UiO : University of Oslo

Jérémie Lagravière

The PGAS Programming Model and Mesh Based Computation: an HPC Challenge

Thesis submitted for the degree of Philosophiae Doctor

Department of Informatics
Faculty of Mathematics and Natural Sciences

University of Oslo
Simula Research Laboratory

2020

Tous les jours on retourne la scène
Juste fauve au milieu de l'arène
On ne renonce pas, on essaye
De regarder droit dans le soleil
Détroit.

You're gonna battle
You're gonna fight
Win or lose, you're gonna be alright
Regardless of the Scoreboard,
You can do anything that you work for.
Apollos Hester.

Tempora mori, tempora mundis recorda.

Preface

This thesis is submitted in partial fulfillment of the requirements for the degree of *Philosophiae Doctor* at the University of Oslo. The research presented here was conducted under the supervision of professor Xing Cai and professor Hoai Phuong Ha and Doctor Johannes Langguth.

The thesis is a collection of three papers, presented in chronological order. The common theme to them is PGAS Programming and High Performance Computing, in particular through the use of UPC and UPC++. The papers are preceded by an introductory chapter that relates them together and provides background information and motivation for the work. The first paper is a joint work with Johannes Langguth, Mohammed Sourouri, Phuong H. Ha and Xing Cai. The second paper is a joint work with Johannes Langguth, Martina Prugger, Lukas Einkemmer,Phuong Hoai Ha and Xing Cai. The third paper is a joint work with Johannes Langguth and Xing Cai, Martina Prugger and Phuong Hoai Ha.

Acknowledgements

I would like to thank my advisors: Professor Xing Cai for his support, patience and invaluable guidance; Professor Phuong Hoai Ha for our great collaboration throughout my Ph.D. work; and Doctor Johannes Langguth for his patience, dedication and his help in my work.
It has been a pleasure to work at Simula Research Laboratory as it provides an amazing work environment for carrying our research.
This Ph.D. was the occasion for me to meet, work and be bound with researchers internationally. Thus, I wish to extend my gratitude to Martina Prugger, Lukas Einkemmer and Alexander Ostermann for welcoming me at University of Innsbruck, Austria. During this work I have had the occasion to collaborate directly with Doctor Martina Prugger, this collaboration led to a friendship that I will value for ever.
Finally, I would like to express my sincere thankfulness to my friends and family, they have been here, all along, to help, listen and care: none of this work would have been possible without them.

Jérémie Lagravière
Oslo, June 2020

List of Papers

Paper I

Jérémie Lagravière, Johannes Langguth, Mohammed Sourouri, Phuong Hoai Ha, Xing Cai

'On the Performance and Energy Efficiency of the PGAS Programming Model on Multicore Architectures'. In *International Conference on High Performance Computing & Simulation (HPCS)*, 18-22 July 2016. DOI: 10.1109/HPCSim.2016.7568416.

Paper II

Jérémie Lagravière, Johannes Langguth, Martina Prugger, Lukas Einkemmer, Phuong Hoai Ha, Xing Cai

'Performance Optimization and Modeling of Fine-Grained Irregular Communication in UPC'. In *Scientific Programming*, vol. 2019, Article ID 6825728, 20 pages, 2019. DOI: https://doi.org/10.1155/2019/6825728

Paper III

Jérémie Lagravière, Johannes Langguth, Martina Prugger, Phuong Hoai Ha, Xing Cai

'A Newcomer In The PGAS World - UPC++ vs UPC: A Comparative Study'. In preparation.

Contents

Preface iii

List of Papers v

Contents vii

List of Figures ix

List of Tables xi

1 Introduction 1
- 1.1 Problem Statement . . . 2
- 1.2 Hardware Architectures . . . 3
- 1.3 Memory and Communication Bound Problems . . . 7
- 1.4 Sparse matrix-vector multiplication . . . 8
- 1.5 The PGAS Programming Model . . . 10
- 1.6 Summary of Papers . . . 12
- 1.7 Discussion . . . 14
- 1.8 Conclusion . . . 17
- References . . . 18

Papers 24

I On the Performance and Energy Efficiency of the PGAS Programming Model on Multicore Architectures 25

II Performance Optimization and Modeling of Fine-Grained Irregular Communication in UPC 35

III A Newcomer In The PGAS World - UPC++ vs UPC: A Comparative Study 57

List of Figures

1.1 Synthetic view of topics involved in this thesis 1
1.2 Uniform Memory Access/Symetric Multi-processing vs Non-Uniform Memory Access. 4
1.3 Many-core ccNUMA architecture architecture for Intel Xeon Phi Knight's Landing. 5
1.4 Cache Coherent-NUMA hardware architecture used for computation on single-node for Paper I. Graphic representation of a two-sockets machine using Intel Xeon CPU E5-2650 6
1.5 Multi-node hardware topology. 7
1.6 Sparse Matrix-Vector Multiplication (SpMV): multiply a sparse matrix A and a dense vector x, and return the result as a dense vector y. 8
1.7 Example of a few popular sparse matrix storage formats. In HYB format, columns for split is 2. 9
1.8 PGAS memory model 11
1.9 The mesh used in Paper II and in an unpublished version of Paper III, models a healthy male human heart acquired by MRI. Image courtesy of Johannes Langguth. 13

List of Tables

1.1 Amount of publications for PGAS languages from 2010 to 2014, available on Google Scholar. Data collected in October 2014. . . 2

Chapter 1

Introduction

In the field of High Performance Computing, Mesh based computations constitute a typical solution to implement simulations. These simulations are used to process problems related to different fields of science and applied-sciences. Mesh based computation is used in a variety of applications such as: heat transfer and fluid mechanics [56], ocean modeling [14], astrophysics [49], and cardiac modeling [18, 45].

There is a direct causality between the cost of doing simulations and the demand for always more computational power. Indeed, in the aforementioned fields of application it is *de-facto* an economical efficient solution to focus on simulations. Additionally, an increased processing power means better results (for instance, higher definition) obtained faster (lower execution time).

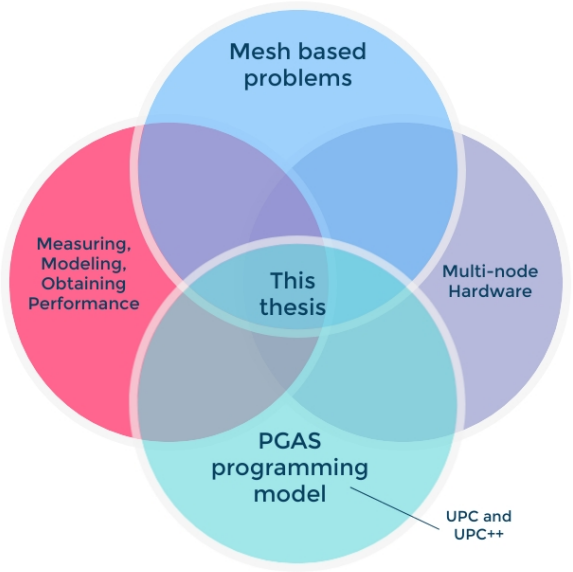


Figure 1.1: Synthetic view of topics involved in this thesis

The focus of this thesis is described synthetically in Figure 1.1 (Page

1). This thesis is 'located' at the crossroads of four main topics: the PGAS programming model, Multi-node Hardware, Performance measurement, prediction and optimization and Mesh Based Computations. In the following sections of this chapter we will describe relevant aspects of these topics as part of a literature review, then in the "Discussion" (Section 1.7, page 14) part of this chapter we will connect this background knowledge and the papers presented in this thesis in order to explain what is the overarching theme for this PhD project.

1.1 Problem Statement

High Performance Computing (HPC) is a key field of research and engineering for scientific computation, data science and artificial intelligence. From a hardware point of view, HPC evolves: supercomputers that are more and more heterogeneous, NUMA architecture that is now extremely common, manycore hardware that is slowly appearing as a satisfying solution to replace limitations in the frequency of processing elements, by increasing the number of processing elements.

In this PhD project we studied the PGAS programming model and used it for High Performance Computing applications. This choice, was made for several reasons: for more and more complex hardware described in the previous paragraph, there are multiple solutions to program for it: simplified frameworks yielding poor performance but offering good programmability, good usability; at the "opposite" of this spectrum there are programming techniques typically involving Message Passing Interface (MPI)+OpenMP [22, 31], Thread Building Block [59], etc. These technologies offer precise control on data flow and control flow, yielding very high performance. However to achieve this they require high effort in programming. PGAS languages offer a different way to program for HPC: by looking at the memory space as a global partitioned space the programming effort can be considered lower than with MPI+OpenMP.

PGAS languages	Amount of publications between 2010 and 2014
UPC	1300
X10	1000
Fortran	800
Chapel	660
Titanium	600
GASNet	360
OpenSHMEM	120

Table 1.1: Amount of publications for PGAS languages from 2010 to 2014, available on Google Scholar. Data collected in October 2014.

In this PhD project, we studied the PGAS programming model by using

two of its implementations: UPC and UPC++. This is already a point that can be pondered or discussed as there are multiple PGAS languages available such as: X10, Chapel, OpenSHMEM, Co-array Fortran, etc [54]. When we started this PhD project in 2014 there were already existing studies about UPC showing promising results in the HPC field [21, 28, 29, 51, 71]. On a more quantitative approach, as shown in Table 1.1 (page 2), there were large amounts of publications available and recently published between 2010 and 2014 (starting date of the PhD project).

After having used UPC for Paper I and II, using UPC++ seemed to be a logical continuation of our research, as this newly available language (Version 1.0 release in September 2019 [65]), can be considered as a "descendant" of UPC, considering that UPC++ inherits some of UPC's features and even the name of the language inherits from "UPC".

To carry out this study, we used several hardware architectures, single-node hardware, multi-node hardware, many-core hardware. The variety of the hardware we used was to test, verify and prove whether the PGAS technologies we had chosen were able to work and achieve satisfying performance on "all-terrains" (i.e. various hardware architectures). We provide a view on obtainable performance with UPC and UPC++, and we provide a performance prediction model for UPC. In addition, to predict performance and assess obtainable performance we implemented a SpMV kernel in relation with the solving of a Partial Differential Equation modeling the behavior of a healthy human heart.

Our problem statement for this PhD thesis, is then: **Is it possible for PGAS languages (in particular UPC and UPC++) to yield high performance and scaling on single-node, multi-node and many-core hardware architectures when used to implement "real world" applications?**

1.2 Hardware Architectures

In the experiments described in this thesis, we have used both single-node and multi-node hardware designs. In this section we present key notions, that are directly related to the hardware architectures used in Papers I, II and III, such as: Non-Uniform Memory Access (NUMA), multi-node hardware and many-core hardware.

1.2.1 NUMA

Non-uniform memory access (NUMA) is a computer memory design used in multiprocessing, where the memory access time depends on the memory location relative to the processor. Under NUMA, a processor can access its own local memory faster than non-local memory (memory local to another processor or memory shared between processors). The benefits of NUMA are limited to particular workloads, notably on servers where the data is often associated strongly with certain tasks or users [9].

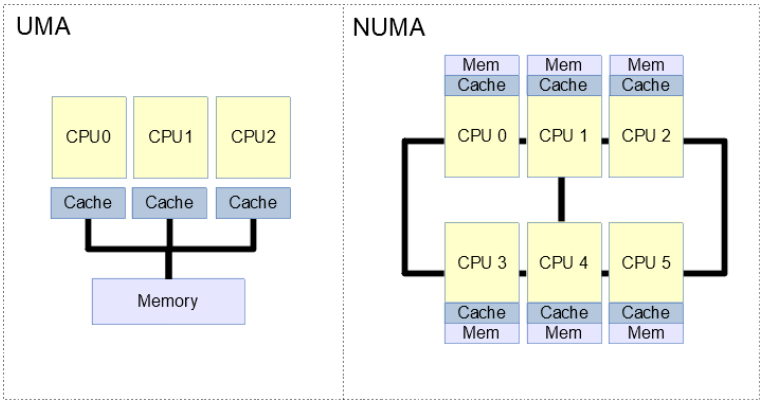


Figure 1.2: Uniform Memory Access/Symetric Multi-processing vs Non-Uniform Memory Access. Source [39].

Modern CPUs operate considerably faster than the main memory they use. In the early days of computing and data processing, the CPU generally ran slower than its own memory. The performance lines of processors and memory crossed in the 1960s with the advent of the first supercomputers. Since then, CPUs increasingly have found themselves "starved for data" and having to stall while waiting for data to arrive from memory. Many supercomputer designs of the 1980s and 1990s focused on providing high-speed memory access as opposed to faster processors, allowing the computers to work on large data sets at speeds other systems could not approach [9].

Figure 1.2 (page 4) presents a comparative view between Uniform Memory Access (UMA) and Non-Uniform Memory Access. In UMA case, any processor (core) can access any memory location with the same access time / latency. Symetric Multi-processing (SMP) or UMA is effective and easy to program but scaling is limited to few processors. It is really complex to ensure "uniform access" when hundreds of CPUs compete to access data in memory[1], thus leading to decrease in performance.

SMP systems cannot deliver sufficient memory throughput for large numbers of processors without causing excessive penalties due to memory latency. ccNUMA hardware architectures offer scalable memory bandwidth while keeping memory latencies reasonable and often delivering latencies in the same class as much smaller systems [33].

NUMA attempts to solve this problem (SMP/UMA) by providing separate memory for each processor. Doing so, NUMA reduces the performance cost when many processing elements attempt to access the same memory. In addition, NUMA/ccNUMA systems have a physically distributed memory that is logically shared. The aggregated memory appears as one single address space. Memory access performance depends on which CPU (core) accesses which parts of memory

[1] indico.cern.ch/event/605204/contributions/2440614/attachments/1471788/2277728/02_Perugia_HPC.pdf

("local" vs. "remote" access). Although there is a single address space (shared memory), there are private caches, or partially shared caches, for the different CPU cores (also true on UMA systems). Cache coherence protocols guarantee consistency between cached data and data in the shared memory at all times [32]. This is illustrated by Figure 1.4 (page 6) with a two-sockets architecture using two Intel Xeon CPU E5-2650, and in a more recent hardware architecture using an Intel Xeon Phi CPU 7250 [36] in Figure 1.3 (page 5).

In all our Papers presented in this thesis we have used NUMA architecture, and we have shown that UPC and UPC++ can both run and deliver performance on this kind of hardware architecture.

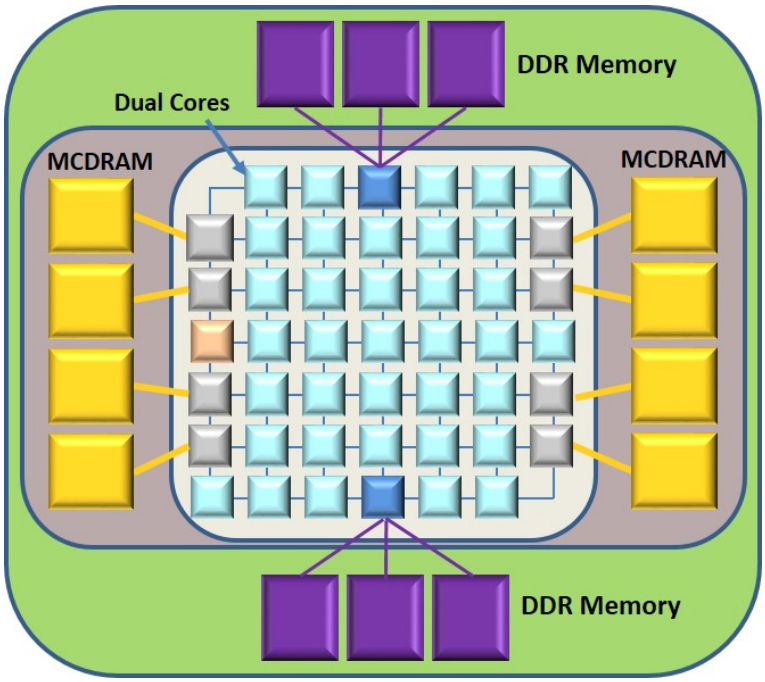


Figure 1.3: Many-core ccNUMA architecture architecture for Intel Xeon Phi Knight's Landing. Source: mrfunk.info/?page_id=181 [16]

New hardware architectures have been introduced recently, such as Intel Xeon Phi [36, 38], AMD Epyc [3, 48] and, in 2019, its Intel proposed XEON® PLATINUM 9200 series. These architectures provide a hardware design including a large number of cores, many NUMA domains, and a cache-coherent system. We have shown that both UPC and UPC++ are able to run and deliver performance on many-core NUMA architecture in Papers II and III.

