Kesselman. Compositional C++: Compositional parallel programming. In *Proceedings of 6th International Workshop in Languages and Compilers for Parallel Computing,* pp. 124–144, 1993.
10. C. Chang, G. Czajkowski, and T. Von Eicken. MRPC: A high performance RPC system for MPMD parallel computing. *Software—Practice and Experiences, 29*(1), 43–66, 1999.

11. C. Chang, G. Czajkowski, T. Von Eicken, and C. Kesselman. Evaluating the performance limitations of MPMD communication. In *Proceedings of ACM/IEEE Supercomputing,* November 1997.
12. J. Y. Cotronis. Message-passing program development by ensemble. In *PVM/MPI 97,* pp. 242–249, 1997.
13. J. Y. Cotronis. Developing message-passing applications on MPICH under ensemble. In *PVM/MPI 98,* pp. 145–152, 1998.
14. J. Y. Cotronis. Modular MPI components and the composition of grid applications. In *Proceedings of the 10th Euromicro Workshop on Parallel, Distributed and Network-Based Processing 2002,* pp. 154–161, 2002.
15. D. Culler,A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumeta, T.Von Eicken, and K. Yelick. Parallel programming in Split-C. In *Proceedings of ACM/IEEE Supercomputing,* pp. 262–273, 1993.
16. I. Foster and K. Chandy. FORTRAN M: A language for modular parallel programming. *Journal of Parallel and Distributed Computing, 26*(1), 24–35, 1995.
17. I. Foster, C. Kesselman, and S. Tuecke. The nexus approach for integrating multithreading and communication. *Journal of Parallel and Distributed Computing, 37*(1), 70–82, 1996.
18. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V.S. Sunderam. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, Cambridge, MA, 1994.
19. A. Grimshaw. An introduction to parallel object-oriented programming with Mentat. Technical Report 91-07, University of Virginia, July 1991.
20. W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *MPI: The Complete Reference, volume 2, The MPI Extensions.* MIT Press, Cambridge, MA, 1998.
21. P.E. Hadjidoukas, E.D. Polychronopoulos, and T.S. Papatheodorou. Integrating MPI and nanothreads programming model. In *Proceedings of the 10th Euromicro Workshop on Parallel, Distributed and Network-Based Processing 2002,* pp. 309–316, 2002.
22. K. Johnson, M. Kaashoek, and D.Wallach. CRL: High-performance all-software distributed shared memory. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP),* pp. 213–226, 1995.
23. O. McBryan. An overview of message passing environments. *Parallel Computing, 20*(4), 417–447, 1994.
24. P. Newton. *A Graphical Retargetable Parallel Programming Environment and Its Efficient Implementation.* PhD thesis, University of Texas at Austin, Department of Computer Science, 1993.
25. P. Newton and J. Dongarra. Overview of VPE: A visual environment for messagepassing. In *Proceedings of the 4th Heterogeneous Computing Workshop,* 1995.
26. J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing, 10*(2), 197–220, 1996.
27. C. Scheurich and M. Dubois. Correct memory operation of cache-based multiprocessors. In *Proceedings of the 14th annual international symposium on Computer Architecture,* pp. 234–243. ACM Press, 1987.

28. V.Y. Shen, C. Richter, M.L. Graf, and J.A. Brumfield. VERDI: a visual environment for designing distributed systems. *Journal of Parallel and Distributed Computing, 9*(2), 128–137, 1990.

29. M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference.* MIT Press, Cambridge, MA, USA, 1996.

CHAPTER 2

The Challenge of Providing a High-Level Programming Model for High-Performance Computing

BARBARA CHAPMAN

2.1 INTRODUCTION

High-performance computing (HPC) is a critical technology that is deployed to design many engineering products, from packaging for electronics components to aeroplanes and their component parts. It is also needed in the design of computers themselves, and in safety testing of automobiles and reactors. It is used in modern drug development and in the creation of synthetic materials. It has helped us investigate global climate change and understand how to preserve rare books. In order to be able to solve these important problems with the required degree of accuracy, HPC codes must efficiently exploit powerful computing platforms. Since these are frequently the newest, and fastest, machines of the day, HPC application developers often become the pioneers who work with engineers and system software designers to identify and overcome a variety of bugs and instabilities. They learn how to write code for new kinds of architectures, and serve as the pilot users of new programming languages, libraries, compilers, and tools for application development and tuning. However, this reprogramming is labor-intensive and inefficient, and HPC applications experts are scarce. High-level programming standards are essential to reduce the human effort involved when applications target new platforms.

But providing a high-level standard for HPC application creation is a major challenge. At any given time, there are a variety of architectures for large-scale computing, with specific features that should be exploited to achieve suitable performance levels. HPC programmers are reluctant to sacrifice performance for ease of programming, and a successful standard must be capable of efficient implementation across these platforms. Moreover, there are additional constraints: sufficient understanding of the programming model is needed so that performance bugs can be

High-Performance Computing: Paradigm and Infrastructure. Edited by L. T. Yang and M. Guo

identified and strategies for overcoming them designed. Applications must be able to scale, at least in the sense that performance is sustained when both problem and machine size are increased. Finally, HPC is a relatively small market and vendors cannot support a multiplicity of programming models that target it, so a standard must be usable for a large fraction of HPC codes.

In this chapter, we briefly discuss recent HPC hardware technology and then consider attempts to define programming standards for the creation of such applications. We first review the programming models MPI and HPF, and the lessons that were learned from these significant standardization efforts. We then consider more recent models, with a particular focus on OpenMP, and discuss its strong points and problems. We then speculate on what is needed to obtain a useful high-level programming model for HPC. The final part of this chapter discusses a strategy for implementing OpenMP on clusters, which we refer to as DMPs, and show that it has great potential, but that a compiler-oriented approach remains a challenge. There is a mismatch between the traditional behavior of HPC applications developers, who are accustomed to the hard work of low-level, machine-specific programming, and the declared goal of high-level approaches that attempt to provide a reduction in development effort. In the future, there may not be the luxury of a choice and it is important that this technology be further improved.

2.2 HPC ARCHITECTURES

2.2.1 Early Parallel Processing Platforms

The 1990s saw the emergence of the first parallel programming "standards" for HPC, and hardware that was deployed at the time dictated the nature of these activities. At the start of the decade, vector supercomputing systems such as those marketed by Cray Research, Fujitsu, and NEC, were widely used to run large-scale applications. Two to four vector processors were combined to form particularly powerful systems with a single shared memory that were efficiently exploited by major numerical computations in weather forecasting, fluid dynamics, and more. Bus-based shared-memory parallel systems (SMPs), consisting of small numbers of RISC CPUs that share memory, were also available, but their uptake faltered when it became clear that it would be hard to extend this technology to larger CPU counts. Transputers had given rise to simple, low-cost clusters that gave some early adopters their first taste of inexpensive parallel computing, and a number of HPC researchers had migrated to the new distributed-memory parallel platforms (DMPs), produced by companies such as Intel, Meiko, and nCUBE. SIMD (single instruction, multiple data) computers (e.g., from Thinking Machines and Maspar), which execute an instruction on all the elements of an array simultaneously, were in use, and there had been experiments with cache-coherent platforms, notably by Kendall Square Research.