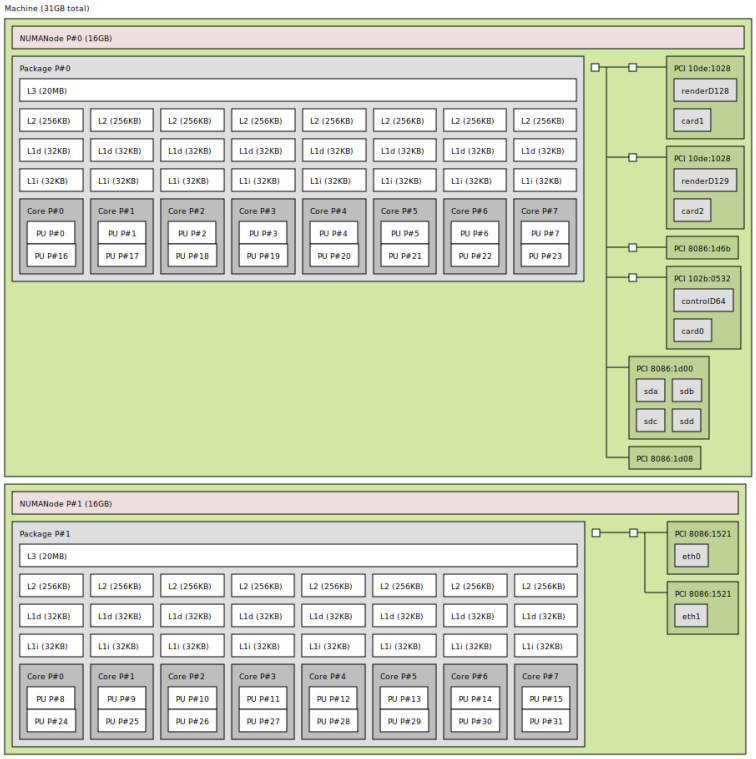


Figure 1.4: Cache Coherent-NUMA hardware architecture used for computation on single-node for Paper I. Graphic representation of a two-sockets machine using Intel Xeon CPU E5-2650 [16]

1.2.2 Multi-node hardware

In Papers I, II and III, we have used multi-node hardware, in this section we present a definition of this type of hardware architecture. Thus, in this section we describe the architecture of the multi-node hardware we used [2].

A supercomputer is a computer with a high level of performance as compared to a regular PC or desktop computer. The performance of a supercomputer is commonly measured in floating-point operations per second (FLOPS) instead of million instructions per second (MIPS).

Supercomputers play an important role in the field of computational science, and are used for a wide range of computationally intensive tasks in various fields, including quantum mechanics, weather forecasting, climate research, oil and gas exploration, molecular modeling, physical simulations and cardiac modeling [58].

As a rough simplification, the modern supercomputer is divided up into nodes. Each node may have multiple processors which are the hearts of the node, as shown in Figure 1.5 (page 7). Each processor has access to memory

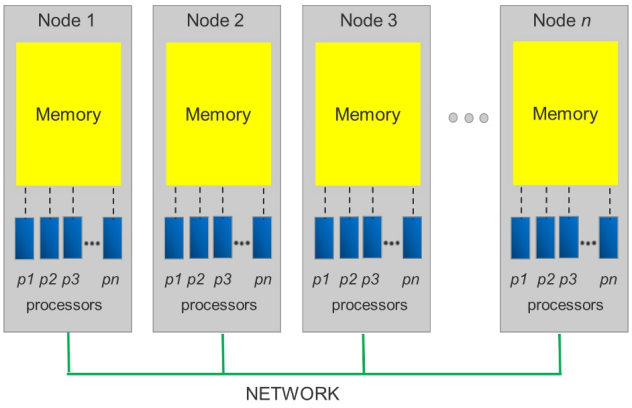


Figure 1.5: Multi-node hardware topology. Source [46].

on its node, but does not have access to the memory on the other nodes. Communication of information can occur between processors within or between nodes. Communication between nodes is achieved through a high-speed network, but is usually considerably slower than communication within a node. In order to benefit from a supercomputer, the programs need to utilise the resources of multiple nodes in parallel. The network connects the nodes to make larger parallel computer clusters [46].

In that sense, supercomputers can be seen as a special case of computer clusters, a logical representation of a multi-node supercomputer can be seen in Figure 1.5 (page 7). A computer cluster is a set of loosely or tightly connected computers that work together so that, in many respects, they can be viewed as a single system. The components of a cluster are usually connected to each other through fast local area networks, with each node (computer used as a server) running its own instance of an operating system. Clusters are usually deployed to improve performance and availability over that of a single computer, while typically being much more cost-effective than single computers of comparable speed or availability [68].

1.3 Memory and Communication Bound Problems

In scientific computing and High Performance Computing (HPC), the "memory wall" [70] has become a major factor affecting the performance of many scientific applications. This is mainly due to the ever increasing sizes of dataset, which lead to large memory footprints. Additionally, the memory wall is exacerbated by architectural trends whereby the system memory is physically fragmented while at the same time local memory hierarchies become deeper. Thus, a memory-intensive application may spend most of its execution time moving data to or from (read or write) the memory hierarchy and its time-to solution may primarily

depend on how efficiently it uses the memory subsystem of the chosen hardware architecture [64].

This kind of applications are called "memory-bound" and this type of applications is the main focus of the papers presented in this PhD thesis. The execution time of memory-bound application is dependent on the speed of the memory, meaning that no matter how high the computing power of the processor the machine is using, the calculation is only limited by the speed of the memory, and this creates a bottleneck in the execution time due to the slower speed of the memory.

1.4 Sparse matrix-vector multiplication

Sparse matrix-vector multiplication (SpMV) of the form $y = Ax$ is a widely used computational kernel existing in many scientific applications. The input matrix A is sparse. The input vector x and the output vector y are dense.

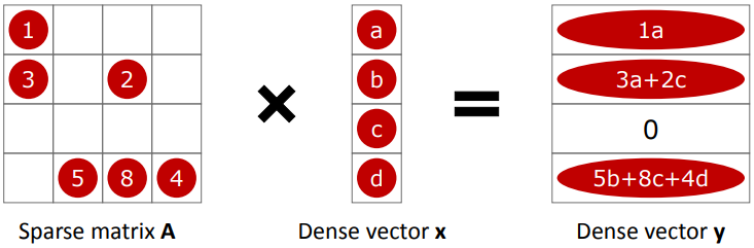


Figure 1.6: Sparse Matrix-Vector Multiplication (SpMV): multiply a sparse matrix A and a dense vector x, and return the result as a dense vector y. Source: [42].

Sparse matrices are involved in linear systems, eigen-systems and partial differential equations (PDE) from a wide range of scientific and engineering domains. Thus, SpMV is considered as a key operation in engineering and scientific computing. For these applications the optimization of the SpMV is very relevant [34].

In the following section we describe storage formats for SpMV. Storage formats have a direct and important impact on the computations, as they influence the programming style used to compute the matrix-vector multiplication involved in SpMV. We chose to describe in this introduction detailed information about some of the available storage formats for SpMV. However, this is not the central focus of our research, storage formats for matrices and SpMV can be seen as a field of research, in our case (Papers II and III), the data storage format was considered as "given".

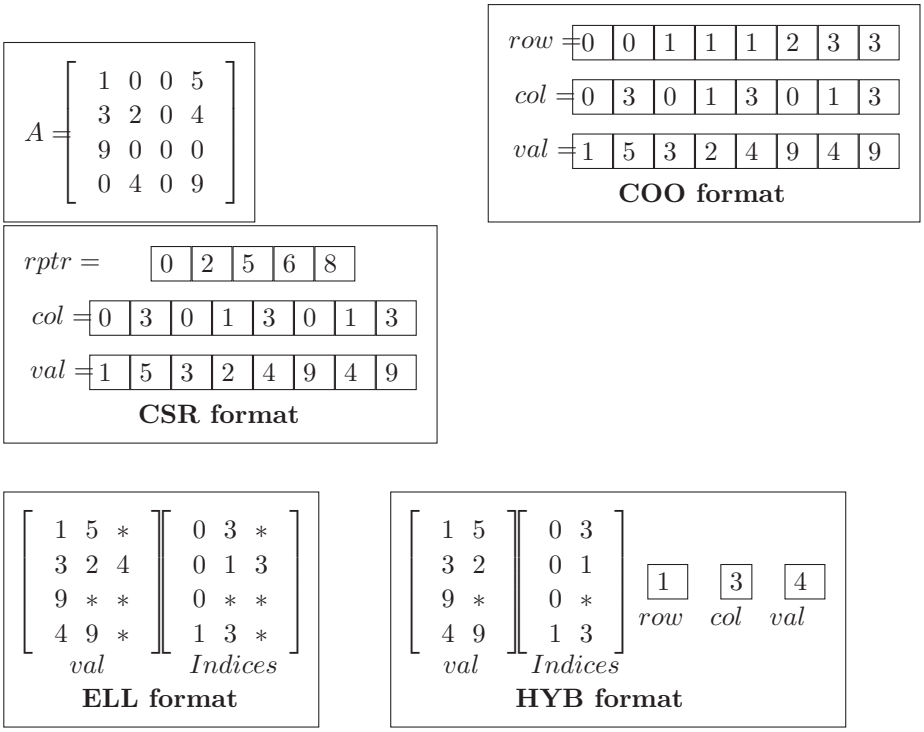


Figure 1.7: Example of a few popular sparse matrix storage formats. In HYB format, columns for split is 2. [10, 11]

Storage Formats for SpMV

Multiple sparse matrix storage formats have been proposed over the years to reduce space in memory by not storing zero elements. The majority of these formats is *general*, meaning they require $O(n)$ space for matrices with n nonzeros. Other formats are only adapted to specific sparsity patterns and can use $O(n^2)$ space in the worst case. For instance, the diagonal format DIA requires that all except for a few (constant in asymptotic terms) matrix diagonals are zero for the format in order to be efficient. In the following section, we briefly review the most popular storage formats. Please refer to existing literature for more in-depth discussion on other storage formats [27, 60].

Figure 1.7 shows an example of a sparse matrix A, and popular sparse representations of A are presented alongside. We denote the number of rows, columns, and nonzero entries of the sparse matrix A with R, C, and NNZ respectively. The *coordinate* (COO) format stores a map from all row and column indices to the value of the corresponding nonzero entries. This is denoted by the dense arrays, row, col, and val, in Figure 1.7. The *compressed sparse row* (CSR) format, which is arguably the most popular format, compresses the

row array to store the start positions of all rows in the corresponding col and val arrays. The *compressed sparse column* (CSC) format is similar to CSR but compresses columns instead.

The *diagonal* (DIA) format stores matrix diagonals as dense vectors, along with an offset that records the location of the diagonal. While some sparse matrices in scientific computing naturally have such a shape, the format can incur a quadratic overhead when storing general sparse matrices. The *ELLPACK* (ELL)format stores the sparse matrix A as a dense rectangular matrix by shifting the nonzeros in each row to the left and zero-padding all rows that have fewer than the maximum number of entries. ELLPACK uses an index matrix to store the corresponding column index for every nonzero element (as shown in Figure 1.7). Furthermore, since the number of operations per row is known beforehand, the computation for each row can be unrolled completely and optimized for SIMD processing and better cache usage. The storage size of ELLPACK thus depends on the maximum number of nonzeros values in a row of A, which can be very large (close to n) for matrices with a large variation in the number of nonzeros per row. The *hybrid* (HYB) format mitigates this problem by using ELLPACK for storing most of the matrix A and COO for additional entries in rows with a large number of nonzeros. This usually reduces the required amount of padding greatly, while maintaining some of the advantages of ELLPACK. Several high-performance libraries such as Intel MKL [37], CUSP [52], and cuSPARSE [53] from NVIDIA implement some or all of these formats. In Papers II and III, where SpMV is involved, we have used the ELLPACK for storing the sparse matrix. In these papers, the ELLPACK format was considered as "given". The focus of our work is not to reconsider the data format for SpMV.

1.5 The PGAS Programming Model

In Papers I, II and III we have used instances of the PGAS programming model, namely Unified Parallel C (UPC) and Unified Parallel C++ (UPC++). In the Discussion section (page 14) of this introduction we address the reasons why we chose PGAS (UPC and UPC++) as a subject for our studies. The goal of the following section is to present general aspects of the Partitioned Global Address Space (PGAS) programming model.

One of the challenges met by the programmers for high performance parallel systems is the somewhat difficult programming environment and constraints. For large-scale parallel machines, the most common model is message passing, popularized in the Message Passing Interface (MPI) [31] standard interface. Relative to competing programming models such as OpenMP [22], threads, data parallelism or automatic parallelization, message passing has the advantage in scalability, since it runs on distributed memory multiprocessors and with performance "tunability", since the programmer has full control over data layout, communication, and load balance [20].

A promising approach to solve these difficulties (such as data distribution, communication, etc.) is the concept of a partitioned global address space (PGAS,

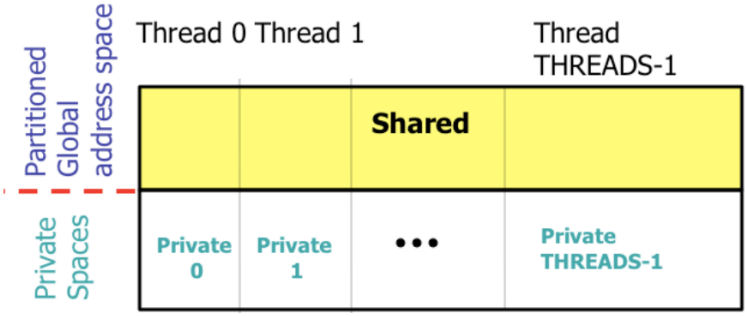


Figure 1.8: PGAS memory model [12, 74]

see e.g., [9, 11, 27]) programming model. As presented in Figure 1.8 (page 11), in a PGAS language the underlying memory concept is that each thread owns a certain part of the systems memory. A part of that memory is shared and some of it is private. Each thread can read and/or write shared memory even if the memory location is owned by another thread. Shared memory location can be accessed whether it is physically located on the same node or not. PGAS languages have been developed for Java (Titanium, X10), Fortran (Co-array Fortran, which is part of the Fortran standard), C (Unified Parallel C or UPC) and C++ (UPC++) [4, 5, 12, 13, 47, 57, 73].

In PGAS languages, in UPC in particular (i.e. not UPC++), the programming language provides solutions for the programmer to implicitly communicate data between threads, whether these threads are located on different, cores, CPU, nodes, NUMA domains, etc. Additionally, languages such as UPC or UPC++ enforces one-sided communication where the sender thread does have to wait for the data transfer to be completed in order to continue processing further instructions. UPC++, one of the latest implementation of the PGAS programming model [65], puts specific emphasis on asynchronous communication in conjunction with the use of promises and futures [13, 50, 73]. The use of promises, futures, asynchronous communication is encouraged by UPC++ guidelines and programming model [4, 5, 73] in order to achieve Asynchronous-PGAS.

PGAS languages provide an alternative to MPI+OpenMP, by offering to programmers a set of solutions to achieve parallel computing, on multiple kinds of hardware platforms (single, multi-node or many-core). These solutions are theoretically providing an ease-of programming designed to lower programming efforts to achieve both parallelization and high performance. In this PhD thesis, we work in particular with UPC and UPC++ languages, in order to assess the achievable performance in an HPC context.

1.6 Summary of Papers

During the course of this PhD project, one paper was published in an international reviewed conference [44] (page 25), one paper was published in international peer-reviewed journal [45] (page 35), and another one is destined to be submitted to an international reviewed conference (see page 57).

The following presents a summary of each paper.

1.6.1 Paper I: On the Performance and Energy Efficiency of the PGAS Programming Model on Multi-core Architectures

Using large-scale multicore systems to get the maximum performance and energy efficiency with manageable programmability is a major challenge. The partitioned global address space (PGAS) programming model enhances programmability by providing a global address space over large-scale computing systems. The complexity of parallel programming is one of the fundamental challenges that the HPC research community faces. In the last decade, the Partitioned Global Address Space model has been presented as one possible solution for this problem. It provides improved programmability while maintaining high performance, which is the primary goal in HPC.

Paper I investigates the performance of UPC compared to OpenMP and MPI. OpenMP and MPI are considered the de-facto standard for both single and multi-node computation. In order to assess the performance of UPC we selected a set of kernels from the NAS Parallel Benchmark (NPB) [17]: Conjugate Gradient, Multi-Grid, Fourier Transform and Integer Sort. These kernels are destined to put stress on the hardware architecture on multiple parameters: computing power and communication between cores, sockets, nodes.

By comparing the obtained performance of UPC and MPI/OpenMP, using both single-node and multi-node hardware architecture, we verified the feasibility of using UPC in modern hardware running compute-bound, memory and communication-bound applications. In this study we show that UPC can perform closely to its competitors. However, we noticed differences in performance that deserved analysis in order to understand "where" the performance variations come from. To achieve this analysis we have chosen a software suiting our single-node hardware equipped with Intel CPU: Intel Performance Counter Monitor or Intel PCM [35].

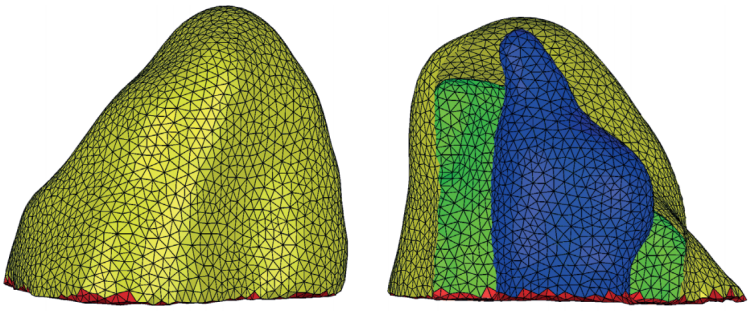
From the hardware perspective we used NUMA [9] architecture by using single-node equipped with two processors located on two sockets, as illustrated by Figure 1.4 (page 6) We have also used multi-node hardware architecture with Abel Supercomputer where the nodes are interconnected with high speed network based on Infiniband/FDR (56 Gbits/s eq 6.78 Gbytes/s) [2].

In paper I, we provide results that show that UPC can compete with its competitors, opening the possibility of broadening our research on PGAS programming model using UPC.

1.6.2 Paper II: Performance optimization and modeling of fine-grained irregular communication in UPC

The goal of Paper II is to focus on performance optimization and performance prediction for UPC language. To do so, we propose new implementations of kernels and we assess the performance of these kernels on both single node and multi-node hardware. Plus, we provide a performance prediction model, that can tell its users what performance can be expected from their UPC programs on a chosen hardware platform.

In Paper II, we continue our efforts in assessing UPC's performance. In this study we decided to focus on implementations of two kernels in UPC: a SpMV and the Heat Equation in 2 dimensions. The SpMV kernel is performed on a 3D mesh presented in Figure 1.9 (page 13).



(a) Side view of the mesh (b) Inside view of the mesh

Figure 1.9: The mesh used in Paper II and in an unpublished version of Paper III, models a healthy male human heart acquired by MRI. Image courtesy of Johannes Langguth.

Paper II, also focuses on the Heat Equation Kernel [67]. For this experiment, we solve the 2D heat diffusion equation: $\frac{\partial \phi}{\partial t} = \frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2}$ on a uniform mesh. We employ the finite difference discretization. SpMV and Heat Equation kernel are implemented in UPC in this Paper II, and we used single and multi-node hardware architecture to assess the performance of UPC as well as the quality of our implementations.

In addition, in Paper II, we provide a performance prediction model for UPC applications. We assess the quality of the predictions perform with our performance prediction model by comparing predicted performance and obtained performance on multi-node hardware for both SpMV and Heat Equation kernels.

Thus, in paper II, we provide results that shows that UPC can achieve strong scaling on multi-node hardware and that this performance can be predicted thanks to a synthetic performance prediction model.

1.6.3 Paper III: A Newcomer In The PGAS World - UPC++ vs UPC: A Comparative Study

In paper III, we focus on UPC++, the latest PGAS model implementation at the time. UPC++ is a programming languages extending C++ [4, 5, 73] which version 1.0 was released in September 2019 [65].

UPC++ has been the object of great efforts to provide both new features, performance and stable implementation of a new PGAS language [65]. Thus, we decided to focus Paper III on studying the obtainable performance and the programming style of UPC++ comparing both of these aspects to UPC.

In Paper III, we provide new implementations of two kernels: a sparse-matrix vector multiplication on a 3D Mesh, presented in Figure 1.9 (page 13) and Heat Equation on a 2-dimension domain. These implementations solve the same kernels as of which are used in Paper II. In Paper III, we present in detail how these kernels are implemented, by showing excerpt of SpMV and Heat Equation codes both in UPC and UPC++. These code listings are presented in PaperIII in order to show how UPC and UPC++ differ and how we implemented our kernels.

The main focus of Paper III is to compare UPC++ and UPC. Thus, we measured obtained performance of newly implemented UPC++ kernels and compared these performance to that of UPC. To do so, we have used single-node and multi-node hardware, as well as many-core hardware platform. This shows the "all-terrain" capabilities of both UPC and UPC++, one of the key features of PGAS languages is "code it once, run it on any compatible hardware platforms". Thus, checking that UPC and UPC++ work and deliver performance on multi-node and many-core platform is an important aspect of Paper III.

1.7 Discussion

In the following, we review the main contribution of each paper presented in this PhD thesis, and for each paper we describe the limitations of our work.

As one of the focuses of this thesis is about obtaining high performance, in paper I we assess UPC's performance by using well-known benchmarks and, in Papers II and III, by also providing specific implementations of "real world" application focusing on memory-bound problems. By "real world" application we specifically mean applications that compute kernels related to natural sciences, in our case, we simulated a diffusion case on a healthy human heart and we also provided an implementation of the more well-known heat equation, which models the diffusion of heat in a given domain. Once we obtained a satisfying level of performance using UPC we decided to provide a performance model, in order to offer the possibility of predicting obtainable performance on a given hardware architecture for a given application. We assessed the accuracy of our performance model by predicting the performance of the application we created, proving that performance is not only obtainable but also repeatable and predictable, proving that PGAS can be a valuable alternative to the **de facto** standard MPI and OpenMP, both in industrial and research fields. In the latest

study presented in this thesis we focus on a newcomer in the PGAS family of programming languages: UPC++. We implemented identical kernels as we did in UPC, and compared the obtained performance in UPC++ to the obtained performance in UPC. We wanted to check and verify whether UPC++ in its version 1.0 is able to yield high performance using "real world" applications and recent hardware architecture. In this section, we provide a description of our contributions in this thesis, and our view on our papers relate to each other, forming a consistent ensemble.

Our first study, presented in Paper I, is about assessing the obtainable performance of UPC using well-known benchmarks using single-node and multi-node hardware. This study was our way to build a solid base to build upon for further studies using PGAS and UPC in particular. This first study can be seen as taking part into a "tradition" of benchmarking UPC's performance using the NAS Parallel Benchmark [6, 7, 17, 40], as studies had been released previously on a similar topic: [19, 30, 41, 62, 72]. Our contribution in this suite of studies is that we provided new performance results, on recent single-node and multi-node hardware, providing additionally energy measurements and power consumption and finalizing our study with a detailed investigation of the behavior of UPC and its competitors in cache and memory on single-node hardware. Additionally, UPC was and still is updated nowadays (latest version was released in September 2019 [12]), thus, providing new performance results using the latest version of UPC was also part of the contribution of this first paper. However, in this study we focused both on getting performance and scalability but also on comparing these performance results to the one we obtained using strictly identical benchmarks with MPI and OpenMP. In other words, observing a satisfying scaling of UPC was not good enough, the scaling and performance in general had to be at least comparable or close to the one obtained with the **de facto** standard technologies that are MPI and OpenMP.

However, paper I was relying on code that is both dated (NPB) and purely on benchmarks that have no real application. Additionally, to keep carrying out research about PGAS and UPC, we needed to provide a solid contribution to the field of research. Also, retrospectively, the choice of NPB seems valid when compared to the available studies using this benchmark suite, however, other benchmarks would have been interesting and as valid as NPB or more, such as: High Performance Linpack (HPL) benchmark [6, 7, 23, 40], which is used for ranking top supercomputers on TOP-500 [24] or Graph-500 [69]. As we focused on energy consumption in Paper I, focusing on LINPACK or Graph-500 would have been a better option as these benchmarks are used in the Green 500, as it ranks computers from the TOP500 list of supercomputers in terms of energy efficiency, typically measured as LINPACK FLOPS per watt [8, 43].

The conclusion of this first study encouraged us to pursue further our work, in paper II, consisting in studying PGAS languages, UPC in particular as it is an active and long time running topic in High Performance Computing (HPC) [28, 29, 51, 71]. We decided to study further UPC's obtainable performance by implementing what we called "real world applications", in our case the main focus was the implementation of a spare-matrix vector multiplication (SpMV)

which is part of the solver of a partial differential equation (PDE) modeling the diffusion of an electric current through a healthy human heart. We chose a SpMV kernel because it is a well-known, often used kernel in implementing and computing mesh-based PDE applications [1]. In addition, SpMV in the HPC field is the center of attention in terms of performance optimizations and evaluations [1, 55, 61, 66]. We thus decided to not only provide insights for UPC's obtainable performance computing SpMV for mesh-based PDE applications but also to provide a solution for predicting performance.

In Paper II, we assess the accuracy of the predicted performance by comparing them to the obtained performance on multi-node hardware. Providing solutions to predict performance in HPC is an active field of study, and it also shows the repeatability, predictability and stability of both our experiments and the technology used for these experiments. The performance prediction model we describe in Paper II is based on parsing the data to compute to enumerate the amount of computation and communication (intra and inter-node), these amounts are then pondered by measured throughput (intra and inter-node) on the chosen hardware architecture. Thanks to this model, we managed to estimate the obtainable performance with a satisfying accuracy. However, we had to do a trade-off between the accuracy of the model and its complexity (i.e. complexity in this context means "amount of information required to make a prediction"). By requiring too much information to the user the risk of getting close to the Bonini's paradox [15, 25] is high. Bonini's paradox is defined as follows: As a model of a complex system becomes more complete, it becomes less understandable. Alternatively, as a model grows more realistic, it also becomes just as difficult to understand as the real-world processing it represents [26].

The last part of this PhD thesis presents Paper III, in this study we focused our efforts on using a newcomer in the PGAS family of programming languages: UPC++ [4, 5, 73]. In Paper III, we use UPC++ as a continuation of our previous work, as UPC++ inherits at least the base naming of "UPC" and UPC++'s version 1.0 was released in September 2019 [65] in addition to being part of the PGAS family of programming languages. In Paper III, we provide new implementations of two kernels Heat Equation 2D and SpMV, which we then executed on a supercomputer and then compared the obtained results with our previous UPC kernels. Paper III, provides also technical insights on how to program with UPC++ and key differences with its ancestor UPC. As a result, with UPC++ we obtained comparable performance to that of UPC, if not better in certain cases. Additionally, in Paper III, we used a manycore hardware architecture based on Intel Xeon Phi CPU 7250 [36, 38]. In this context both UPC and UPC++ achieved strong scaling with comparable performance.

Paper III, however, does not present all of the newest features offered by UPC++, such as asynchronous Remote Procedure Call (RPC) or communication overlapping with computation thanks to asynchronous communication. This is mainly due to the time constraint of the PhD project. In addition, we wanted to compare UPC++ a in form that is closest to the UPC counterpart. This is clearly a very important key aspect of future work: implementing, for instance, our SpMV kernel with communication and computation overlapping and hopefully getting

superior performance compared to that of UPC would demonstrate the strong interest in adopting UPC++ for future PGAS implementations. Additionally, providing new performance results using newer hardware architecture and equipment, would be a valid way of assessing both UPC and UPC++ performance and reliability.

1.8 Conclusion

In the Introduction chapter of this PhD thesis, we have presented key aspects used in the papers presented in the present document. The goal of our work is to show whether it is possible to obtain high performance and scaling on single-node, multi-node and many-core hardware architectures using PGAS languages (UPC and UPC++) when used to implement "real world" applications.

In Paper I, we assess the obtainable performance of UPC using NPB benchmark [17] on multi-node hardware architecture.

We provide in Paper II in addition to a newly implemented SpMV kernel a performance prediction model, that can help its user to both evaluate how a UPC application can perform on a pre-selected hardware platform and define the best data distribution strategy to get the highest performance possible.

In Paper III, we provide newly implemented kernels using UPC++. UPC++ is a newly available implementation of the PGAS paradigm and a "descendant" of UPC. In this study we show that it is possible to obtain satisfying performance and scaling on both multi-node and many-core hardware architectures.

The main conclusion of our work corresponds to an answer to our problem statement:

- UPC can deliver comparable performance to MPI and OpenMP when used on single-node or multi-node when running well-known benchmarks such as NPB benchmarks suite
- UPC delivers stable/repeatable performance, and it can be predicted thanks to our performance prediction model
- UPC and UPC++ can deliver satisfying performance on single-node, multi-node and many-core hardware, when used for "real world applications" such as SpMV or Heat Equation
- The ease of programming allegedly provided by UPC and UPC++ is not always applicable, and additional efforts are required from the programmer when aiming at high performance when computing algorithm such as SpMV or Heat Equation

There are multiple ways in which our work can be extended: UPC, UPC++ can be used in conjunction with CUDA, used for GPGPU computing. Additionally, we have used UPC++ in a UPC-like fashion, using specific features offered by UPC++ would definitely be a valid option to consider. Also, newly arrived AI chips arrive [63], the future of PGAS could be in implementing software for this new kind of hardware architecture.

References

[1] Abdelfattah, A., Ltaief, H., and Keyes, D. “High performance multi-GPU SpMV for multi-component PDE-based applications”. In: *European Conference on Parallel Processing*. Springer. 2015, pp. 601–612.

[2] *The Abel computer cluster*. www.uio.no/english/services/it/research/hpc/abel/. [Online; accessed 17-December-2019].

[3] AMD. *CPU - Epyc - 7702P*. www.amd.com/en/products/cpu/amd-epyc-7702p. [Online; accessed 19-December-2019]. 2017.

[4] Bachan, J. et al. “The UPC++ PGAS library for exascale computing”. In: *Proceedings of the Second Annual PGAS Applications Workshop*. ACM. 2017, p. 7.

[5] Bachan, J. et al. “UPC++: A High-Performance Communication Framework for Asynchronous Computation”. In: *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2019, pp. 963–973.

[6] Bailey, D. H. et al. “The NAS parallel benchmarks summary and preliminary results”. In: *Supercomputing’91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*. IEEE. 1991, pp. 158–165.

[7] Bailey, D. et al. *The NAS parallel benchmarks 2.0*. Tech. rep. Technical Report NAS-95-020, NASA Ames Research Center, 1995.

[8] Barroso, L. A. “The price of performance”. In: *Queue* vol. 3, no. 7 (2005), pp. 48–53.

[9] Baxter, W. F. et al. *Symmetric multiprocessing computer with non-uniform memory access architecture*. US Patent 5,887,146. 1999.

[10] Bell, N. and Garland, M. *Efficient sparse matrix-vector multiplication on CUDA*. Tech. rep. Nvidia Technical Report NVR-2008-004, Nvidia Corporation, 2008.

[11] Bell, N. and Garland, M. “Implementing sparse matrix-vector multiplication on throughput-oriented processors”. In: *Proceedings of the conference on high performance computing networking, storage and analysis*. ACM. 2009, p. 18.

[12] Berkeley. *UPC Implementation From Berkeley*. upc.lbl.gov. last accessed on 18/12/2019.

[13] Berkeley Lab. *UPC++ publications*. bitbucket.org/berkeleylab/upcxx/wiki/Publications. last accessed on 18/12/2019. 2019.

[14] Blayo, E. and Debreu, L. “Adaptive mesh refinement for finite-difference ocean models: first experiments”. In: *Journal of Physical Oceanography* vol. 29, no. 6 (1999), pp. 1239–1250.

[15] Bonini, C. *Simulation of information and decision systems in the firm*. Ford Foundation doctoral dissertation series. Prentice-Hall, 1963.

[16] Broquedis, F. et al. “hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications”. In: *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*. Ed. by IEEE. Pisa, Italy, Feb. 2010.

[17] Browning, D. “NAS Kernels Survey Report”. In: Report RND-92-003, February 1992, NAS Systems Division, NASA Ames Research ... 1985.

[18] Bruaset, A. M. and Tveito, A. *Numerical Solution of Partial Differential Equations on Parallel Computers*. Vol. LNCSE 51. Springer, 2006.

[19] Cantonnet, F. et al. “Productivity analysis of the UPC language”. In: *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.* IEEE. 2004, p. 254.

[20] Chen, W.-Y. et al. “A performance analysis of the Berkeley UPC compiler”. In: *Proceedings of the 17th annual international conference on Supercomputing*. ACM. 2003, pp. 63–73.

[21] Christadler, I., Erbacci, G., and Simpson, A. D. “Performance and productivity of new programming languages”. In: *Facing the Multicore-Challenge II*. Springer, 2012, pp. 24–35.

[22] Dagum, L. and Menon, R. “OpenMP: An industry-standard API for shared-memory programming”. In: *Computing in Science & Engineering*, no. 1 (1998), pp. 46–55.

[23] Dongarra, J. J., Luszczek, P., and Petitet, A. “The LINPACK benchmark: past, present and future”. In: *Concurrency and Computation: practice and experience* vol. 15, no. 9 (2003), pp. 803–820.

[24] Dongarra, J. J., Meuer, H. W., Strohmaier, E., et al. “TOP500 supercomputer sites”. In: *Supercomputer* vol. 13 (1997), pp. 89–111.

[25] Dooley, K. “Simulation research methods”. In: *Companion to organizations* (2002), pp. 829–848.

[26] Dutton, J. M. and Starbuck, W. H. “Computer simulation models of human behavior: A history of an intellectual technology”. In: *IEEE Transactions on Systems, Man, and Cybernetics*, no. 2 (1971), pp. 128–171.

[27] Filippone, S. et al. “Sparse Matrix-Vector Multiplication on GPGPUs”. In: *ACM Transactions on Mathematical Software* vol. 43, no. 4 (Jan. 2017), 30:1–30:49.