Vector systems were expensive to purchase and operate. As a result, they were usually operated by centers that serviced the needs of a number of paying cus-

tomers. DMPs could be built in many different sizes, from small to (in principle) arbitrarily large configurations, and so it was possible to build machines to suit the budgets of individual organizations. They appeared to be the most suitable basis for the massive parallelism envisioned by leaders of the day. As the technology stabilized, organizations increasingly procured their own DMP supercomputers. Thus, the use of distributed memory platforms, as well as the size of systems built, continued to grow steadily throughout the decade and their share of the HPC market rose accordingly [49]. However, vendor-supplied systems such as the Meiko CS2, Cray T3D and T3E, and IBM's SP architectures, were not the only DMPs in use throughout the 1990s. Such systems essentially consist of a number of individual computers, complete with CPU and memory, connected via a custom high-speed network and with system support to ensure that data is rapidly transported between the different memories. Similarly, workstations connected by a local-area network began to be used together to compute a single job, a trend that was greatly encouraged by the emergence of third-party software to simplify parallel job submission. They became known as clusters of workstations (COWs). Although the Ethernet networks used to build COWs were slow in comparison to the custom networks of true DMPs, they were much cheaper to build.

In the latter half of the 1990s, most U.S. hardware vendors began to produce SMPs. In contrast to those built in the late 1980s, these relatively inexpensive systems are intended for wide usage as desktop computers and powerful servers. With the tremendous increase in clock speed and size of memory and disk storage, some applications that formerly required HPC hardware could be executed on a single SMP. Perhaps more significantly, for the first time, parallel computers were accessible to a broad swathe of users.

2.2.2 Current HPC Systems

Today, vector supercomputers continue to provide highest levels of performance for certain applications, but remain expensive. SMPs are widely marketed, with CPU counts from two to over a hundred, as with the Sun Microsystems 6800 series. Most hardware vendors now combine DMP and SMP technologies to provide the cycles demanded by large-scale computations. As a result, they have two distinct levels of architectural parallelism.

COWs have become commonplace; clusters consisting of many thousands of processors are now deployed in laboratories as well as industry (e.g., for seismic processing). The cost of networking has decreased and its performance has increased rapidly, so that the performance penalty associated with early COWs is much reduced. Since COWs may be configured as clusters of SMPs, they no longer differ substantially from custom architectures in structure. The availability of low-cost COWs built on the open-source Linux operating system, commodity chips, and commodity networks is driving most hardware vendors to offer similar solutions, in addition to proprietary platforms. A recent survey of hardware vendors [17] showed that they expect the trend toward use of clustered SMPs for HPC to continue; systems of this kind already dominate the Top500 list.

However, there are interesting alternatives. Some HPC platforms enable code running on a CPU to directly access the entire system's memory, that is, to directly reference data stored on another SMP in the system. Some have global addressing schemes and others provide support for this in the network. The so-called ccNUMA (cache coherent nonuniform memory access) systems marketed by HP and SGI [46] transparently fetch data from remote memory when needed, storing it in the local cache and invalidating other copies of the same dataset as needed. They can be thought of as large virtual SMPs, although code will execute more slowly if it refers to data stored on another SMP (hence the nonuniform cost of access). The size of such machines has continued to grow as the technology matures, and they have become an important platform for HPC applications.

Current research and development in high-end architectures considers large-scale platforms with global memory and huge numbers of concurrently executing threads. Early platforms with these characteristics include IBM's BlueGene and the Cray Research Tera. In another recent development, hardware manufacturers are now producing chips that provide additional parallelism in the form of multiple cores and multiple simultaneously executing threads that share a single processor (e.g., Intel's hyperthreading). This introduces a different kind of shared-memory parallelism that will need to be exploited by HPC applications. No machine has thus far convincingly closed the "memory gap," despite the introduction of hierarchies of cache and mixtures of shared and private cache. The trend toward global memory and concurrently executing threads has intensified interest in shared- (global-) memory programming models.

The HPC market does not dominate the agenda of hardware vendors. Although it remains an important test bed for new technologies, its market share is declining. Further, even traditional HPC players have become price conscious, and many now exploit relatively inexpensive commodity technologies rather than custom HPC platforms. Thus, many CPUs configured in HPC systems today were designed for the mass market. The power of an HPC platform is derived solely from the exploitation of multiple CPUs. We now discuss the application programming interfaces (APIs) that have been created for this purpose and discuss their benefits and limitations.

2.3 HPC PROGRAMMING MODELS: THE FIRST GENERATION

Creating a parallel application that utilizes the resources of a parallel system well is a challenging task for the application developer, who must distribute the computation involved to a (possibly large) number of processors in such a way that each performs roughly the same amount of work. Moreover, the parallel program must be designed to ensure that data is available to a processor that requires it with as little delay as possible. This implies that the location of the data in the memory system must be carefully chosen and coordinated with the distribution of computation. Overheads for global operations, including start-up times, may be significant, especially on machines with many processors, and their impact must be minimized. The

programming models presented here and in the next section all involve the user in the process of distributing work and data to the target machine, but differ in the nature of that involvement.

During the infancy of parallel computing, researchers proposed a variety of new languages for parallel programming, but few experienced any real use. The major hardware vendors created programming interfaces for their own products. Although a standards committee met, with the goal of defining a portable API for shared memory programming [52], no agreement emerged. The providers of DMPs developed library interfaces for their systems that enabled applications to exchange data between processors (e.g., Intel's NX library). It was common practice for programmers to rewrite HPC applications for each new architecture, a laborious procedure that often required not only prior study of the target machine but also the mastery of a new set of language or library features for its programming.

The 1990s saw the first de facto standardization of parallel programming APIs. Vendor extensions for vector and SIMD machines were fairly well understood and commonalities were identified among the different sets of features provided. Several of these were standardized in the array extensions that are part of Fortran 90. Broad community efforts produced HPF and MPI and a subsequent vendor consortium defined OpenMP for parallel application development. The APIs that we discuss below have not (with the exception of a few features of HPF) been adopted by the relevant ISO committees, and thus are standards only in the sense that they have been endorsed by a significant community of hardware and compiler vendors and their users.

2.3.1 The Message Passing Interface (MPI)

When a DMP is used to execute an application (we include COWs implicitly in this category from now on), data stored in memory associated with a processor must be transferred via the system's network to the memory associated with another processor whenever the code executing on the latter needs it. The most popular solution to programming such machines gives full control over this data transfer to the user. In an analogy to human interaction, it is referred to as message passing; a "message" containing the required data is sent from the code running on one processor to the remote code that requires it. MPI, or Message Passing Interface, was developed by a group of researchers and vendors to codify what was largely already established practice when their deliberations began in 1993 [32]. It consists of a set of library routines that are invoked from within a user code written in Fortran or C to transfer data between the processes involved in a computation. MPI was designed for DMPs at a time when the use of SMP, SIMD, and vector technology had begun to decline. It was soon available on all parallel platforms of interest, in both public domain and proprietary implementations.

MPI realizes a *local* model of parallel computing, as the user must write program code for each participating processor, inserting MPI routines to send or receive data as needed. Most MPI programs are created according to the so-called SPMD (single program, multiple data) model, where the same code runs on each processor, parameterized by the MPI-supplied process identification number. Although it is possi-

ble to write an MPI program using few library routines, the API provides rich functionality that enables optimization of many common kinds of communication. It provides for modularity in program design, particularly via the notion of groups of processes and subsequently introduced features for parallel I/O. It is applicable to both Fortran and C.

Considerable work is involved in creating an MPI code, and the cost of program maintenance is high. The programmer must set up buffers, move communications in order to minimize delays, modify code to minimize the amount of data communicated, and decide when it is advantageous to replicate computations to save communication. New kinds of errors, such as deadlock and livelock, must be avoided; debugging is nontrivial, especially for high CPU counts. Unless exceptional care is taken, the selected data distribution is hardwired into the program code, as it determines the placement and content of communications. The most appropriate MPI implementation of a program may differ between platforms; it is not uncommon to see multiple MPI versions of an application, each tuned for a target system of interest.

MPI functionality was already partially implemented on most platforms of the day at the time of its introduction. Open-source implementations rapidly became available. Application developers found that they could understand the model and its performance characteristics, and that it had the generality to describe the parallelism required to solve a great variety of problems. Thus, it is not ease of programming, but good performance, coupled with wide availability of the API, that led to its popularity. Programming models that assume a distributed memory can be very reasonably implemented on shared-memory platforms also, and thus it remained applicable to virtually all HPC architectures, even after the reintroduction of SMPs into the marketplace. MPI versions of a large number of important applications have been developed and are in broad use. It remains the most widely used API for parallel programming.

2.3.2 High-Performance Fortran (HPF)

On an architecture in which the memory is physically distributed, the time required to access nonlocal data may be significantly higher than the time taken to access locally stored data, and the management of data assumes great importance. This was the insight that resulted in the design of High Performance Fortran (HPF) [21], the first effort to provide a high-level programming model for parallel computing that received broad support.

Many researchers and vendor representatives took an active part in the definition of HPF, which appeared at approximately the same time as MPI. The HPF Forum began meeting regularly from early 1992 to design an implicit parallel programming model, that is, a model in which the user provides information to the compiler so that the latter can generate the explicitly parallel code. The forum kept to a tight schedule, producing its first language definition in just over a year. As the name implies, the features were designed for use with Fortran, and in particular with Fortran 90.

HPF realizes a *global* model of parallel computing: there is one program with a single thread of control that will be translated into multiple processes by the com-

piler. The core of HPF consists of a set of compiler directives with which the user may describe how a program's data is to be distributed among the CPUs of a machine. The API also provides two forms of *parallel loop,* as well as intrinsic functions and library routines that permit the user to query the system (e.g., to obtain information on the number of executing processors) and to perform certain arithmetical computations. It supports interoperability via a so-called extrinsic interface that enables HPF code to invoke procedures written under a different API. The central constructs of HPF offer a wide variety of options for mapping the program's data structures to the target platform. The platform may be represented by one or more user-declared logical *processor arrays,* which serve to give them a shape (and possibly size) and enable arrays to be distributed to them. Arrays in the program may be directly distributed to processors or they may be aligned with a *template* in order to ensure that they are mapped in a coordinated fashion. Templates are logical structures that are introduced for this purpose only; they may not appear in executable statements. The initial HPF definition provided for block and cyclic distributions that are applied to dimensions of an array or template. Any or all dimensions of an array may be left undistributed, in which case all elements of the corresponding dimensions are mapped to all processors. HPF defines a *block distribution* that maps a contiguous segment of an array dimension to each target processor; segments are the same size, with the possible exception of the last one. The *cyclic distribution* maps segments of a user-specified length in round-robin fashion to the target processors. It is relatively easy to compute the location of a given array element from its indices and the distribution's specification, particularly for block distributions. It is also a simple matter to compute the set of array elements that are local to a given process. These are often referred to as data *owned* by that process, and the basic strategy for compiling HPF code is to generate the code to be computed by each processor according to the "*owner computes*" rule. Thus, in HPF the user-specified data distribution determines the work distribution.

The task of the compiler is to generate the explicitly parallel processes, one of which will run on each participating processor. It must allocate storage for local data, based upon the user-inserted directives as well as for copies of nonlocal data needed by a process, and must assign computations to each processor. Storage for nonlocal data cannot be efficiently created without knowledge of its extent; thus, notation was introduced to permit specification of the "*shadow*" region of an array. The default strategy for distributing the work is then derived from the data distribution as indicated above. This requires a translation of global data references into local and nonlocal data references. The compiler must moreover insert the necessary communications to satisfy nonlocal references. If a computation includes indirect referencing, some of the corresponding analysis must be performed at run time. It is essential for performance that the communications be optimized to reduce their impact on performance, typically by aggregating them and starting them as early as possible to permit data transfer to be overlapped with computation. Separate compilation of procedures results in the need to specify mappings for formal procedure parameters and a costly reorganization of data is liable to be incurred whenever it does not verifiably match the distribution of the corresponding actual arguments.

The HPF translation process is complex and easily impaired if sufficient information is not available to do one of these tasks well. For instance, if it is not clear that a communication statement can be removed from an inner loop and combined with other data transfers, the resulting code may spend too much time transferring data. Run-time testing may alleviate some of the problems associated with lack of information, but in some cases, severe performance degradation may result from an inability to perform optimizations in the compiler.

HPF requires extensive program analysis and, in particular, accurate data dependence analysis to perform this translation. It also requires a full set of loop transformations to improve the translation process. For example, loop interchange might permit communications to be fused. If the data defined by two statements in a loop nest have different owners, then loop distribution might be able to split them into multiple loops, for which individual iterations can be assigned to processes without conditionals, otherwise, each of these statements would be conditionally executed in every iteration and the evaluation of the conditional would increase the overhead of the translation.

The HPF effort was preceded by a number of research projects that had the ambitious goal of enabling application developers to design programs for DMPs in much the same way as they were accustomed to on a sequential machine (e.g. [11]), and by several prototypes that demonstrated some of the compiler technology needed for their implementation (e.g. [15, 20]) . The standardization activity sparked off a good deal more research into a variety of compilation techniques that are needed to support any high-level parallel programming API. In addition to further work on loop transformations and their application, the benefits of interprocedural analysis for this and other complex translations became apparent. Fortran array reshaping and the passing of noncontiguous data, such as a strided array section, at procedure boundaries cause difficulties that have parallels in other contexts. Realignment and redistribution may occur implicitly at procedure boundaries, and a variety of efforts worked on the problem of optimizing the required all-to-all data exchange, which is particularly expensive when cyclic distributions are involved. Other activities aimed to reduce the implementation cost of reduction operations. Work to translate the HPF independent loop, the iterations of which are spread among the processes, required that the nonlocal data accesses be determined and the data fetched prior to the loop's execution. A strategy was developed to handle indirect array references, for which these datasets cannot be determined at compile time. Referred to as the *inspector–executor* strategy [24, 39], it essentially consists of a new "inspector" loop that partially executes the loop's code to determine the required data, then determines which elements are nonlocal and communicates with other processes to exchange information on the required elements, and, finally, exchanges the actual data. Only then is the original loop executed. This strategy is prohibitively expensive and is only appropriate if the constructed communication sets can be reused. The problem it attempts to solve is incurred in other high-level models, particularly when arrays are indirectly referenced.