[28] El-Ghazawi, Tarek et al. “UPC Benchmarking Issues”. In: *Parallel Processing, 2001. International Conference on*. IEEE. 2001, pp. 365–372.

[29] El-Ghazawi, Tarek et al. “UPC Performance and Potential: A NPB Experimental Study”. In: *Supercomputing, ACM/IEEE 2002 Conference*. IEEE. 2002, pp. 17–17.

[30] El-Ghazawi, T. A. et al. “Evaluation of UPC on the Cray X1”. In: *Cray User Group Proceedings*. 2005.

[31] Gropp, W. et al. “A high-performance, portable implementation of the MPI message passing interface standard”. In: *Parallel computing* vol. 22, no. 6 (1996), pp. 789–828.

[32] Hager, G. and Wellein, G. *Introduction to High Performance Computing for Scientists and Engineers*. 1st. Boca Raton, FL, USA: CRC Press, Inc., 2010.

[33] Hewlett-Packard. *ccNon-Uniform Memory Access - HP Systems*. support.hpe.com/hpsc/doc/public/display?docId=emr_na-c02670417. [Online; accessed 16-December-2019]. 2003.

[34] Hong, C. et al. “Efficient sparse-matrix multi-vector product on GPUs”. In: *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*. ACM. 2018, pp. 66–79.

[35] Intel. *Intel PCM Official Webpage*. software.intel.com/en-us/articles/intel-performance-counter-monitor. last accessed on 18/12/2019.

[36] Intel. *Intel® Xeon Phi™ Processor 7250*. ark.intel.com/content/www/us/en/ark/products/94035/intel-xeon-phi-processor-7250-16gb-1-40-ghz-68-core.html. [Online; accessed 16-December-2019]. 2016.

[37] Intel Software. *Intel Math Kernel Library*. software.intel.com/en-us/mkl. [Online; accessed 17-December-2019].

[38] Jeffers, J. and Reinders, J. *Intel Xeon Phi coprocessor high performance programming*. Newnes, 2013.

[39] Jérémie Lagravière. *Generic Heterogeneous Operating System*. [Online; accessed 16-December-2019]. 2014.

[40] Jin, H.-Q., Frumkin, M., and Yan, J. “The OpenMP implementation of NAS parallel benchmarks and its performance”. In: (1999).

[41] Jin, H., Hood, R., and Mehrotra, P. “A practical study of UPC using the NAS Parallel Benchmarks”. In: *Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*. ACM. 2009, p. 8.

[42] Kaixi Hou, Wu-chun Feng, Shuai Che. *Auto-Tuning Strategies for Parallelizing Sparse Matrix-Vector (SpMV) Multiplication on Multi- and Many-Core Processors*. [Online; accessed 07-January-2020]. 2020.

[43] Kamil, S., Shalf, J., and Strohmaier, E. “Power efficiency in high performance computing”. In: *2008 IEEE International Symposium on Parallel and Distributed Processing*. IEEE. 2008, pp. 1–8.

[44] Lagraviere, J. et al. “On the performance and energy efficiency of the pgas programming model on multicore architectures”. In: *2016 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE. 2016, pp. 800–807.

[45] Lagravière, J. et al. “Performance Optimization and Modeling of Fine-Grained Irregular Communication in UPC”. In: *Scientific Programming* vol. 2019 (2019).

[46] Lasse Amundsen, Martin Landrø and Børge Arntsen. *Supercomputers for Beginners - Part II*. www.geoexpro.com/articles/2016/01/supercomputers-for-beginners-part-ii. [Online; accessed 08-January-2020]. 2016.

[47] LBNL and UC Berkeley. *UPC publications*. upc.lbl.gov/publications/. last accessed on 18/12/2019.

[48] Lepak, K. et al. "The next generation amd enterprise server product architecture". In: *IEEE hot chips* vol. 29 (2017).

[49] LeVeque, R. J. et al. *Computational Methods for Astrophysical Fluid Flow: Saas-Fee Advanced Course 27. Lecture Notes 1997 Swiss Society for Astrophysics and Astronomy*. Vol. 27. Springer Science & Business Media, 2006.

[50] Liskov, B. and Shrira, L. *Promises: linguistic support for efficient asynchronous procedure calls in distributed systems*. Vol. 23. 7. ACM, 1988.

[51] Mallón, Damián A et al. "Performance evaluation of MPI, UPC and OpenMP on multicore architectures". In: 2009.

[52] NVIDIA Developer. *CUSP*. developer.nvidia.com/cusp. [Online; accessed 17-December-2019].

[53] NVIDIA Developer. *cuSPARSE*. developer.nvidia.com/cusparse. [Online; accessed 17-December-2019].

[54] "Partitioned Global Address Space (PGAS) Languages". In: *Encyclopedia of Parallel Computing*. Ed. by Padua, D. Boston, MA: Springer US, 2011, pp. 1465–1465.

[55] Pinar, A. and Heath, M. T. "Improving performance of sparse matrix-vector multiplication". In: *SC'99: Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*. IEEE. 1999, pp. 30–30.

[56] Pletcher, R. H., Tannehill, J. C., and Anderson, D. *Computational fluid mechanics and heat transfer*. CRC press, 2012.

[57] Prugger, M. "High-resolution numerical schemes for hyperbolic conservation laws, and their performance on modern HPC architectures". PhD dissertation. University of Innsbruck, 2017.

[58] Rau, B. R. et al. "The Cydra 5 departmental supercomputer: Design philosophies, decisions, and trade-offs". In: *Computer* vol. 22, no. 1 (1989), pp. 12–35.

[59] Reinders, J. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. " O'Reilly Media, Inc.", 2007.

[60] Saad, Y. *SPARSKIT: a basic tool kit for sparse matrix computations*. Tech. rep. RIACS-TR-90-20. Research Institute for Advanced Computer Science, May 1990.

[61] Saule, E. e. "Performance evaluation of sparse matrix multiplication kernels on Intel Xeon phi". In: *International Conference on Parallel Processing and Applied Mathematics*. Springer. 2013, pp. 559–570.

[62] Serres, O. et al. “Experiences with UPC on TILE-64 processor”. In: *2011 Aerospace Conference*. IEEE. 2011, pp. 1–9.

[63] Shan Tang. *AI Chip (ICs and IPs)*. github.com/basicmi/AI-Chip. 2020.

[64] Tikir, M. M. et al. “A genetic algorithms approach to modeling the performance of memory-bound computations”. In: *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. ACM. 2007, p. 47.

[65] UPC++. *UPC++ Version Changelog*. bitbucket.org/berkeleylab/upcxx/wiki/ChangeLog. Online; accessed 28-October-2019.

[66] Vazquez, F. et al. “Improving the performance of the sparse matrix vector product with GPUs”. In: *2010 10th IEEE International Conference on Computer and Information Technology*. IEEE. 2010, pp. 1146–1151.

[67] Widder, D. V. *The heat equation*. Vol. 67. Academic Press, 1976.

[68] Wikipedia. *Computer cluster — Wikipedia, The Free Encyclopedia*. en.wikipedia.org/w/index.php?title=Computer%20cluster. [Online; accessed 08-January-2020]. 2020.

[69] Wikipedia. *Graph500 — Wikipedia, The Free Encyclopedia*. en.wikipedia.org/w/index.php?title=Graph500. [Online; accessed 28-December-2019]. 2019.

[70] Wulf, W. A. and McKee, S. A. “Hitting the memory wall: implications of the obvious”. In: *ACM SIGARCH computer architecture news* vol. 23, no. 1 (1995), pp. 20–24.

[71] Yelick, Katherine et al. “Productivity and Performance Using Partitioned Global Address Space Languages”. In: 2007.

[72] Zhang, Z. and Seidel, S. “Benchmark measurements of current UPC platforms”. In: *19th IEEE International Parallel and Distributed Processing Symposium*. IEEE. 2005, 8–pp.

[73] Zheng, Y. et al. “UPC++: a PGAS extension for C++”. In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE. 2014, pp. 1105–1114.

[74] Zoran Budimlić, Mack Joyner. *Fundamentals of Parallel Programming*. wiki.rice.edu/confluence/display/PARPROG/COMP322. last accessed on 18/12/2019.

Papers

Paper I

On the Performance and Energy Efficiency of the PGAS Programming Model on Multicore Architectures

Jérémie Lagravière, Johannes Langguth, Mohammed Sourouri, Phuong H. Ha, Xing Cai

Published in *International Conference on High Performance Computing & Simulation (HPCS)*, 18-22 July 2016. DOI: 10.1109/HPCSim.2016.7568416

On the Performance and Energy Efficiency of the PGAS Programming Model on Multicore Architectures

Jérémie Lagravière & Johannes Langguth
Simula Research Laboratory
NO-1364 Fornebu, Norway
jeremie@simula.no
langguth@simula.no

Mohammed Sourouri
NTNU
Norwegian University of Science and Technology
NO-7491 Trondheim, Norway
mohammed.sourouri@iet.ntnu.no

Phuong H. Ha
The Arctic University of Norway
NO-9037 Tromsø, Norway
phuong.hoai.ha@uit.no

Xing Cai
Simula Research Laboratory
NO-1364 Fornebu, Norway
xingca@simula.no

***Abstract*—Using large-scale multicore systems to get the maximum performance and energy efficiency with manageable programmability is a major challenge. The partitioned global address space (PGAS) programming model enhances programmability by providing a global address space over large-scale computing systems. However, so far the performance and energy efficiency of the PGAS model on multicore-based parallel architectures have not been investigated thoroughly. In this paper we use a set of selected kernels from the well-known NAS Parallel Benchmarks to evaluate the performance and energy efficiency of the UPC programming language, which is a widely used implementation of the PGAS model. In addition, the MPI and OpenMP versions of the same parallel kernels are used for comparison with their UPC counterparts. The investigated hardware platforms are based on multicore CPUs, both within a single 16-core node and across multiple nodes involving up to 1024 physical cores. On the multi-node platform we used the hardware measurement solution called High definition Energy Efficiency Monitoring tool in order to measure energy. On the single-node system we used the hybrid measurement solution to make an effort into understanding the observed performance differences, we use the Intel Performance Counter Monitor to quantify in detail the communication time, cache hit/miss ratio and memory usage. Our experiments show that UPC is competitive with OpenMP and MPI on single and multiple nodes, with respect to both the performance and energy efficiency.**

I. Introduction & Motivation

The overarching complexity of parallel programming is one of the fundamental challenges that the HPC research community faces. In the last decade, the Partitioned Global Address Space model (PGAS) has been established as one possible solution for this problem. It promises improved programmability while maintaining high performance, which is the primary goal in HPC. In recent years, energy efficiency has become an additional goal. Optimizing energy efficiency without sacrificing computational performance is the key challenge of energy-aware HPC, and an absolute requirement for attaining Exascale computing in the future.

In this study we investigate whether PGAS can meet the goals of performance and energy efficiency. We focus on UPC, one of the most widely used PGAS implementations, and compare it to MPI and OpenMP. OpenMP offers ease of programming for shared memory machines, while MPI offers high performance on distributed memory supercomputers.

PGAS combines these advantages through a simple and unified memory model. On a supercomputer, this means that the programmer can access the entire memory space as if it is a single memory space that encompasses all the nodes. Through a set of functions that makes data *private* or *shared*, PGAS languages ensure data consistency across the different memory regions. When necessary, shared data is transferred automatically between the nodes through a communication library such as GASnet [1].

Recent studies [2], [3] advocate the use of PGAS as a promising solution for HPC. Many have focused on the evaluation of PGAS performance and UPC in particular [4]–[10]. However, the previous UPC studies have not taken energy efficiency into consideration. This motivates us to investigate UPC's energy efficiency and performance using the latest CPU architecture with advanced support for energy and performance profiling.

For our evaluation we use the well-established NAS Benchmark. We use MPI, OpenMP, and UPC implementations [11], [12] to compare the performance and energy efficiency of the different programming models. The energy measurements of the single-node system are obtained by using Intel PCM [13]. The multi-node performance measurements are obtained on an Intel Xeon based supercomputer , the energy measurement are obtained on this platform by using High Definition Energy Efficiency Monitoring (HDEEM) [14]. We provide an analysis of the single-node measurements in order to explain the difference in performance and energy efficiency, by focusing on the cache performance and the memory traffic of MPI, OpenMP and UPC.

This paper improves upon previous works [4]–[6], [8]–[10] by: (1) providing measurements for a larger number of nodes and cores for MPI, OpenMP and UPC on recent single-node and multi-node systems (up to 1024 physical cores); (2) including energy measurements obtained on both a single-node system and a multi-node system; (3) making an effort to understand the differences in energy efficiency and

performance between UPC, OpenMP and MPI.

The remainder of this paper is organized as follows: Section II briefly presents the UPC framework and why we have chosen this programming language. Section III describes the benchmark chosen for this study. Section IV explains the hardware and software set-up used for running our experiments, the results of which are presented in Section V and discussed in Section VI. Section VII concludes the paper.

II. PGAS Paradigm and UPC

PGAS is a parallel programming model that has a logically partitioned global memory address space, where a portion of it is local to each process or thread. A special feature of PGAS is that the portions of the shared memory space may have an affinity for a particular process, thereby exploiting locality of reference [15], [16].

In the PGAS model, each node has access to both private and shared memory. Accessing the shared memory to either read or write data can imply inter-node communication which is handled automatically by the runtime. Remote access to memory work in an RDMA (Remote Direct Memory Access) fashion, using one-sided communication. However, most PGAS languages are built over a low-level communication layer which limits their physical capabilities. Thus, RDMA is available only if the underlying hardware and software support it.

In recent years several languages implementing the PGAS model have been proposed. UPC, which is essentially an extension of the C language, was one of the first ones and also one of the most stable [17]. Other members of the PGAS family of languages include the Fortran counterpart, Coarray Fortran [18], X10 [19], and Cray Chapel [20]. In addition, libraries such as Global Arrays [21] and SHMEM/OpenSHMEM [22] which implement PGAS functionality are available.

The key characteristics of UPC are: a parallel execution model of Single Program Multiple Data (SPMD); distributed data structures with a global addressing scheme, with static or dynamic allocation; operators on these structures, with affinity control; and copy operators between private, local shared, and distant shared memories.

Additionally, multiple open-source implementations of the UPC compiler and runtime environment are available, in particular Berkeley UPC [23], GCC/UPC [24] and CLANG/UPC [25].

III. The NAS Benchmark

The NAS Benchmark [26] consists of a set of kernels that each provides a different way of testing the capabilities of a supercomputer. The NAS Benchmark was originally implemented in Fortran and C. We use both the Fortran and C implementations for OpenMP and MPI, as well as the UPC version of the benchmark [12]. For our study, we select four kernels: Integer Sort (IS), Conjugate Gradient (CG), Multi-Grid (MG), and Fourier Transformation (FT).

CG refers to the *conjugate gradient* method used to compute an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix.

MG is a simplified *multigrid* kernel. Multigrid (MG) methods in numerical analysis solve differential equations using a hierarchy of discretizations.

FT is a three-dimensional partial differential equation solver using *Fast Fourier Transformations*. This kernel performs the essence of many spectral codes. It is a rigorous test of all-to-all communication performance.

IS represents a large *integer sort*. This kernel performs a sorting operation that is important in particle method codes. It evaluates both integer computation speed and communication performance.

CG, IS, MG and FT are selected because they are the most relevant ones: stressing memory, communication and computation. They involve very different communication patterns, which is important for evaluating the performance of the selected languages (see Section V). The other kernels in the NAS Benchmark are of limited relevance to this study. See [26] for their descriptions.

IV. Experimental Setup

In this section we describe the software and hardware solutions that we used to carry out our experiments. We ran the NAS kernels both on a single-node machine and on Taurus [27], a supercomputer operated by Dresden University of Technology, using a varying number of cores and nodes.

A. Hardware

Table I shows the specifications of the systems used in our experiments. The single-node system is equipped with Intel Sandy Bridge processors and the multi-node supercomputer is equipped with Intel Haswell processors.

TABLE I
Experimental setup: hardware

	Single Node	Multi Node
CPUs	2	2 (per node)
Cores	16	24 (per node)
CPU model	Intel Xeon E5-2650 @ 2.00GHz	Intel Xeon E5-2680 v3 @ 2.50GHz
Interconnect	N/A	Infiniband: 6.8 GB/s

B. Software

On the single-node machine we used UPC version 2.22.0, the Intel Compiler version 15.0.1, and MPI Library 5.0 Update 2 for Linux. On the multi-node supercomputer we used UPC version 2.22.0 and using Bull XMPI version 1.2.8.4.