The HPF Forum created a powerful and extensive programming API that was intended to be an easy means for creating parallel code. It led to much progress in compiler technology for DMPs and for parallel systems in general. Yet the introduction

of data distributions into a sequential program is not sufficient to enable the compiler to achieve consistently efficient code. At the time HPF was introduced, the cost of exchanging data between processes on a DMP was extremely high and the penalty for a compiler's insertion of excessive communications was significant. Moreover, as a result of the complexity of the compilation process and the need to develop Fortran 90 compiler technology concurrently, HPF products were slow to emerge.

The task of modifying a program to perform well under HPF was poorly understood, and the apparent simplicity was deceptive. There were exceptional penalties for "getting it wrong." It required a great deal of insight into the HPF translation process and a commitment that few were willing to bring to a new and relatively untested technology. HPF compilers had different strengths and weaknesses. Performance results and experiences were inconclusive and sometimes contradictory. Moreover, the stringent time constraints adhered to by the HPF Forum did not permit HPF1.0 to adequately address the needs of unstructured computations; these were of great importance at the time in the HPC community. Although HPF2.0 introduced generalized block distribution and indirect array distributions, these came much later and the latter proved to be very hard to implement. Finally, HPF did not address the needs of the growing fraction of HPC codes that were written in C.

MPI implementations were available by the time HPF compilers emerged, and they applied a technology that was already successfully deployed in HPC. Thus, HPF did not enjoy the success achieved by MPI despite the conveniences it provided (cf. Figure 2.1). Nevertheless, work on this API led to an upsurge of research into a variety of compiler optimizations and provided compiler technology that is useful for the realization of other high-level parallel programming APIs.

```
!HPF$ DISTRIBUTE (*,BLOCK)::A,B
!HPF$ SHADOW(0,1:1):: A
!HPF$ INDEPENDENT
      do j = 2, SIZE-1
        do i = 2, SIZE-1
          a(i, j) = (b(i - 1, j) + b(i + 1, j) + b(i, j - 1) +
b(i, j + 1)) / 4
        enddo
      enddo
!HPF$ INDEPENDENT
      do j = 2, SIZE-1
        do i = 2, SIZE-1
          b(i, j) = a(i, j)
        enddo
      enddo
!HPF$ INDEPENDENT REDUCTION(sum)
      do j = 1,SIZE
        do i = 1,SIZE
          sum = sum + b(i,j)
        end do
      end do
```

(a)

```
Double precision a(0:SIZE,SIZE/nproc+1), b(0:SIZE,SIZE/nproc+1)
Call MPI_INIT()
Call MPI_Comm_size(MPI_COMM_WORLD, nproc, ierr)
Call MPI_Comm_rank(MPI_COMM_WORLD, myid, ierr)
do j = 2, SIZE/nproc-1
  do i = 2, SIZE-1
     a(i, j) = (b(i - 1, j) + b(i + 1, j) + b(i, j - 1) +b(i, j + 1)) / 4
  enddo
enddo
If (myid .lt. npes-1) Send my last column to my east neighbor myid+1,
If (myid .ne. 0)      Receive a column from my west neighbor myid-1
If (myid .ne. 0)      Sends my first column to west neighbor myid-1
If (myid .ne. npes-1)  Receives data from my east neighbor myid+1
do j = 2, SIZE/nproc-1
    do i = 2, SIZE-1
       b(i, j) = a(i, j)
    enddo
 enddo
 do j = 1,SIZE/nproc
    do i = 1,SIZE
       sum = sum + b(i,j)
    end do
  end do
Call MPI_Reduce(…)
Call MPI_FINALIZE(ierr)
```

(b)

Figure 2.1 (a) Jacobi HPF program and (b) Jacobi pseudo-MPI code.

2.4 THE SECOND GENERATION OF HPC PROGRAMMING MODELS

The next generation of programming models for HPC has focused on shared memory. On a shared memory platform, the main goal is to perform an even distribution of the work in a program to the executing processors. The relatively low penalty incurred when accessing nonlocal data on DSM systems has led to notable successes in the deployment of shared-memory APIs on these platforms too. A shared-memory programming style has also been realized on DMPs. The shared-memory model is generally considered to be more user-friendly than its distributed-memory counterparts.

2.4.1 OpenMP

Both HPF and MPI were introduced to support application development on DMPs. Yet shortly thereafter, SMPs were being sold in large quantities. OpenMP was introduced to fulfill the need for an API for this class of platforms [37]. First introduced with an interface for Fortran late 1997 by an influential group of computer vendors, it was extended by C and C++ versions just a year later. The OpenMP programming model is based upon earlier attempts to provide a standard API for shared-memory parallel programming and was thus already familiar to many in the HPC community; it primarily supports the parallelization of loops needed for many scientific and engineering computations. It is much easier to parallelize unstructured computations under OpenMP than under either of the above-mentioned APIs, since there is no need to partition unstructured data.

Like HPF, OpenMP is an implicit parallel programming model, consisting of a set of compiler directives and several library routines, that realizes a global model of parallel computing. It is based upon the fork–join execution model, in which a single master thread begins execution and spawns a team of worker threads to perform computations in parallel regions. It is the task of the user to specify the parallel regions and to state how the work is to be shared among the team of threads that will execute them. The directives enable the declaration of such regions, the sharing of work among threads, and their synchronization. Loop iterations may be statically or dynamically distributed in blocks or in cyclic fashion, array computations distributed, and work divided into a set of parallel sections. Parallel regions may differ in the number of threads assigned to them at run time. The user is responsible for detecting and dealing with race conditions in the code, as well as for thread coordination. Critical regions and atomic updates may be defined; variable updates may be ordered, or a set of statements executed by the first thread to reach them only. The OpenMP library implements locks. An innovative feature of OpenMP is the inclusion of orphan directives, directives that are dynamically, rather than lexically, within the scope of a parallel region. They greatly simplify the code migration process, since there is no need to extract code from subprograms for worksharing, and they enable large parallel regions to be created. OpenMP permits private variables, with potentially different values on each thread, within parallel regions. Other data is shared. Fortran programs may have thread-private common blocks, and

private variables may be SAVEd. Since OpenMP is a shared memory model, sequence and storage association is not affected by its directives. A conditional compilation feature facilitates the maintenance of a single source code for both sequential and parallel execution.