1) UPC: On the multi-node supercomputer, the UPC compiler and runtime were built with a specific option enable-segment-large in order to support large memory systems. The UPC applications were compiled using a fix number of threads and a fix network choice: *upcc -T1024 -network=mxm*. The UPC implementations were run using a fixed number of threads and fixed shared heap size and thread binding: *upcrun -n number_of_processes -bind-threads -shared-heap=3765MB ./application*. On the single-node system the UPC application were using the symmetric multiprocessing (smp) network conduit. The following environment variables were defined for *UPC GASNET_PHYSMEM_MAX=63G* which indicates to GASNET [1] to use 63GB of RAM on

each node. We also used the environment variable *GASNET_PHYSMEM_NOPROBE=1* which indicates to GASNET to avoid memory detection on each node. Except for FT-D-16 where we used a modified GASNET environment variable: *GASNET_PHYSMEM_MAX=254G* because FT requires more RAM than the other kernels.

2) MPI: On the multi-node supercomputer and single-node system MPI applications were compiled using the *-O3 -mcmodel=medium* flag in order to handle larger data in memory.

3) OpenMP: On the single-node system we used OpenMP version 4.0 and we used *numactl* and the *OMP_NUM_THREADS* environment variable to bind the threads and define the number of threads.

4) Benchmarking: For each measurement we executed three separate runs and reported the best result. Doing so filters out the OS interference.

On the single-node machine we used size Class **C** [11], [28] for each kernel. For CG, Class C, the number of rows is 150000. For MG, Class C, the grid size is $512 \times 512 \times 512$. For FT, Class C, the grid size is $512 \times 512 \times 512$. For IS, Class C, the number of keys is 2^{27}. On the multi-node supercomputer we used for each kernel Class **D**, except for IS which is not available in Class **D** in the UPC implementation [12]. For CG, Class D, the number of rows is 1500000. For MG, Class D, the grid size is $1024 \times 1024 \times 1024$. For FT, Class D, the grid size is $2048 \times 1024 \times 1024$.

In our study the comparison between the different implementations is fair as the number of operations (expressed in Million Of Operation - MOP) is identical in all implementations (OpenMP / MPI / UPC). The number of MOP is reported by the benchmarks. For example, the number of MOP for CG in size C is precisely 143300 for all implementations (OpenMP, MPI and UPC).

Each kernel was run using up to 1024 CPU cores and thus 64 nodes of the supercomputer. However, for CG, limitations in the UPC implementation prevent us from using more than 256 cores, see Figure 1.

Sizes **C** and **D** provide data sets that are sufficiently large to exceed the cache size of the test systems [29] [28].

5) Thread Binding: Thread binding or thread pinning is an approach that associates each thread with a specific processing element. In our experiments we applied thread/process binding to the physical cores.

C. Energy Measurements

1) Multi-Node Platform: On the multinode platform we have chosen High Definition Energy Efficiency Monitoring (HDEEM) [14]. HDEEM is a hardware based solution for energy measurements, meaning that additional hardware is used to measure energy in a supercomputer. HDEEM is an intra-node measurement tool, which indicates that the additional hardware that performs the energy measurements is located inside each node of the supercomputer [30]. The authors in [14] define four criteria, including spatial and temporal granularity, accuracy and scalability, for power and energy measurement. HDEEM can achieve an accuracy of 99.5% over 270 nodes by using an appropriate filtering approach to prevent the aliasing effect. HDEEM is based on an FPGA solution to achieve spatial fine-granularity by measuring every blade, CPU and DRAM power separately, with a sampling rate of 1,000 Sa/s over 500 nodes [31]. Our measurements do not take into account the energy consumption of the network between the nodes of the multi-nodes platform.

2) Single-Node Platform: On the single-node platform we have chosen a software based solution in order to measure the CPU and RAM energy consumption and memory and cache usage. Intel Performance Monitor (Intel PCM) is used for the energy efficiency experiments on the single-node platform and to measure memory and cache usage [13]. Intel PCM uses the Machine Specific Registers (MSR) and RAPL counters to disclose the energy consumption details of an application [32]. Intel PCM is able to identify the energy consumption of the CPU(s) and the RAM. Quick-Path Interconnect energy consumption is not taken into account because Intel PCM was unable to provide measurement on the chosen hardware platform.

The RAPL values do not result from physical measurement. They are based on a modeling approach that uses a "set of architectural events from each core, the processor graphics, and I/O, and combines them with energy weights to predict the package's active power consumption" [33]. Previous studies have demonstrated that using a counter-based model is reasonably accurate [34]–[37]. The RAPL interface returns energy data. There is no timestamp attached to the individual updates of the RAPL registers, and no assumptions besides the average update interval can be made regarding this timing. Therefore, no deduction of the power consumption is possible other than averaging over a fairly large number of updates. For example, averaging over only 10 ms would result in an unacceptable inaccuracy of at least 10% due to the fact that either 9, 10, or 11 updates may have occurred during this short time period [38]. In our experiments the execution time of each kernel is always above a second, thus estimating the power data by averaging reported Joules over time is accurate. For convenience, in our study, we use the word "measurement" to mention the values reported by Intel PCM.

We do not track and report the energy consumption curve along each kernel's execution time line, instead we choose to report the total energy consumption ($Joules$) and the average power consumption ($Watt = Joules/seconds$).

V. Results

In this paper, we consider two metrics to measure the performance and energy efficiency. To evaluate the performance we use Million Operations Per Second (MOPS). This metric is used for both the multi-node measurements and single-node measurements. To evaluate the energy efficiency, Millions Operations Per Seconds over Watts (MOPS per Watt) is used as the energy efficiency metric. This metric is used for both the multi-node measurements and single-node measurements. The *500 Green - Energy Efficient High Performance*

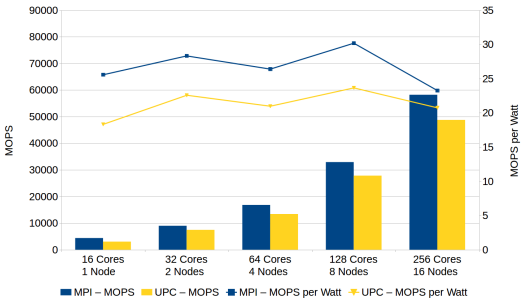


Fig. 1. Multi-node performance and energy efficiency of the CG kernel - Class D

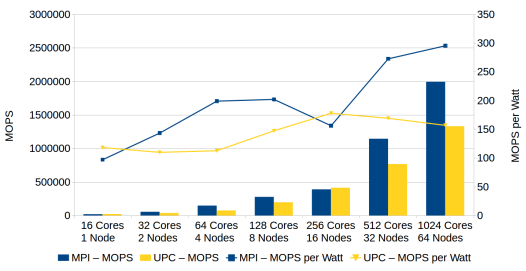


Fig. 2. Multi-node performance and energy efficiency of the MG kernel - Class D

Computing Power Measurement Methodology [39] advises this measurement methodology. We remark that MOPS per Watt is equivalent to MOP/Joules: $(MOP/seconds)/Watt = (MOP/seconds)/(Joules/seconds) = MOP/Joules$

For single-node measurements, we use the following notation: [kernel name]-[number of threads/processes]-[1S / 2S]. For instance, *CG-8-2S* stands for "Conjugate Gradient kernel running on 8 threads (or processes for MPI) spread over two sockets".

For multi-node measurements, we use the following notation [kernel name]-[class]-[number of threads/processes]. For instance, MG-D-256 stands for the "Multigrid kernel of class D running on 256 threads (or processes for MPI)". Each node always used the maximum number of physical cores, *i.e.* 16.

A. Measurement on Multi-Node Architecture

To the best of our knowledge, this is the first investigation of UPC's energy efficiency. Our experimental results show that the energy efficiency of UPC, MPI, and OpenMP implementations scale over the number of cores and are comparable to each other. Figures 1-4 show the multi-node performance expressed in MOPS and the energy efficiency expressed in MOPS per Watt, for the four kernels implemented in UPC and MPI. Each kernel ran on up to 1024 cores, except CG where the UPC implementation cannot run on more than 256 threads.

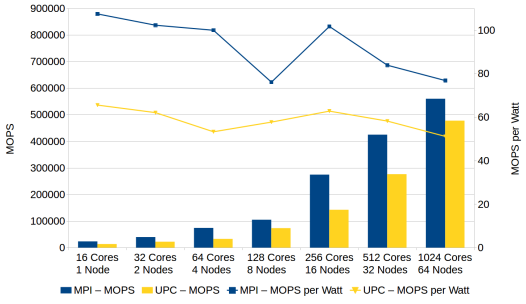


Fig. 3. Multi-node performance and energy efficiency of the FT kernel - Class D

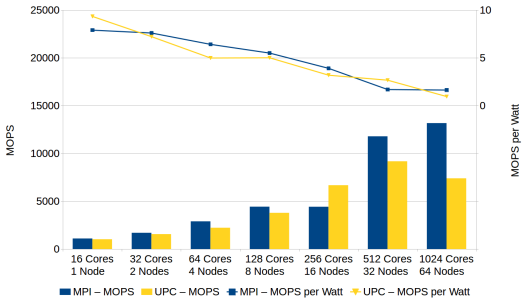


Fig. 4. Multi-node performance and energy efficiency of the IS kernel - Class C. On the left Y-axis the scale has been shifted for readability purposes.

The performance results (bars) show that the CG, MG and FT kernels scale over the number of cores independently of the language. MPI is a clear winner when running on more than 32 cores, however UPC achieves a performance that is close to that of MPI, particularly for the CG and MG kernels. These results match previous studies, in particular [9].

IS is aside in terms of performance, because of the size of the data to process, size C, is smaller than size D and causes both UPC and MPI not being able to scale on more than 256 cores. For 512 and 1024 cores runs, IS-C does not deliver good performance as the communication cost outbalances the computation performance.

Figures 1-4 show a complex relation between performance (bars) and energy efficiency (lines). While the performance of both UPC and MPI go up with increasing numbers of cores and nodes, the energy efficiency is at best staying constant or diminishing. The only exception is the MG benchmark, where higher energy efficiency is achieved by using more cores and nodes. For the IS benchmark that uses the size of Class C, in particular, the energy measurements obtained by HDEEM are gradually dominated by the non-scalable initialization phase as the number cores increases. (In comparison, the energy measurements obtained by Intel PCM on the single-node system do not include the initialization phase.)

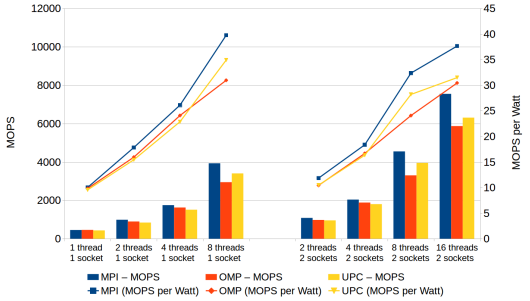


Fig. 5. Single-node performance and energy efficiency of the CG kernel - Class C

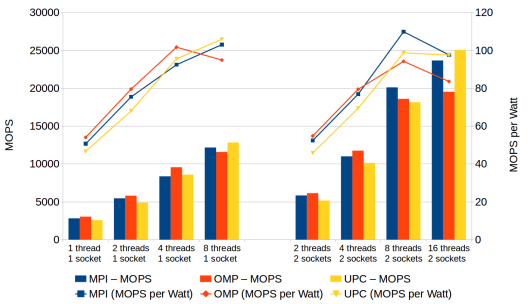


Fig. 6. Single-node performance and energy efficiency of the MG kernel - Class C

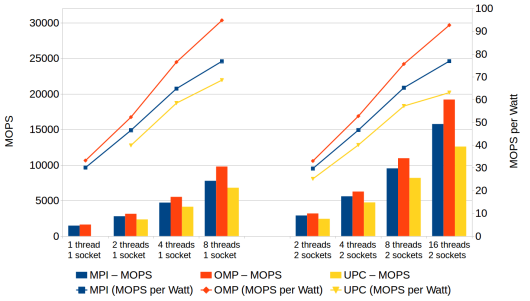


Fig. 7. Single-node performance and energy efficiency of the FT kernel - Class C

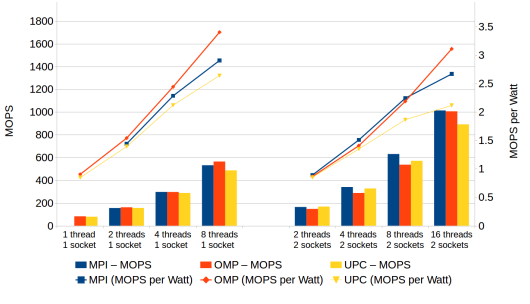


Fig. 8. Single-node performance and energy efficiency of the IS kernel - Class C

B. Measurements on Single-Node Architecture

Our experimental results show that the energy efficiency of UPC, MPI, and OpenMP implementations scale over the number of cores and are comparable to each other. In this section we give more details about these results and in Section VI we provide an analysis of the difference in performance between UPC, OpenMP, and MPI.

Figures 5-8 show both the performance expressed in MOPS for the four kernels (bars) and the energy efficiency expressed in MOPS per Watt (lines). Each kernel ran on up to 16 cores. As the single-node system is equipped with two CPUs we ran the kernels on both one socket and two sockets for 2,4 and 8 threads counts. Each of these figure shows the results for all three programming models. The energy efficiency results show that the kernels scale over the number of threads/cores independently of the language. In Figures 5-8, when measured on one-socket, the energy consumption of the idle socket and memory controller were not taken into account. There is no clear winner since none of the chosen languages is better than the other two competitors for all the kernels.

UPC was not able to run FT-C on one thread and MPI was not able run IS-C on one thread.

Even though there is no global *winner* in the obtained single-node measurements, UPC is able to compete with both OpenMP and MPI. UPC arrives in second place for CG-8-1S, CG-8-2S, CG-16-2S, IS-2-2S, IS-4-2S, IS-8-2S and MG-4-1S.

As described in [9], on a single-node platform, UPC scales well over more CPU cores and competes well with OpenMP and MPI. However, the performance of MPI or OpenMP is better in many cases.

For general-purpose architectures with high static power such as Intel Sandy Bridge, energy efficiency is directly connected to performance results. Therefore, the best results in energy efficiency are achieved, in most cases, for the kernels and thread-counts delivering the highest performance in MOPS.

However, by looking closely at the performance and performance per watt, it is possible to highlight that running a program in MPI, OpenMP or UPC over only 1 socket instead of two, when it is possible, delivers better energy efficiency. In Table II, we report values for CG in order to show a comparison between runs over 2,4 and 8 cores using 1 or 2 sockets. Table II is divided in two parts: the first part where the measurements on 1 socket do not include the measurement of the second idling socket, the second part where the idle socket measurements are included in the reported values. Table II corresponds to Figure 5, we chose to only represent the value of CG because the results of CG in terms of energy

TABLE II
COMPARISON OF COMPUTATION AND ENERGY PERFORMANCE OF CG OVER 2,4, AND 8 THREADS USING 1 AND 2 SOCKETS.
SYNTAX: *CG-2-2S / CG-2-1S* STANDS FOR "MEASUREMENTS FROM CG RUNNING OVER 2 THREADS USING 2 SOCKETS COMPARED TO MEASUREMENTS FROM CG RUNNING OVER 2 THREADS USING 1 SOCKET"

Second socket not included when not used	MPI			OpenMP			UPC		
CG	Performance gain in %	Additional energy usage in %	Change in MOPS / Watt in %	Performance gain in %	Additional energy usage in %	Change in MOPS / Watt in %	Performance gain in %	Additional energy usage in %	Change in MOPS / Watt in %
CG-2-2S / CG-2-1S	9.64	50.36	-33.49	8.92	52.85	-34.57	13.50	47.20	-32.06
CG-4-2S / CG-4-1S	16.53	42.22	-29.69	15.84	44.42	-30.76	19.18	40.03	-28.59
CG-8-2S / CG-8-1S	15.71	22.83	-18.58	12.07	28.63	-22.26	16.14	23.85	-19.24

Second socket always included even when not used	MPI			OpenMP			UPC		
CG	Performance gain in %	Additional energy usage in %	Change in MOPS / Watt in %	Performance gain in %	Additional energy usage in %	Change in MOPS / Watt in %	Performance gain in %	Additional energy usage in %	Change in MOPS / Watt in %
CG-2-2S / CG-2-1S	9.64	8.86	-8.14	8.92	10.02	-9.10	13.50	5.28	-5.01
CG-4-2S / CG-4-1S	16.53	7.83	-7.26	15.84	8.68	-8.00	19.18	4.79	-4.58
CG-8-2S / CG-8-1S	15.71	0.63	-0.62	12.07	3.55	-3.43	16.14	0.23	-0.21

efficiency difference between one socket and two sockets are representative of all the kernels. In the first part of Table II, we can see that CG-2-2S in MPI (CG running over two threads using two sockets) delivers better performance than CG-2-1S in MPI (CG running over two threads using one socket): +9.64% MOPS, but it costs +50,36% in energy (Joules) and delivers -33.49% MOPS per Watt. In the second part of Table II, we can see that the same comparison, including values from the second idle socket, gives a similar conclusion as before: using two sockets consumes +8.86% energy and delivers -8.14% in MOPS per Watt. These observations are also visible in Figure 5, 6, 7 and 8: the lines representing the MOPS per Watt are higher for kernels using one socket for equivalent threads/process count than that of kernels using two sockets.