The basic compilation strategy for OpenMP is relatively simple, as the language expects the application developer to indicate which loops are parallel and, therefore, this analysis is not required. No communications need be inserted. The main focus is on the generation of thread-specific code to implement the parallel regions. Most often, parallel regions are converted to procedures that are then adapted via the insertion of calls to a thread library such as pThreads [33]. Since iterations are distributed, there is no need for compiler analysis to determine the locus of computation.

OpenMP can be used in several different ways. The straightforward approach does not require the user to analyze and possibly modify an entire application, as is the case with both HPF and MPI. Instead, each loop is parallelized in turn. Although this does require the user to deal with data dependencies in small regions of a program, it is generally considered appropriate for nonexpert application developers [5]. If the implicit barriers after many constructs are not eliminated where possible, they may degrade performance. Problems arise when data being updated is stored in more than one cache of an SMP and if attention is not paid to this, performance gains may be moderate only. Even nonexpert users may need to learn about cache. However, the requirements of HPC applications generally necessitate a different strategy for using OpenMP, which relies on the user to explicitly distribute work and data among the threads that will execute it. Thread-private data structures are created to hold local data and shared-data structures are only declared to hold data needed by more than one thread. Since OpenMP provides thread identification, work can be explicitly allocated to threads as desired. The strategy minimizes interactions between threads and can yield high performance and scalability [50]. Although fewer details need be specified than with MPI, this so-called SPMD-style OpenMP presupposes a more extensive program modification, but it is likely to be easier to maintain than a corresponding MPI code, since fewer lines are added.

The OpenMP developers did not present the specification until compilers were nearly ready, in an effort to avoid the frustrations caused by the slow arrival of HPF compilers. Tools quickly became available for OpenMP application development, debugging, and performance analysis (e.g. [26, 38]). This, its applicability to all major HPC programming languages, the relative ease of its use, its support for single-source as well as incremental-application development facilitated its rapid adoption as the de facto standard for shared memory parallel programming by the remaining hardware vendors and the community.

However, the current language was primarily designed to facilitate programming of modest-sized SMPs, and OpenMP parallel programs cannot be executed on DMPs. Many codes executed on large clusters of SMPs are programmed using a combination of MPI and OpenMP, such that MPI performs synchronization and data transfer takes place between the different SMPs, whereas OpenMP is used to

program within each SMP. Where more than one kind of parallelism is naturally present in an application, this is a strategy that can work well. Unfortunately, the easiest way to develop such a program, particularly if development begins with an MPI code, is to create OpenMP parallel regions between MPI library calls. This is not an efficient way to use OpenMP, and performance results can be disappointing. Several hybrid strategies are discussed in [9, 25, 48].

2.4.1.1 OpenMP on DSM Platforms. OpenMP is also implementable on ccNUMA systems. The threads that execute a parallel region may run on processors across multiple nodes (e.g. [28]) of the machine. When a loop is parallelized via a worksharing construct, for instance, loop iterations are executed on threads distributed across the system, and the program data may also be spread across the different node memories. It is likely that a thread will reference data stored in nonlocal memory. Unfortunately, large numbers of remote memory accesses can lead to significant network traffic and substantial delays.

In order to obtain high performance from ccNUMA systems [44], coordinated placement of data and computations over the memories is needed [7, 18, 43] to ensure that the data accessed by each thread is largely local to the processor on which that thread is currently executing. OpenMP does not provide any mechanisms for controlling the placement of data relative to computation, so vendors have supplied means by which this may be influenced. One of these is a default "first touch" strategy that allocates data in memory local to the first thread that accesses it; if the application developer carefully distributes the initial data references across threads, this simple mechanism may suffice. An alternative is to automatically migrate pages of data so that they are near to threads that reference them. This is appealing, as it is transparent to the user. However, pages of memory are fairly large, and without application insight it can be hard for the system to determine when it is appropriate to move data. Thus, more research into this area is needed [36]. Finally, both HP and SGI have provided custom extensions to OpenMP to permit the user to express the data locality requirements of an application (e.g. [6, 12, 13, 47]), although it is not easy to provide these features in an API that permits incremental program development. The basic approach embodied by HPFD has been used, but page-based data distributions have also been included. Despite considerable overlap of functionality, the syntax and a number of details differ substantially in these vendor-specific features. Partly as a result of this, they are not widely used; most application developers rely on "first touch" allocation and privatization of data, although the specification of thread affinity appears to be a useful concept. SPMD-style codes on DSM platforms have been shown to outperform MPI and to scale better, so that the difference increases with large CPU counts [50].

2.4.2 Other Shared-Memory APIs

Other APIs have been developed recently that are important to some sectors of the community. One of these is Global Arrays (GA), which applies a shared-memory programming model to DMPs, and another is Unified Parallel C (UPC), which fo-

cuses on providing support for DSM platforms. The latter in particular introduces several ideas that may influence future programming models for HPC.

2.4.2.1 Global Arrays (GA). GA is a library [34] that was designed to simplify the programming methodology on DMPs by providing a shared-memory abstraction. It provides routines that enable the user to specify and manage access to shared data structures, called *global arrays,* in a FORTRAN, C, C++, or Python program. A GA program consists of a collection of independently executing processes, each of which is thus able to access data declared to be shared without interfering with other processes. Programmers can write their parallel program using this *local* model as if they have shared-memory access, specifying the layout of shared data at a higher level. GA permits the user to specify block-based data distributions corresponding to HPF BLOCK and GEN_BLOCK distributions, which map identical-length chunks and arbitrary-length chunks of data to processes, respectively. *Global arrays* are accordingly mapped to the processors executing the code. Each GA process is able to independently and asynchronously access these distributed data structures via *get* or *put* routines. GA combines features of both message-passing and shared-memory programming models, leading to both simple coding and efficient execution. The GA programming model forces the programmer to determine the needed locality for each phase of the computation. By tuning the algorithm to maximize locality, portable high performance is easily obtained. GA has been widely implemented and has been used to develop several HPC applications.

2.4.2.2 Unified Parallel C (UPC). UPC [8, 14] is an extension of C that aims to support application development on DSMs or other systems by providing for a global address space and multithreaded programming. In contrast to OpenMP, it provides data mappings and expects the programmer to consider and express data locality explicitly. UPC code is *global,* and roughly equivalent to SPMD-style OpenMP code in that it typically combines some shared data with largely local computations. Arrays and pointers may be declared as shared and given an HPF-like data distribution; all other data objects are private. A parallel loop is defined, the iterations of which are assigned to threads by the compiler rather than via a user-specified loop schedule. An interesting feature of UPC is that it allows the user to specify whether strict or relaxed consistency is to be applied to memory references, whereby the former refers to sequential consistency and the latter requires a thread-local consistency only, so that computations may be reorganized so long as there are no dependencies within that thread (only) to prevent it. Threads execute asynchronously unless the user explicitly inserts a fence, barrier, or a split barrier that permits the execution of local code while waiting for other threads to reach the same point in their execution. UPC can be implemented on systems that provide support for asynchronous read and write operations, such that a processor is able to independently fetch nonlocal data. It is thus more easily implemented on DMPs than OpenMP is, but lacks the simpler development approach, including incremental development, and does not support single-source operation. Implementations of UPC exist on several platforms. *Co-Array Fortran* (CAF) is often considered to be

the Fortran equivalent to UPC. This is not altogether true; CAF does not provide global data and provides a *local* model of computation.