VI. DISCUSSION

In this section, we give an analysis of the differences in performance and energy efficiency that were observed in the previous section.

Intel PCM provides access to metrics such as memory traffic and hit and miss rates for L2 and L3 cache. In this section we will use measurements of these metrics to analyze the differences in performance and energy efficiency among OpenMP, MPI, and UPC. Table III shows the measurements obtained via Intel PCM. In Table III, we use different metrics: Memory traffic ($read + write$) expressed in GigaBytes, L2 and L3 cache hit ratio, and L3 access given in millions of accesses. The results are given for each kernel (CG, MG, FT and IS) over 1, 2, 4 and 8 cores using one socket and 2, 4, 8 and 16 cores using two sockets.

By using Table III, we can for instance analyze the performance and energy efficiency of UPC, OpenMP and MPI for CG-4-2S and CG-8-2S presented in Figure 5. In CG-8-2S, MPI obtains the best performance and energy efficiency, UPC is second in performance and energy efficiency and OpenMP is third. By using the data from Table III it is possible to explain this result: UPC has an increased L2 cache hit ratio compared to OpenMP ($0.56 > 0.14$). MPI is better than OpenMP in CG-8-2S, because it has fewer L3 accesses ($11211 < 17023$) and better L2 hit ratio ($0.57 > 0.14$).

In MG-16-2S, UPC obtains the best performance compared to MPI and OpenMP because it has lower memory traffic than OpenMP ($452.36\ GB < 520.69\ GB$) and MPI ($452.36\ GB < 487.4\ GB$), better L3 cache hit ratio than OpenMP ($0.28 > 0.19$) and MPI ($0.28 > 0.2$) and fewer L3 accesses than OpenMP ($2782 < 3663$) and MPI ($2782 < 4390$).

In FT-16-2S, OpenMP obtains the best performance and energy efficiency due to having lower memory traffic than UPC ($497.33\ GB < 1009\ GB$) and MPI ($497.33\ GB < 878.8\ GB$), better L3 hit ratio than UPC ($0.76 > 0.26$) and MPI ($0.76 > 0.43$) and a lower volume of L3 accesses than UPC ($1681 < 22562$) and MPI ($1681 < 8319$).

In IS-16-2S, MPI is better than OpenMP and UPC because it has higher L3 cache hit ratio than OpenMP ($0.13 > 0.04$) and UPC ($0.13 > 0.07$) and a lower level of L3 accesses than OpenMP ($2815 < 4550$) and UPC ($2815 < 5143$). UPC in IS-8-2S wins over OpenMP because of its slightly better L3 cache hit ratio ($0.07 > 0.04$).

Globally the results presented show a correlation between the number of cores used and the achieved power efficiency. However we noticed in Table II a possible trade-off between performance and performance per watt by running programs over 1 or 2 sockets depending on what is the chosen goal (pure performance or energy efficiency). We used Table III to analyze the difference in performance between MPI, OpenMP and UPC over various threads/processes counts and sockets counts. We saw that by considering the memory traffic, L3 cache hit ratio, L2 cache hit ratio and the volume of access to L3 cache, that more insight into the performance can be obtained.

VII. CONCLUSION

In this study, we have investigated and provided insights into UPC energy efficiency and performance using the latest CPU architecture with advanced support for energy and perfor-

TABLE III
MEMORY TRAFFIC, L3 CACHE HIT RATION AND L2 CACHE HIT RATIO, FOR CG, MG, FT AND IS IMPLEMENTED IN MPI, OPENMP AND UPC
RESULTS ARE PRESENTED FOR 2,4 AND 8 CORES USING ONE AND TWO SOCKETS AND 16 CORES USING TWO SOCKETS

	MPI				OpenMP				UPC			
CG - Size C	Mem traffic (GB)	L3 hit ratio	L3 access $\times 10^6$	L2 hit ratio	Mem traffic (GB)	L3 hit ratio	L3 access $\times 10^6$	L2 hit ratio	Mem traffic (GB)	L3 hit ratio	L3 access $\times 10^6$	L2 hit ratio
1 Threads - 1 socket	833.52	0.81	16049	0.14	834.71	0.81	16049	0.14	834.58	0.81	16049	0.14
2 Threads - 1 socket	846.14	0.77	16883	0.27	836.03	0.81	16049	0.14	856.01	0.77	16883	0.27
2 Threads - 2 sockets	845.22	0.77	17391	0.27	841.93	0.81	16798	0.14	854.4	0.77	17725	0.27
4 Threads - 1 socket	854.25	0.77	16883	0.27	834.26	0.81	16049	0.14	878.64	0.77	16883	0.27
4 Threads - 2 sockets	859.01	0.77	17474	0.27	842.96	0.81	16823	0.14	881.10	0.77	17817	0.27
8 Threads - 1 socket	878.89	0.74	11345	0.56	832.17	0.81	16049	0.14	909.09	0.73	12042	0.56
8 Threads - 2 sockets	881.81	0.74	11211	0.57	841.31	0.80	17023	0.14	912.42	0.73	11827	0.56
16 Threads - 2 sockets	908.34	0.74	11535	0.565	840.23	0.80	17014	0.14	968.8	0.72	12439	0.56

	MPI				OpenMP				UPC			
MG - Size C	Mem traffic (GB)	L3 hit ratio	L3 access $\times 10^6$	L2 hit ratio	Mem traffic (GB)	L3 hit ratio	L3 access $\times 10^6$	L2 hit ratio	Mem traffic (GB)	L3 hit ratio	L3 access $\times 10^6$	L2 hit ratio
1 Threads - 1 socket	388.53	0.38	1174	0.83	372.25	0.31	1103	0.87	369.09	0.52	810	0.78
2 Threads - 1 socket	459.01	0.35	1540	0.82	437.43	0.28	1439	0.85	435.71	0.48	1100	0.75
2 Threads - 2 sockets	383.96	0.42	1005	0.83	370.92	0.31	1084	0.87	370.87	0.53	794	0.78
4 Threads - 1 socket	545.86	0.26	3031	0.76	513.15	0.23	2078	0.84	436.42	0.46	1178	0.75
4 Threads - 2 sockets	406.04	0.39	1275	0.81	434.59	0.27	1448	0.86	373.88	0.52	850	0.77
8 Threads - 1 socket	476.24	0.20	4355	0.33	521.27	0.20	3415	0.79	453.60	0.28	2843	0.49
8 Threads - 2 sockets	400.37	0.23	2722	0.38	510.78	0.23	2061	0.84	388.32	0.33	1861	0.57
16 Threads - 2 sockets	487.40	0.20	4390	0.32	520.69	0.19	3663	0.795	452.36	0.28	2782	0.49

	MPI				OpenMP				UPC			
FT - Size C	Mem traffic (GB)	L3 hit ratio	L3 access $\times 10^6$	L2 hit ratio	Mem traffic (GB)	L3 hit ratio	L3 access $\times 10^6$	L2 hit ratio	Mem traffic (GB)	L3 hit ratio	L3 access $\times 10^6$	L2 hit ratio
1 Threads - 1 socket	617.90	0.38	8905	0.56	471.60	0.79	1400	0.43				
2 Threads - 1 socket	852.26	0.50	6244	0.63	470.57	0.79	1437	0.42	958.65	0.31	16610	0.56
2 Threads - 2 sockets	853.49	0.51	6063	0.63	471.54	0.80	1378	0.41	959.68	0.31	16587	0.57
4 Threads - 1 socket	867.25	0.49	6753	0.62	475.96	0.79	1446	0.41	960.74	0.31	16597	0.59
4 Threads - 2 sockets	865.76	0.50	6291	0.64	470.93	0.79	1424	0.41	960.21	0.31	16432	0.58
8 Threads - 1 socket	881.36	0.42	8767	0.64	491.11	0.76	1601	0.43	977.94	0.28	19582	0.56
8 Threads - 2 sockets	873.64	0.48	6941	0.64	475.76	0.78	1463	0.42	975.48	0.30	17667	0.57
16 Threads - 2 sockets	878.80	0.43	8319	0.64	497.33	0.76	1681	0.42	1009	0.26	22562	0.56

	MPI				OpenMP				UPC			
IS - Size C	Mem traffic (GB)	L3 hit ratio	L3 access $\times 10^6$	L2 hit ratio	Mem traffic (GB)	L3 hit ratio	L3 access $\times 10^6$	L2 hit ratio	Mem traffic (GB)	L3 hit ratio	L3 access $\times 10^6$	L2 hit ratio
1 Threads - 1 socket					32.39	0.03	5600	0.63	44.10	0.07	2629	0.62
2 Threads - 1 socket	47.67	0.13	1846	0.53	32.23	0.06	2817	0.62	45.68	0.08	2463	0.60
2 Threads - 2 sockets	47.52	0.13	1800	0.53	33.84	0.03	5833	0.62	45.75	0.07	2771	0.61
4 Threads - 1 socket	50.99	0.13	2115	0.49	32.10	0.06	2833	0.62	49.57	0.06	3900	0.56
4 Threads - 2 sockets	50.80	0.13	2092	0.50	33.33	0.05	3867	0.62	49.50	0.07	3523	0.57
8 Threads - 1 socket	56.27	0.10	3280	0.46	32.22	0.05	3460	0.62	57.78	0.04	8400	0.47
8 Threads - 2 sockets	55.68	0.13	2423	0.46	33.08	0.05	3911	0.62	56.84	0.05	6889	0.49
16 Threads - 2 sockets	60.77	0.13	2815	0.42	33.24	0.04	4550	0.61	62.93	0.07	5143	0.44

mance profiling. We have measured the energy efficiency and the computational performance of four kernels from the NAS Benchmark, both on a single-node system and on a multi-node supercomputer using three different programming models: UPC, MPI, and OpenMP. On the multi-node supercomputer we observed that UPC is almost always inferior to MPI in terms of performance, although UPC scales well to 1024 cores and 64 nodes, the maximum system size used in this study.

From the measurements performed on the single-node system, we observed that by using more cores the performance and the energy efficiency both increase for the four selected kernels on the chosen hardware platform. The conclusion is not the same on the multi-node computer used in our experiments: the energy efficiency, except in one case, is not increasing with higher numbers of cores and nodes.

We would like to highlight that on the single-node system UPC can compete with MPI and OpenMP in terms of both computational speed and energy efficiency. We obtained this

conclusion by analyzing the results produced on the single-node system in order to localize the origin of difference in performance. We found that data locality is the main reason for the difference in performance.

Our conclusions about UPC are compatible with results obtained in previous studies, in particular [8], [9]. We confirm that UPC can compete with both MPI and OpenMP in performance on a single-node.

In addition we provided a thorough analysis of the performance by looking at the L3 cache hit ratio, L2 cache hit ratio, memory traffic and L3 access of MPI, OpenMP and UPC. And we studied the results of UPC, MPI and OpenMP, running the selected kernels from the NAS Benchmarks on 2,4 and 8 cores by using one and two sockets in order to show the interest of the trade-off between energy efficiency and performance.

In future work, we will explore hardware accelerators such as Many Integrated Cores (MIC) and GPUs. These accelerators are well-known for being more energy efficient than CPUs for many applications.

Furthermore, it would be interesting to investigate why UPC, MPI and OpenMP differ in their communication pattern. In order to enhance the energy measurements, we aim for a more fine grained approach of measuring the energy consumption. By studying the energy cost of computation, communication between nodes, and between CPU and memory separately, we can suggest improvements to energy consumption both in the user codes and in the UPC compiler and runtime environment.

ACKNOWLEDGMENTS

This work was supported by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 671657, the European Union Seventh Framework Programme (EXCESS project, grant n°611183) and the Research Council of Norway (PREAPP project, grant n°231746/F20).

REFERENCES

[1] "GASNET Official Webpage," last accessed on 17/12/2015. [Online]. Available: http://gasnet.lbl.gov

[2] Milthorpe, Josh et al., "PGAS-FMM: Implementing a distributed fast multipole method using the X10 programming language," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 3, pp. 712–727, 2014.

[3] Shan, Hongzhang et al., "A Preliminary Evaluation of the Hardware Acceleration of the Cray Gemini Interconnect for PGAS Languages and Comparison with MPI," *SIGMETRICS Perform. Eval. Rev.*

[4] El-Ghazawi, Tarek et al., "UPC Performance and Potential: A NPB Experimental Study," in *Supercomputing, ACM/IEEE 2002 Conference*. IEEE, 2002, pp. 17–17.

[5] ——, "UPC Benchmarking Issues," in *Parallel Processing, 2001. International Conference on*. IEEE, 2001, pp. 365–372.

[6] François, Cantonnet et al., "Performance Monitoring and Evaluation of a UPC Implementation on a NUMA architecture," 2003.

[7] Jose, Jithin et al., "Unifying UPC and MPI runtimes: experience with MVAPICH," in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*. ACM, 2010, p. 5.

[8] Mallón, Damián A et al., "Performance evaluation of MPI, UPC and OpenMP on multicore architectures," 2009.

[9] Shan, Hongzhang et al., "A programming model performance study using the NAS parallel benchmarks," 2010.

[10] Yelick, Katherine et al., "Productivity and Performance Using Partitioned Global Address Space Languages," 2007.

[11] "NAS Benhcmark Official Webpage," last accessed on 18/12/2015. [Online]. Available: http://www.nas.nasa.gov/publications/npb.html

[12] "NAS Benchmark Implemented in UPC," last accessed on 18/12/2015. [Online]. Available: https://threads.hpcl.gwu.edu/sites/npb-upc

[13] Intel, "Intel PCM Official Webpage," last accessed on 18/12/2015. [Online]. Available: https://software.intel.com/en-us/articles/intel-performance-counter-monitor

[14] D. e. a. Hackenberg, "Hdeem: High Definition Energy Efficiency Monitoring," in *Energy Efficient Supercomputing Workshop (E2SC), 2014*. IEEE, 2014.

[15] Coarfa, Cristian et al., "An evaluation of global address space languages: co-array fortran and unified parallel C," 2005.

[16] Wikipedia, "Wikipedia Definition of PGAS - Last accessed on 18/12/2015," 2015.

[17] Marc Tajchman, CEA, "Programming paradigms using PGAS-based languages. Figure used with the courtesy of Marc Tajchman," 2015. [Online]. Available: http://www-sop.inria.fr/manifestations/cea-edf-inria-2011/slides/tajchman.pdf

[18] "Coarray Fortran Official Webpage," last accessed on 17/12/2015. [Online]. Available: https://gcc.gnu.org/wiki/Coarray

[19] "X10 Official Webpage," last accessed on 17/12/2015. [Online]. Available: http://x10-lang.org

[20] "Cray - Chapel Official Webpage," last accessed on 17/12/2015. [Online]. Available: http://chapel.cray.com

[21] "Global Arrays Official Webpage," last accessed on 17/12/2015. [Online]. Available: http://hpc.pnl.gov/globalarrays/papers/

[22] "OpenSHMEM Official Webpage," last accessed on 17/12/2015. [Online]. Available: http://openshmem.org/site/

[23] Berkeley, "UPC Implementation From Berkeley," last accessed on 18/12/2015. [Online]. Available: http://upc.lbl.gov

[24] Intrepid Technology Inc., "UPC Implementation on GCC," last accessed on 18/12/2015. [Online]. Available: http://www.gccupc.org/

[25] "CLANG UPC," last accessed on 18/12/2015. [Online]. Available: https://clangupc.github.io

[26] Bailey, David H et al., "The NAS Parallel Benchmarks," 1991.

[27] UiO, "Taurus SuperComputer Official Webpage," last accessed on 14/04/2015. [Online]. Available: https://tu-dresden.de

[28] "NAS Benhcmark Official Webpage - Problem Size," last accessed on 18/12/2015. [Online]. Available: www.nas.nasa.gov/publications/npb_problem_sizes.html

[29] Wong, Frederick C et al., "Architectural requirements and scalability of the NAS parallel benchmarks," in *Supercomputing, ACM/IEEE 1999 Conference*. IEEE, 1999, pp. 41–41.

[30] F. e. a. Almeida, "Energy measurement tools for ultrascale computing: A survey," *Supercomputing frontiers and innovations*, vol. 2, no. 2, pp. 64–76, 2015.

[31] M. F. e. a. Dolz, "Ardupower: A low-cost wattmeter to improve energy efficiency of hpc applications," in *Green Computing Conference and Sustainable Computing Conference (IGSC), 2015 Sixth International*. IEEE, 2015, pp. 1–8.

[32] Benedict, Shajulin, "Energy-aware performance analysis methodologies for HPC architectures—An exploratory study," *Journal of Network and Computer Applications*, vol. 35, no. 6, pp. 1709–1719, 2012.

[33] E. e. a. Rotem, "Power-management architecture of the intel microarchitecture code-named sandy bridge," *IEEE Micro*, no. 2, 2012.

[34] R. Joseph and M. Martonosi, "Run-time power estimation in high performance microprocessors," in *Proceedings of the 2001 international symposium on Low power electronics and design*. ACM, 2001.