2.4.3 Is A Standard High-Level API for HPC in Sight?

Although MPI is a de facto standard for DMPs, it is error-prone and, above all, time-consuming. The HPC community is small and its ability to sustain even a single specific high-level programming model has been questioned. But the importance of a parallel programming API for DMPs that facilitates programmer productivity is increasingly recognized. If such an API is to be broadly used in HPC, it must be expressive enough to allow application developers to specify the parallelism in their applications, it must be capable of efficient implementation on most HPC platforms, and it must enable users to achieve performance on these machines. OpenMP is designed for shared-memory systems and emphasizes usability, but does not provide similar levels of portability. However, it enjoys strong and ongoing vendor support and for this reason it is currently the best candidate for developing a truly portable high-level API. Moreover, it may also facilitate exploitation of chip multithreading and multicore systems. It potentially has a much larger market since it may satisfy the needs of a range of applications that do not necessarily belong to HPC but can exploit single SMPs.

But there are a number of challenges that must be met if OpenMP is to become a widely used API in the HPC community. Unlike MPI, which can be implemented on SMPs with reasonable efficiency, OpenMP is hard to realize efficiently on DMPs. The current approach is to implement OpenMP on top of software that provides virtual shared memory on a DMP, a so-called software-distributed shared-memory (DSM) system, such as Omni [40, 42] or TreadMarks [1, 22]. Although this technology does not yet provide the performance levels required for HPC applications, it is the subject of a good deal of ongoing work and a production version of TreadMarks is expected to appear soon. Other proposals have been made that extend OpenMP to DMPs [31].

OpenMP is not without other difficulties. One of these is the relative ease with which erroneous codes can be produced. It is up to the application developer to insert all necessary synchronization to coordinate the work of the independently executing threads. In particular, this requires the error-prone, manual recognition of data dependencies and the correct ordering of accesses. The power of OpenMP lies in the ease with which large parallel regions may be created and these may span many procedures, complicating this task. Although the solution may lie in tool support rather than language modification, it is an issue that requires attention.

OpenMP currently supports only the simplest expression of hierarchical parallelism [2]. There is no facility for the distributed execution of multiple levels of a loop nest in parallel within a single parallel region, let alone the ability to assign work to threads within the same SMP. Yet being able to exploit this hierarchy could be critical to achieving scalable performance. This might be realized by the provision of means to indicate how resources should be allocated to nested parallel regions, possibly with some architectural description. Other challenges include the

need to provide additional features that support the creation of codes for very large platforms. Starting up threads across an entire machine, for example, can become very costly, as can the allocation of shared data and global operations such as reductions. There is currently no concept corresponding to the MPI process groups, and this is certainly needed. Unstructured computations will require more language and compiler support for efficient execution across large numbers of SMPs. Some applications require synchronization that is hard to express in the current OpenMP language [45]. On the other hand, a compiler could, in some circumstances, automatically determine which variables can be safely privatized in a parallel region to relieve the user of this task. A number of proposals exist for extending OpenMP in these and other ways (e.g. [3, 27, 29, 36]).

A slight relaxation of the semantics of OpenMP might also improve its performance for some applications. The current standard requires that the user state that there are no dependencies between a pair of consecutive parallel loops. Otherwise, the compiler will ensure that one loop has been completed in its entirety via an expensive barrier before the next loop can be computed. But, in reality, it is often possible do better than that and languages such as UPC provide finer grain synchronization. A more flexible OpenMP translation strategy [51] might determine the dependencies between the sets of loop iterations that will be executed by a single thread. At run time, an iteration set may then be executed as soon as those it depends on have completed. This replaces many of the barriers in an OpenMP code by synchronization between pairs of threads.

2.5 OPENMP FOR DMPS

We now turn to the most challenging of the problems facing OpenMP, that of extending it for execution on DMPs. The most straightforward approach to do so relies on the memory management capabilities of software DSMs to handle a code's shared data. Under this approach, an OpenMP program does not need to be modified for execution on a DMP; the software DSM is responsible for creating the illusion of shared memory by realizing a shared address space and managing the program data that has been declared to be shared. However, there are inherent problems with this translation. Foremost among these is the fact that the management of shared data is based upon pages, which incurs high overheads. Software DSMs perform expensive data transfers at explicit and implicit barriers of a program, and suffer from false sharing of data at page granularity. They typically impose constraints on the amount of shared memory that can be allocated. One effort [16] translates OpenMP to a hybrid MPI and software DSM in order to overcome some of the associated performance problems. This is a difficult task, and the software DSM could still be a performance bottleneck. Much better performance can be obtained if the user creates an SPMD-style OpenMP code prior to this parallelization, in order to minimize the shared data. Array padding may then help reduce some of the memory conflicts. Few, if any, language extensions are needed to apply this translation.

In the following, we describe an alternative approach to translating OpenMP to DMPs in order to provide some insight into this translation. Its strong point is that the compiler inserts explicit communications and is thus able to control the data that is transferred as well as the point of transfer. The weakness of the approach is that it is rather complex, introducing some of the difficulties that proved problematic for HPF, albeit with lower penalties. A major challenge is the need to determine a data distribution for shared data structures. It relies on a translation from OpenMP to Global Arrays, described above [23].

2.5.1 A Basic Translation to GA

It is largely straightforward to translate OpenMP programs into GA programs because both have the concept of shared data and the GA library features match most OpenMP constructs. However, before the translation occurs, it is highly beneficial to transform OpenMP code into the so-called SPMD style [30] in order to improve data locality. The translation strategy [23] follows OpenMP semantics. OpenMP threads correspond to GA processes and OpenMP shared data are translated to global arrays that are distributed among the GA processes; all variables in the GA code that are not global arrays are called private variables. OpenMP private variables will be translated to GA private variables as will some OpenMP shared data. OpenMP scalars and private variables are replicated to each GA process. Small or constant shared arrays in OpenMP will also be replicated; all other shared OpenMP arrays must be given a data distribution and they will be translated to global arrays.

Shared pointers in OpenMP must also be distributed. If a pointer points to a shared array, it must be determined whether the contents of the current pointer are within the local portion of the shared memory. Otherwise, *get* or *put* operations are required to fetch and store nonlocal data and the pointer will need to point to the local copy of the data. If a shared pointer is used directly, an array can be substituted for it and distributed according to the loop schedule because GA does not support the distribution of C pointers.

The basic translation strategy assigns loop iterations to each process according to OpenMP static loop schedules. For this, the iteration sets of the original OpenMP parallel loops must be calculated for each thread. Furthermore, the regions of shared arrays accessed by each OpenMP thread when performing its assigned iterations must be computed. After this analysis, we determine a block-based data distribution and insert the corresponding declarations of global arrays. The loop bounds of each parallel loop are modified so that each process will work on its local iterations. Elements of global arrays may only be accessed via *get* and *put* routines. Prior to computation, the required global array elements are gathered into local copies. If the local copies of global arrays are modified during the subsequent computation, they must be written back to their unique "global" location after the loop has completed. GA synchronization routines replace OpenMP synchronization to ensure that all computation as well as the communication to update global data have completed before work proceeds.

An example of an OpenMP program and its corresponding GA program is given in Figure 2.2. The resulting code computes iteration sets based on the process ID. Here, array *A* has been given a block distribution in the second dimension, so that each processor is assigned a contiguous set of columns. Nonlocal elements of global array *A* in Figure 2.2(b) are fetched using a *get* operation followed by synchronization. The loop bounds are replaced with local ones. Afterward, the nonlocal array elements of *A* are put back via a *put* operation with synchronization.

Unfortunately, the translation of synchronization constructs (CRITICAL, ATOMIC, and ORDERED) and sequential program sections (serial regions outside parallel regions, OpenMP SINGLE and MASTER) may become nontrivial. GA has several features for synchronization that permit their expression; however, the translated codes may not be efficient.

2.5.2 Implementing Sequential Regions

Several different strategies may be employed to translate the statements enclosed within a sequential region of OpenMP code, including I/O operations, control flow constructs (IF, GOTO, and DO loops), procedure calls, and assignment statements. A straightforward translation of sequential sections would be to use exclusive master process execution, which is suitable for some constructs including I/O operations. Although parallel I/O is permitted in GA, it is a challenge to transform OpenMP sequential I/O into GA parallel I/O. The control flow in a sequential region must be executed by all the processes if the control flow constructs enclose or are enclosed by any parallel regions. Similarly, all the processes must execute a procedure call if the procedure contains parallel regions, either directly or indirectly. The properties of the data involved are used to determine the different GA execution

```
1 !$OMP PARALLEL
SHARED(a)
2  do k = 1 , MAX
3 !$OMP DO
4   do j = 1 , SIZE_J ( j )
5     do i = 2 , SIZE-1
6         a(i,j)= a(i-1,j)+a(i+1,j)+ …
7     enddo
8   enddo
9 !$OMP END DO
10 enddo
11 !$OMP END PARALLEL
```

```
1 OK=ga_create (MT_DBL, SIZE_X , SIZE_Y , 'A' ,
2                       SIZE_X, SIZE_Y/ nproc , g_a )
3 do k = 1 , MAX
4 ! compute new lower bound and upper bound for each process
5 (new_low, new_upper) = …
6 ! compute the remote array region read for each thread
7 (jlo, jhi)=  …
8 call  ga_get ( g_a, 1, SIZE , jlo , jhi , a, ld )
9 call ga_sync ( )
12 do j = new_low , new_upper
13    do i = 2 , SIZE-1
14       a (i, j) = a(i-1, j) +a(i+1,j)…
15    enddo
16 enddo
18 ! compute remote array region written
19 call ga_put ( g_a , 1 , SIZE , jlo , jhi , a , ld)
20 call ga_sync ( )
21 enddo
```

Figure 2.2 (a) An OpenMP program and (b) the corresponding GA program.

strategies chosen for an assignment statement in sequential parts of an OpenMP program:

1. If a statement writes to a variable that will be translated to a *GA private* variable, this statement is executed redundantly by each process in a GA program; each process may fetch the remote data that it will read before executing the statement. A redundant computation can remove the requirement of broadcasting results after updating a GA private variable.
2. If a statement writes to an element of an array that will be translated to a global array in GA (e.g., S[i]=. . .), this statement is executed by a single process. If possible, the process that owns the shared data performs the computation. The result needs to be put back into the "global" memory location.

Data dependences need to be maintained when a global array is read and written by different processes. To ensure this, synchronization may be inserted after each write operation to global arrays during the translation stage; at the code optimization stage, we may remove redundant *get* or *put* operations, and aggregate communications of neighboring data if possible.

2.5.3 Data and Work Distribution in GA

GA provides simple block-based data distributions only and supplies features to make them as efficient as possible. There are no means for explicit data redistribution. GA's asynchronous one-sided communication transfers the required array elements, rather than pages of data, and it is optimized to support the transfer of sets of contiguous or strided data, which are typical for HPC applications. These provide performance benefits over software DSMs. With block distributions, it is easy to determine the location of an arbitrary array element. However, since these distributions may not provide maximum data locality, they may increase the amount of data that needs to be gathered and scattered before and after execution of a code region, respectively. In practice, this tends to work well if there is sufficient computation in such code regions. GA only requires that the regions of global arrays that are read or written by a process be calculated to complete the communication; GA handles the other details. It is fast and easy for GA to compute the location of any global data element. We may optimize communication by minimizing the number of *get/put* operations and by merging small messages into larger ones.

It is the task of the user to specify the distribution of global data in a GA program; when converting OpenMP programs to the corresponding GA ones, the compiler must, therefore, choose the distribution. A suitable strategy for doing so may be based upon simple rules such as the following:

1. If most loop index variables in those loop levels immediately enclosed by PARALLEL DO directives sweep over the same dimension of a shared array in an OpenMP program, we perform a one-dimensional distribution for the corresponding array in this dimension.

2. If different dimensions of a shared array are swept over almost evenly by parallel loops, we may perform multidimensional distribution for this array.
3. If parallel loops always work on a subset of a shared array, we may distribute this shared array using a GEN_BLOCK distribution; otherwise, a BLOCK distribution may be deployed. In the former case, the working subset of the shared array is distributed evenly to each thread; the first and last threads will be assigned any remaining elements of arrays at the start and end, respectively.

This translation could be improved with an interactive tool that collaborates with the user. Another possible improvement would be to perform data distribution based on the most time-consuming parallel loops. Statically, we may estimate the importance of loops. However, user information or profile results, even if based on a partial execution, are likely to prove much more reliable. A strategy that automates the instrumentation and partial execution of a code with feedback directly to the compiler might eliminate the need for additional sources of information.

OpenMP static loop scheduling distributes iterations evenly. When the iterations of a parallel loop have different amounts of work to do, dynamic and guided loop scheduling can be deployed to balance the workload. It is possible to translate all forms of the OpenMP loop schedule. However, the GA code corresponding to dynamic and guided-loop scheduling may have unacceptable overheads, as it may contain many *get* and *put* operations transferring small amounts of data. Other work distribution strategies need to be explored that take data locality and load balancing into account.

For irregular applications, it may be necessary to gather information on the global array elements needed by a process; whenever indirect accesses are made to a global array, the elements required in order to perform its set of loop iterations cannot be computed. Rather, an inspector–executor strategy is needed to analyze the indirect references at run time and then fetch the data required. The resulting datasets need to be merged to minimize the number of required *get* operations. We enforce static scheduling and override the user-given scheduling for OpenMP parallel loops that include indirect accesses. The efficiency of the inspector–executor implementation is critical. In a GA program, each process can determine the location of data read/written independently and can fetch it asynchronously. This feature may substantially reduce the inspector overhead compared with HPF or any other paradigm that provides a broader set of data distributions. Here, each process independently executes an inspector loop to determine the global array elements (including local and remote data) needed for its iteration set and their locations. The asynchronous communication can be overlapped with local computations, if any.