[35] G. Contreras and M. Martonosi, "Power prediction for intel xscale® processors using performance monitoring unit events." IEEE, 2005.

[36] K. e. a. Singh, "Real time power estimation and thread scheduling via performance counters," *ACM SIGARCH Computer Architecture News*, 2009.

[37] B. e. a. Goel, "Portable, scalable, per-core power estimation for intelligent resource management," in *Green Computing Conference, 2010 International*. IEEE, 2010.

[38] D. e. a. Hackenberg, "Power measurement techniques on standard compute nodes: A quantitative comparison," in *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*. IEEE, 2013.

[39] Top Green 500, "The 500 Green - Energy Efficient High Performance Computing Power Measurement Methodology," last accessed on 23/05/2015.

Paper II

Performance Optimization and Modeling of Fine-Grained Irregular Communication in UPC

Jérémie Lagravière, Johannes Langguth, Martina Prugger, Lukas Einkemmer, Phuong Hoai Ha, Xing Cai

Published in *Scientific Programming*, vol. 2019, Article ID 6825728, 20 pages, 2019. DOI: https://doi.org/10.1155/2019/6825728

II

Hindawi
Scientific Programming
Volume 2019, Article ID 6825728, 20 pages
https://doi.org/10.1155/2019/6825728

Research Article

Performance Optimization and Modeling of Fine-Grained Irregular Communication in UPC

Jérémie Lagravière,[1] Johannes Langguth,[1] Martina Prugger,[2] Lukas Einkemmer,[2] Phuong Hoai Ha,[3] and Xing Cai[1,4]

[1]*Simula Research Laboratory, P.O. Box 134, NO-1325 Lysaker, Norway*
[2]*University of Innsbruck, Technikerstraße 13, A-6020 Innsbruck, Austria*
[3]*The Arctic University of Norway, NO-9037 Tromsø, Norway*
[4]*University of Oslo, NO-0316 Oslo, Norway*

Correspondence should be addressed to Xing Cai; xingca@simula.no

Received 26 September 2018; Revised 14 January 2019; Accepted 27 January 2019; Published 3 March 2019

Academic Editor: Manuel E. Acacio Sanchez

The Unified Parallel C (UPC) programming language offers parallelism via logically partitioned shared memory, which typically spans physically disjoint memory subsystems. One convenient feature of UPC is its ability to automatically execute between-thread data movement, such that the entire content of a shared data array appears to be freely accessible by all the threads. The programmer friendliness, however, can come at the cost of substantial performance penalties. This is especially true when indirectly indexing the elements of a shared array, for which the induced between-thread data communication can be irregular and have a fine-grained pattern. In this paper, we study performance enhancement strategies specifically targeting such fine-grained irregular communication in UPC. Starting from explicit thread privatization, continuing with block-wise communication, and arriving at message condensing and consolidation, we obtained considerable performance improvement of UPC programs that originally require fine-grained irregular communication. Besides the performance enhancement strategies, the main contribution of the present paper is to propose performance models for the different scenarios, in the form of quantifiable formulas that hinge on the actual volumes of various data movements plus a small number of easily obtainable hardware characteristic parameters. These performance models help to verify the enhancements obtained, while also providing insightful predictions of similar parallel implementations, not limited to UPC, that also involve between-thread or between-process irregular communication. As a further validation, we also apply our performance modeling methodology and hardware characteristic parameters to an existing UPC code for solving a 2D heat equation on a uniform mesh.

1. Motivation

Good programmer productivity and high computational performance are usually two conflicting goals in the context of developing parallel code for scientific computations. *Partitioned global address space* (PGAS) [1–4], however, is a parallel programming model that aims to achieve both goals at the same time. The fundamental mechanism of PGAS is a global address space that is conceptually shared among concurrent processes that jointly execute a parallel program. Data exchange between the processes is carried out by a low-level network layer "under the hood" without explicit involvement from the programmer, thus providing good productivity. The shared global address space is logically partitioned such that each partition has affinity to a designated owner process. This awareness of data locality is essential for achieving good performance of parallel programs written in the PGAS model because the globally shared address space may actually encompass many physically distributed memory subsystems.

Unified Parallel C (UPC) [5, 6] is an extension of the C language and provides the PGAS parallel programming model. The concurrent execution processes of UPC are termed as *threads*, which execute a UPC program in the style of single-program-multiple-data. The data variables of each thread are of two types: *private* and *shared*. Variables of the

second type, accessible by all the threads, are found in the globally shared address space. In particular, shared data arrays of UPC provide programmer friendliness because any thread can use a global index to access an arbitrary array element. If the accessing thread does not own the target element, between-thread communication will be carried out automatically.

Another parallelization-simplifying feature of UPC is that shared arrays allow a straightforward distribution of data ownership among the threads. However, the simple data distribution scheme adopted for shared arrays may bring disadvantages. First, balancing the computational work among the threads can be more challenging than other parallel programming models that allow an uneven (static or dynamic) distribution of array elements to account for the possibly inhomogeneous cost per element. Second, for a UPC program where between-thread memory operations are inevitable, which is true for most scientific applications, the only mechanism for a programmer to indirectly control the impact of remote-memory traffic is to tune the block size constant of shared arrays. Third, all nonprivate memory operations (i.e., between threads) are considered in UPC to be of one type. There is no way for UPC to distinguish intra-compute-node memory operations (between threads running on the same hardware node) from their internode counterparts. The latter require explicitly transferring data over some interconnect between the nodes, which is considerably more costly.

In this paper, we will closely investigate the second and third disadvantages mentioned above. This will be done in the context of fine-grained remote-memory operations that have irregular thread-to-thread communication patterns. They can arise from irregular and indirectly indexed accesses to the elements of shared arrays in UPC. Our objectives are two-fold. First, we want to study the impact of several performance-enhancing techniques in UPC programming: privatizing for-loop iterations among threads, explicitly casting pointers-to-shared to pointers-to-local, adopting bulk memory transfers instead of individual remote-memory accesses, and message condensing with consolidation. Second, and more importantly, we will propose performance models for a representative example of scientific computations that induce fine-grained and irregular UPC remote-memory operations. Based on a simple philosophy of quantifying the occurrences and volumes of two categories of interthread memory traffic: local interthread traffic (within a compute node) and remote interthread traffic (between nodes), these performance models only need a small number of hardware characteristic parameters to provide a realistic performance prediction. This helps to understand and further tune the obtainable performance on an existing hardware system, while giving insightful predictions of the achievable scalability on upcoming new platforms.

We will focus on a specific category of matrix-vector multiplication operations where the involved matrix is sparse and has a constant number of nonzero values per row. Such a computational kernel appears in many branches of computational science. A performance challenge is that the nonzero values per matrix row can spread irregularly with respect to the columns. Consequently, any computer implementation will involve irregular and indirectly indexed accesses to numerical arrays. The resulting fine-grained irregular data accesses need to be handled with care, particularly in parallel implementations. We thus choose this computational kernel as a concrete case of fine-grained irregular communication that can occur in the UPC code. We want to demonstrate that proper code transformations can be applied to a naive, programmer-friendly but inefficient UPC implementation, for obtaining considerable enhancements of the computing speed. Moreover, the obtained performance enhancements can be backed up by conceptually simple performance models.

The remainder of this paper is organized as follows. Section 2 explains the basic mechanism of shared arrays, the cornerstone of UPC programming. Then, Section 3 gives a numerical description of our target computational problem and a naive UPC implementation. Thereafter, Section 4 shows in detail three programming strategies that transform the naive implementation for increasingly better performance. In Section 5, three performance models are developed to match with the three code transformations. Section 6 presents an extensive set of numerical experiments and time measurements both for showing the impact of the code transformations and verifying the performance models developed. Relevant related work is reviewed in Section 7, whereas Section 8 shows how our performance modeling methodology can easily be extended to simpler 2D calculations on a uniform mesh, before Section 9 concludes the entire paper with additional comments.

2. Shared Arrays of UPC

Shared data arrays, whose elements are freely accessible by all the threads, typically constitute the main data structure of a UPC program. They thus deserve a separate introduction in this section. The most common scenario is that the elements of a shared array have an evenly distributed thread affinity. This gives a straightforward approach to data partitioning while providing a user-controllable mechanism of data locality. The standard upc_all_alloc function (see e.g. [6]), to be used by all the UPC implementations in this paper for array allocation, has the following syntax:

shared void *upc_all_alloc (size_t nblks, size_t nbytes).

Note that shared is a specific type qualifier. Also, upc_all_alloc needs to be *collectively* called by all threads to allocate a shared array. Upon return, a private pointer to the allocated shared array, or a private *pointer-to-shared* in the more rigorous UPC terminology, becomes available on each thread . The allocated shared array consists of nblks blocks in total, whose affinity is distributed evenly among the threads in a *cyclic* order. The value of nbytes is the number of bytes occupied per block , which translates the block size , i.e., number of elements per block, as nbytes/sizeof(one element). The blocks that have affinity to the same owner thread are physically allocated *contiguously* in the owner thread's local memory.

This data ownership distribution scheme, which in many cases determines an associated work partitioning, has the advantage of a simple mapping between a global element index and the owner thread ID. It can be described as follows:

$$\text{owner\_thread\_id} = \left\lfloor \frac{\text{global\_index}}{\text{block\_size}} \right\rfloor \text{modulo THREADS}, \tag{1}$$

where THREADS is a built-in variable of UPC that stores the total number of threads participating in a parallel execution. The value of THREADS is fixed at either compile time or run time.

Accessing the elements of a shared array by their global indices, although programmer-friendly, can potentially incur considerable overhead. This is because a pointer-to-shared has three fields: the owner thread ID, the phase (i.e., the element offset within the affinity block), and the corresponding local memory address [5]. The standard upc_threadof(shared void *ptr) function of UPC returns the owner thread ID of an element that is pointed to by the pointer-to-shared ptr. Every access through a pointer-to-shared requires updating the three fields and thus always incurs overhead. Moreover, if the accessing thread is different from the owner thread, a behind-the-scene data transfer between the two threads has to be carried out. For indirectly indexed accesses of the elements in a shared array, a compiler cannot batch the individual between-thread data exchanges for the purpose of message aggregation (such as described in [7, 8]). The individual between-thread data exchanges are thus particularly costly when the accessing thread runs on a different compute node than the owner thread.

3. SpMV and a Naive UPC Implementation

This section is devoted to explaining the target computational kernel of this paper and presenting a naive UPC implementation.

3.1. Definition of Sparse Matrix-Vector Multiplication. Mathematically, a general matrix-vector multiplication is compactly denoted by $y = Mx$. Without loss of generality, we assume that the matrix M is square, having n rows and n columns. The input vector x and the result vector y are both of length n. Then, the general formula for computing element number i of the result vector y is as follows (using zero-based indices):

$$y(i) = \sum_{0 \le j < n} M(i, j)x(j). \tag{2}$$

If most of the $M(i, j)$ values are zero, M is called a *sparse* matrix. In this case, the above formula becomes unnecessarily expensive from a computational point of view. A more economic formula for computing $y(i)$ in a sparse matrix-vector multiplication (SpMV) is thus

$$y(i) = \sum_{M(i,j) \ne 0} M(i, j)x(j), \tag{3}$$

which only involves the nonzero values of matrix M on each row. Moreover, it is memory-wise unnecessarily expensive to store all the n^2 values of a sparse matrix, because only the nonzero values are used. This prompts the adoption of various compact storage formats for sparse matrices, such as the coordinate format (COO), compressed sparse row format (CSR), compressed sparse column format (CSC), and the EllPack format [9].

In particular, for sparse matrices that have a fixed number of nonzero values per row, it is customary to use the EllPack storage format, which conceptually uses two 2D tables. Both tables are of the same size, having n rows and the number of columns equaling the fixed number of nonzeros per row. The first table stores all the nonzero values of the sparse matrix, whereas the second table stores the corresponding column indices of the nonzeros. Moreover, if we assume that all the values on the main diagonal of a sparse matrix M are nonzero, which is true for most scientific applications, it is beneficial to split M as

$$M = D + A, \tag{4}$$

where D is the main diagonal of M and A contains the off-diagonal part of M. Then, a modified EllPack storage format can employ a 1D array of length n to store the entire main diagonal D. There is no need to store the column indices of these nonzero diagonal values, because their column indices equal the row indices by definition. Suppose r_{nz} now denotes the fixed number of nonzero off-diagonal values per row. For storing the nonzero values in the off-diagonal part A, it is customary to use two 1D arrays both of length $n \cdot r_{\text{nz}}$ (instead of two $n \times r_{\text{nz}}$ 2D tables), one stores the nonzero off-diagonal values consecutively row by row, whereas the other stores the corresponding integer column indices.

Following such a modified EllPack storage format, a straightforward sequential C implementation of SpMV is shown in Listing 1, where the integer array J contains the column indices of all nonzero off-diagonal values.

The sparsity pattern of the M matrix, i.e., where its nonzeros are located, is described by the array J of column indices. The actual pattern is matrix-dependent and irregular in general, meaning that each $x(i)$ value is irregularly used multiple times in computing several values in the result vector y. This has an important bearing on the achievable performance of a typical computer implementation, because the actual sparsity pattern of the M matrix may affect the level of data reuse in the different caches of a computer's memory system. Additionally, for the case of parallel computing, some of the values in the x vector have to be shared between processes (or threads). The irregular data reuse in the x vector will thus imply an irregular data sharing pattern. The resulting communication overhead is determined by the number of process pairs that need to share some values of the x vector, as well as the amount of shared data between each pair. The impact of these issues on different UPC implementations of SpMV will be the main subject of study in this paper. In the following, we first present a naive UPC implementation, whereas code transformation strategies that aim to improve the performance will be discussed in Section 4.

```
for (int i = 0; i < n; i++) {
  double tmp = 0.0;
  for (int j = 0; j < r_nz; j++)
    tmp += A[i * r_nz + j] * x[J[i * r_nz + j]];
  y[i] = D[i] * x[i] + tmp;
}
```

Listing 1: A straightforward sequential implementation of SpMV using a modified EllPack storage format.

3.2. A Naive UPC Implementation. The user-friendliness of UPC allows for an equally compact and almost identical implementation of the SpMV computational kernel (starting from the line of upc_forall in Listing 2) as the straightforward C implementation in Listing 1. An immediate advantage is that parallelization is automatically enabled through using the upc_forall construct of UPC, which deterministically divides the iterations of a for-loop among the threads. The five involved data arrays, which are all allocated by upc_all_alloc as shared arrays, are evenly distributed among the UPC threads in a block-cyclic order. More specifically, the arrays x, y, and D adopt a programmer-prescribed integer value, BLOCKSIZE, as their block size associated with the affinity distribution. The arrays A and J, both of length $n \cdot r_{nz}$, use $r_{nz} *$ BLOCKSIZE as their block size. This gives a consistent thread-wise data distribution for the five shared data arrays.

The UPC implementation of SpMV shown in Listing 2 is clean and easy to code. The parallelization details, i.e., data distribution and work partitioning, are an inherent part of the language definition of UPC. Since the number of nonzeros per matrix row is assumed to be fixed, the adopted thread-wise data distribution for the shared D, A, J, and y arrays is perfect, in that each thread will only access its owned blocks of these arrays. For the shared array x, whose values are indirectly accessed via the column index array J, underlying data transfers between the threads are inevitable in general.

As will be detailed later, the irregular column positions of the nonzero values (stored in the array J) will cause fine-grained and irregular between-thread data exchanges associated with the shared array x in the straightforward UPC implementation shown in Listing 2. Tuning the value of BLOCKSIZE can change the pattern and volume of between-thread communication. However, to ensure good performance, proper code transformations of such a naive UPC implementation are necessary.

4. Strategies of Performance Enhancement

This section studies three programming strategies that can be applied to transforming the naive UPC implementation. The main purpose is to reduce the impact of implicit between-thread data exchanges that are caused by irregular and indirectly indexed accesses to the shared array x. At the same time, we also want to eliminate some of the other types of avoidable overhead associated with UPC programming.

4.1. Explicit Thread Privatization. Some of the programmer-friendly features of UPC are accompanied with performance penalties. Relating to the naive UPC implementation of SpMV in Listing 2, these concern automatically dividing the iterations of a for-loop among threads by upc_forall and allowing any thread to access any element of a shared array. See Section 2 about the latter.

The upc_forall construct of UPC is a collective operation. In the example of upc_forall ($i = 0$; $i < n$; i++; &$y[i]$) used in Listing 2, all the threads go through the *entire* for-loop and check the affinity of each iteration, by comparing whether upc_threadof(&$y[i]$) equals the built-in MYTHREAD value that is unique per thread. Although only iterations having an affinity equaling MYTHREAD are executed by a thread, it is not difficult to see the overhead due to excessive looping and calls to the standard upc_threadof function behind the scene.