2.5.4 Irregular Computation Example

FIRE™ is a fully interactive fluid-dynamics package for computing compressible and incompressible turbulent fluid. *gccg* is a parallelizable solver in the FIRE™

Benchmarks [4] that uses orthomin and diagonal scaling. It is used here to illustrate how to translate codes with irregular data accesses to GA.

Figure 2.3 displays the most time-consuming part of the *gccg* program. In the approach described above, we perform array region analysis to determine how to handle the shared arrays in OpenMP. Shared arrays *BP, BS, BW, BL, BN, BE, BH, DIREC2,* and *LCC* are privatized during the initial compiler optimization to improve locality of OpenMP codes, since each thread performs work only on an individual region of these shared arrays. In the subsequent translation, they will be replaced by GA private variables. In order to reduce the overhead of the conversion between global and local indices, global indices may be preserved for the list of arrays above when declaring them and we may allocate memory for array regions per process dynamically if the number of processes is not a constant. Shared array *DIREC1* is distributed via global arrays according to the work distribution in the two parallel do loops in Figure 2.3. A subset of array *DIREC1* is updated by all threads in the first parallel loop; the second parallel loop accesses *DIREC1* indirectly via *LCC*. We distribute *DIREC1* using a GEN_BLOCK distribution according to the static loop schedule in the first parallel loop in order to maximize data locality, as there is no optimal solution of data distribution for *DIREC1* in the second loop. The array region *DIREC1*[*NINTC1*:*NINTCF*] is mapped to each process evenly in order to balance the work. Since *DIREC1* is declared as [1:*N*], the array regions [1:*NINTC1*] and [*NINTCF*:*N*] must be distributed as well. We distribute these two regions to process 0 and the last process for contiguity, thus using a GEN_BLOCK distribution (as in HPF) as shown in Figure 2.4, assuming four processors are involved.

As before, work distribution is based on the OpenMP loop schedule. In the case in which all data accesses will be local (the first loop of Figure 2.3), loop iterations

```
!$OMP PARALLEL
        DO I = 1, iter
!$OMP DO
        DO 10 NC=NINTCI,NINTCF
          DIREC1(NC)=DIREC1(NC)+RESVEC(NC)*CGUP(NC)
    10 CONTINUE
!$OMP END DO
!$OMP DO
         DO 4 NC=NINTCI,NINTCF
           DIREC2(NC)=BP(NC)*DIREC1(NC)
      X          - BS(NC)  * DIREC1(LCC(1,NC))
      X          - BW(NC) * DIREC1(LCC(4,NC))
      X          - BL(NC)  * DIREC1(LCC(5,NC))
      X          - BN(NC)  * DIREC1(LCC(3,NC))
      X          - BE(NC)  * DIREC1(LCC(2,NC))
      X          - BH(NC) * DIREC1(LCC(6,NC))
    4  CONTINUE
!$OMP END DO
    END DO
!$OMP END PARALLEL
```

Figure 2.3 An OpenMP code segment in *gccg* with irregular data accesses.

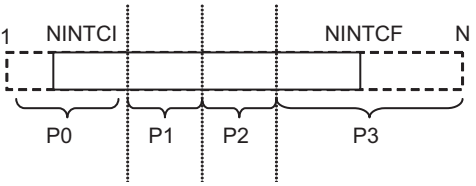

Figure 2.4 GEN_BLOCK distribution for array DIREC1.

are divided evenly among all the threads and data are referenced contiguously. If a thread reads or writes some nonlocal data in a parallel loop, array region analysis enables us to calculate the contiguous nonlocal data for regular accesses and we can fetch all the nonlocal data within one communication before these data are read or written. Fortunately, we do not need to communicate for the first loop in Figure 2. 3 due to completely local accesses. But in the second loop of Figure 2.3, some data accesses are indirect and thus actual data references are unknown at compiling time. Therefore, we cannot generate efficient communications based upon static compiler analysis, and at least one communication per iteration has to be inserted. This would incur very high overheads. Hence, an inspector–executor strategy [24, 39] is employed to avoid them.

The inspector–executor approach is a simplification of the strategy developed within the HPF context, as it can be realized by a fully parallel loop, as shown in Figure 2.3. We detect the values for each indirection array in the allocated iterations of each GA process. One way to handle this is to let a hash table save the indices of nonlocal accesses and generate a list of communications for remote array regions. Each element in the hash table represents a region of a global array, which is the minimum unit of communication. Using a hash table can remove duplicated data communications that will otherwise arise if the indirect array accesses from different iterations refer to the same array element. We need to choose the optimal region size of a global array to be represented by a hash table element. This will depend on the size of the global array, data access patterns, and the number of processes, and needs to be further explored. The smaller the array regions, the more small communications are generated. But if we choose a larger array region, the generated communication may include more unnecessary nonlocal data. Another task of the inspector is to determine which iterations access only local data, so that we may overlap nonlocal data communication with local data computation.

A straightforward optimization of the inspector is to merge neighboring regions into one large region in order to reduce the number of communications. The inspector loop in Figure 2.5 only needs to be performed once during execution of the *gccg* program, since the indirection array remains unmodified throughout the program. Our inspector is lightweight because:

1. The location of global arrays is easy to compute in GA due to the simplicity of GA's data distributions.

```
DO iteration=local_low, local_high
   If (this iteration contains non-local data) then
      Store the indices of non-local array elements into a hash table
      Save current iteration number in a nonlocal list
    Else
       Save current iteration number in a local list
    Endif
Enddo
Merge contiguous communications given in the hash table
```

Figure 2.5 Pseudocode for an inspector.

2. The hash table approach enables us to identify and eliminate redundant communications.
3. All of the computations of the inspector are carried out independently by each process.

These factors imply that the overheads of this approach are much lower than is the case in other contexts and that it may be viable even when data access patterns change frequently, as in adaptive applications. For example, an inspector implemented using MPI is less efficient than our approach as each process has to generate communications for both sending and receiving, which rely on other processes' intervention.

The executor shown in Figure 2.6 performs the computation in a parallel loop following the iteration order generated by the inspector. It prefetches nonlocal data via nonblocking communication, here using the nonblocking *get* operation *ga_nbget*() in GA. Simultaneously, the iterations that do not need any nonlocal data are executed so that they are performed concurrently with the communication.

```
! non-local data gathering
Call ga_nbget(….)
DO iteration1=1, number_of_local_data
   Obtain the iteration number from the local list
   Perform the local computation
Enddo
! wait until the non-local data is gathered
Call ga_nbwait()
Do iteration2=1, number_of_nonlocal_data
    Obtain the iteration number from the non-local list
    Perform computation using non-local data
enddo
```

Figure 2.6 Pseudocode for an executor.