To get rid of the unnecessary overhead associated with upc_forall, we can let each thread work directly with the loop iterations that have the matching affinity. Note that the affinity distribution of the i-indexed loop iterations can be easily determined using the value of BLOCKSIZE. Such an explicit thread privatization of the loop iterations also opens up another opportunity for performance enhancement. Namely, all the globally indexed accesses to the shared arrays y, A, J, and D (except x) can be replaced by more efficient accesses through private pointers and local indices. This is achievable by using the well-known technique of casting pointers-to-shared to pointers-to-local [10]. Following these two steps that can be loosely characterized as *explicit thread privatization*, the naive UPC implementation can be transformed as shown in Listing 3.

It can be observed in Listing 3 that each thread now only traverses its designated rows of the sparse matrix. The computational work per thread is executed by going through its owned blocks in the shared arrays y, D, A, and J, for which each thread is guaranteed to never touch blocks owned by the other threads. Pointers to these four shared arrays are cast to their local counterparts loc_y, loc_D, loc_A, and loc_J, since array accesses through private pointers and local indices are the most efficient. On the other hand, casting pointer-to-shared x cannot be done, because the indirect accesses of form x[loc_J[$k * r_{nz} + j$]] may lead to situations where the accessing thread is different from the owner thread. Compared with the naive UPC implementation in Listing 2, the transformed version after explicit thread privatization will have a much better performance. However, further performance improvement can be obtained by also "privatizing" the global accesses to the shared array x. This can be achieved by two code transformations, of different programming complexities and performance gains, which are to be detailed below.

4.2. Block-Wise Data Transfer between Threads. Although Listing 3 improves the naive UPC implementation by the approaches of explicit thread privatization, each thread still indirectly accesses the elements of the shared array x through global indices that are stored in the array J (now cast to pointer-to-local loc_J per block). When an indirectly

```
/* Total number of blocks in every shared array */
int nblks = n/BLOCKSIZE + (n% BLOCKSIZE) ?1 : 0;
/* Allocation of five shared arrays */
shared [BLOCKSIZE] double *x = upc_all_alloc (nblks, BLOCKSIZE * sizeof(double));
shared [BLOCKSIZE] double *y = upc_all_alloc (nblks, BLOCKSIZE * sizeof(double));
shared [BLOCKSIZE] double *D = upc_all_alloc (nblks, BLOCKSIZE * sizeof(double));
shared [r_nz * BLOCKSIZE] double *A = upc_all_alloc (nblks, r_nz * BLOCKSIZE * sizeof(double));
shared [r_nz * BLOCKSIZE] int *J = upc_all_alloc (nblks, r_nz * BLOCKSIZE * sizeof(int));
// ...
/* Computation of SpMV involving all threads */
upc_forall (int i = 0; i < n; i++; &y[i]) {
  double tmp = 0.0;
  for (int j = 0; j < r_nz; j++)
    tmp += A[i * r_nz + j] * x[J[i * r_nz + j]];
  y[i] = D[i] * x[i] + tmp;
}
```

Listing 2: A naive UPC implementation of SpMV using a modified EllPack storage format.

```
/* Allocation of the five shared arrays x, y, D, A, J as in the naive implementation */
// ...
/* Instead of upc_forall, each thread directly handles its designated blocks */
int mythread_nblks = nblks/THREADS + (MYTHREAD < (nblks% THREADS) ?1 : 0);
for (int mb = 0; mb < mythread_nblks; mb++) {
  int offset = (mb * THREADS + MYTHREAD) * BLOCKSIZE;
  /* casting shared pointers to local pointers */
  double *loc_y = (double*) (y + offset);
  double *loc_D = (double*) (D + offset);
  double *loc_A = (double*) (A + offset * r_nz);
  int *loc_J = (int*) (J + offset * r_nz);
  /* computation per block */
  for (int k = 0; k < min(BLOCKSIZE, n-offset); k++) {
    double tmp = 0.0;
    for (int j = 0; j < r_nz; j++)
      tmp += loc_A[k * r_nz + j] * x[loc_J[k * r_nz + j]];
    loc_y[k] = loc_D[k] * x[offset + k] + tmp;
  }
}
```

Listing 3: An improved UPC implementation of SpMV by explicit thread privatization.

indexed $x[\text{loc\_}J[k * r_{nz} + j]]$ value has affinity with MYTHREAD, the overhead only concerns updating the three fields of a pointer-to-shared. If MYTHREAD is different from the owner thread, however, a behind-the-scene data transfer will be executed in addition. Moreover, these between-thread data transfers will happen one by one, because a typical compiler is unable to batch the individual transfers. The extraoverhead is particularly high if the owner and accessing threads reside on different compute nodes. To avoid the potentially high overhead associated with $x[\text{loc\_}J[k * r_{nz} + j]]$, we can create a private copy of x on each thread and transfer the needed blocks from x to the private copy before carrying out the SpMV. The resulting UPC implementation is shown in Listing 4.

In Listing 4, we have used the one-sided communication function upc_memget of UPC to transfer all the needed blocks, one by one, from the shared array x into a thread-private local copy named mythread_x_copy. The syntax of upc_memget is as follows:

> void upc_memget(void *dst, shared const void *src, size_t n).

We have thus completely avoided accessing values of the shared array x. However, there are a few "prices" paid on the way. First, each thread needs to allocate its own mythread_x_copy array of length n. This obviously increases the total memory usage. Second, the actual computation of SpMV on each thread needs to be preceded by transporting all the needed blocks from the shared array x into the corresponding places of mythread_x_copy. Specifically, a needed block from x is defined as having at least one $x[\text{loc\_}J[k * r_{nz} + j]]$ value that will participate in calculating

```
/* Allocation of the five shared arrays x, y, D, A, J as in the naive implementation */
// ...
/* Allocation of an additional private x array per thread */
double *mythread_x_copy = (double*) malloc(n * sizeof(double));
/* Prep-work: check for each block of x whether it has values needed by MYTHREAD; make a private boolean array
"block_is_needed" of length nblks */
// ...
/* Transport the needed blocks of x into mythread_x_copy */
for (int b = 0; b < nblks; b++)
  if (block_is_needed[b])
    upc_memget(&mythread_x_copy[b * BLOCKSIZE], &x[b * BLOCKSIZE], min(BLOCKSIZE, n − b * BLOCKSIZE) *
sizeof(double));
/* SpMV: each thread only goes through its designated blocks */
int mythread_nblks = nblks/THREADS + (MYTHREAD < (nblks% THREADS) ?1 : 0);
for (int mb = 0; mb < mythread_nblks; mb++) {
  int offset = (mb * THREADS + MYTHREAD) * BLOCKSIZE;
  /* casting shared pointers to local pointers */
  double *loc_y = (double*) (y + offset);
  double *loc_D = (double*) (D + offset);
  double *loc_A = (double*) (A + offset * r_nz);
  int *loc_J = (int*) (J + offset * r_nz);
  /* computation per block */
  for (int k = 0; k < min(BLOCKSIZE, n-offset); k++) {
    double tmp = 0.0;
    for (int j = 0; j < r_nz; j++)
      tmp += loc_A[k * r_nz + j] * mythread_x_copy[loc_J[k * r_nz + j]];
    loc_y[k] = loc_D[k] * mythread_x_copy[offset + k] + tmp;
  }
}
```

Listing 4: An improved UPC implementation of SpMV by block-wise communication.

the designated elements in y on each thread (with MYTHREAD as its unique ID). We note that each needed block is transported in its *entirety*, independent of the actual number of x values needed in that block. This also applies to the blocks of x that are owned by MYTHREAD. The whole procedure of transporting the needed blocks of x, implemented as the for-loop indexed by b in Listing 4, will result in time usage overhead. Nevertheless, this additional time usage is often compensated by avoiding the individual accesses to the shared array x. Third, to identify whether a block of x is needed by MYTHREAD requires prescreening the designated blocks of the array J (not shown in Listing 4). This is typically considered a negligible "one-time" cost if the same sparse matrix, or the same sparsity pattern shared among several sparse matrices, is repeatedly used in many SpMV operations later.

4.3. Message Condensing and Consolidation. One shortcoming of the transformed UPC code shown in Listing 4 is that each needed block from x is transported in its entirety. This will lead to unreasonably large messages, when only a small number of values in a block of x is needed by MYTHREAD. Also, several messages may be transported (instead of one consolidated message) between a pair of threads, where each message has a rigid length of BLOCKSIZE. To condense and consolidate the messages, we can carry out a different code transformation as follows.

4.3.1. Preparation Step. Each thread checks, in a "one-time" preparation step, which of its owned x values will be needed by the other threads. We also ensure that only one message is exchanged between each pair of communicating threads. The length of a message from thread T_1 to thread T_2 equals the number of *unique* values in the x blocks owned by T_1 that are needed by T_2. All the between-thread messages are thus condensed and consolidated. After this preparation step, the following private arrays are created on each thread:

```
int *mythread_num_send_values, *mythread_num_recv_values;
int **mythread_send_value_list, **mythread_recv_value_list;
double **mythread_send_buffers;
```

All the above private arrays have length THREADS (in the leading direction). If mythread_num_send_values[T] >0, it means that MYTHREAD needs to pack an outgoing message of this length for thread T as the receiver. Correspondingly, mythread_send_value_list[T] points to a list of local indices relative to a pointer-to-local, which is cast from &x[MYTHREAD * BLOCKSIZE], so that the respective needed x values can be efficiently extracted and packed together as the outgoing message mythread_send_buffers[T] toward thread T. The one-sided communication command upc_memput, which is of the following syntax:

void upc_memput(shared void *dst, const void *src, size_t n)

will be used to transfer each outgoing message.

The meaning of mythread_num_recv_values[T] applies to the opposite communication direction. Also, the content of mythread_recv_value_list[T] will be needed by MYTHREAD to unpack the incoming message from thread T. One particular issue is that the upc_memput function requires a pointer-to-shared available on the destination thread. To this end, we need the following shared array with a block size of THREAD, where each array element is itself a pointer-to-shared:

```
shared[] double* shared [THREADS] shared_recv_buffers
[THREADS * THREADS];
```

An important task in the preparation step is to let each thread go through the following for-loop to allocate the individual buffers, in UPC's globally shared address space, for its expected incoming messages:

```
for (int T = 0; T < THREADS; T++)
  if (int length = mythread_num_recv_values[T] > 0)
    shared_recv_buffers[MYTHREAD * THREADS + T]
     = (shared[] double*)upc_alloc(length * sizeof
     (double));
```

It should be noted that the standard upc_alloc function should be called by only one thread. The entire array that is allocated by upc_alloc has affinity to the calling thread while being accessible by all the other threads [6]. In the above for-loop, each thread (using its unique MYTHREAD value) only calls upc_alloc inside its affinity block of shared_recv_buffers.

4.3.2. Communication Procedure. When the preparation step described above is done, we need to invoke a communication procedure to precede each SpMV computation. The communication procedure first lets each thread (with MYTHREAD as its unique ID) pack an outgoing message for every thread T that has mythread_num_send_values [T] > 0, by extracting the respective needed values from its owned blocks of the shared array x (cast to a pointer-to-local), using the local indices stored in mythread_send_value_list [T]. Then, the one-sided communication function upc_memput is called to send every ready-packed outgoing message to its destination thread. Thereafter, the upc_barrier command is posted to ensure that all the interthread communication is done, which means that all the expected messages have arrived on the respective destination threads. Finally, each thread unpacks every incoming message by copying its content to the respective positions in the thread-private array mythread_x_copy. Each thread also copies its owned blocks from the shared array x to the corresponding positions in the thread-privatemythread_x_copy. The entire communication procedure can be seen in Listing 5.

4.3.3. Implementation. By incorporating the above preparation step and communication procedure, we can create a new UPC implementation of SpMV in Listing 5. Specifically, each pair of communicating threads exchanges only one message containing the actually needed x values. As a "price," the new version has to introduce additional data structures in the preparation step and involve message packing and unpacking in the communication procedure.

5. Performance Models

We consider the performance model of a parallel implementation as a formula that can theoretically estimate the run time, based on some information of the target work and some characteristic parameters of the hardware platform intended. Roughly, the time usage of a parallel program that implements a computation comprises the time spent on the computational work and the parallelization overhead. The latter is mostly spent on various forms of communication between the executing processes or threads.

The three UPC implementations shown in Section 4 carry out identical computational work. However, they differ greatly in how the between-thread communication is realized, with respect to both the frequency and volume of the beween-thread data transfers. As will be demonstrated in Section 6, the time usages of the three transformed UPC implementations are very different. This motivates us to derive the corresponding performance models, with a special focus on modeling the communication cost in detail. Such theoretical performance models will help us to understand the actual computing speed achieved, while also providing hints on further performance tuning.

5.1. Time Spent on Computation. Due to a fixed number of nonzeros per matrix row, the amount of floating-point operations per thread is linearly proportional to the number of $y(i)$ values that are designated to each thread to compute. For all the UPC implementations in this paper, the shared array y is distributed in a block-cyclic manner, with a programmer-prescribed block size of BLOCKSIZE. Recall that the array y is of length n; thus, the number of y blocks assigned per thread, $B^{\text{comp}}_{\text{thread}}$, is given by the following formula:

$$B^{\text{comp}}_{\text{total}} = \left\lceil \frac{n}{\text{BLOCKSIZE}} \right\rceil,$$

$$B^{\text{comp}}_{\text{thread}} = \left\lfloor \frac{B^{\text{comp}}_{\text{total}}}{\text{THREADS}} \right\rfloor + \begin{cases} 1, & \text{if MYTHREAD} < \left(B^{\text{comp}}_{\text{total}} \text{ modulo THREADS}\right), \\ 0, & \text{else.} \end{cases} \tag{5}$$

Due to a low ratio between the number of floating-point operations and the induced amount of data movement in the memory hierarchy, the cost of computation for our SpMV example is determined by the latter, as suggested by the well-known Roofline model [11]. Our strategy is to derive the minimum amount of data movement needed between the main memory and the last-level cache. More specifically, the

```
/* Allocation of the five shared arrays x, y, D, A, J as in the naive implementation */
// ...
/* Allocation of an additional private x array per thread */
double *mythread_x_copy = (double*) malloc(n * sizeof(double));
/* Preparation step: create and fill the thread-private arrays of int *mythread_num_send_values, int *mythread_num_recv_values,
int **mythread_send_value_list, int **mythread_recv_value_list, double **mythread_send_buffers. Also, shared_recv_buffers is
prepared. */
// ...
/* Communication procedure starts */
int T, k, mb, offset;
double *local_x_ptr = (double*)(x + MYTHREAD * BLOCKSIZE);
for (T = 0; T < THREADS; T++)
  if (mythread_num_send_values[T] > 0) /* pack outgoing messages */
    for (k = 0; k < mythread_num_send_values[T]; k++)
      mythread_send_buffers[T][k] = local_x_ptr[mythread_send_value_list[T][k]];
for (T = 0; T < THREADS; T++)
  if (mythread_num_send_values[T] > 0) /* send out messages */
    upc_memput(shared_recv_buffers[T * THREADS + MYTHREAD], mythread_send_buffers[T], mythread_num_send_values
    [T] * sizeof(double));
upc_barrier;
int mythread_nblks = nblks/THREADS + (MYTHREAD < (nblks% THREADS) ?1 : 0);
for (mb = 0; mb < mythread_nblks; mb++) /* copy own x-blocks */
  offset = (mb * THREADS + MYTHREAD) * BLOCKSIZE;
  memcpy(&mythread_x_copy[offset], (double*)(x + offset), min(BLOCKSIZE, n-offset) * sizeof(double));
}
for (T = 0; T < THREADS; T++)
  if (mythread_num_recv_values[T] > 0) {/* unpack incoming messages */
    double *local_buffer_ptr = (double*) shared_recv_buffers[MYTHREAD * THREADS + T];
    for (k = 0; k < mythread_num_recv_values[T]; k++)
      mythread_x_copy[mythread_recv_value_list[T][k]] = local_buffer_ptr[k];
  }
/* Communication procedure ends */
/* SpMV: each thread only goes through its designated blocks */
for (mb = 0; mb < mythread_nblks; mb++) {
  offset = (mb * THREADS + MYTHREAD) * BLOCKSIZE;
  /* casting shared pointers to local pointers */
  double *loc_y = (double*) (y + offset);
  double *loc_D = (double*) (D + offset);
  double *loc_A = (double*) (A + offset * r_nz);
  int *loc_J = (int*) (J + offset * r_nz);
  /* computation per block */
  for (k = 0; k < min(BLOCKSIZE, n-offset); k++) {
    double tmp = 0.0;
    for (int j = 0; j < r_nz; j++)
      tmp += loc_A[k * r_nz + j] * mythread_x_copy[loc_J[k * r_nz + j]];
    loc_y[k] = loc_D[k] * mythread_x_copy[offset + k] + tmp;
  }
}
```

Listing 5: An improved UPC implementation of SpMV by message condensing and consolidation.

following formula gives the minimum data traffic (in bytes) from/to the main memory for computing each $y(i)$ value:

$$D_{\min}^{\text{comp}} = r_{\text{nz}} \cdot (\text{sizeof(double)} + \text{sizeof(int)}) + 3 \cdot \text{sizeof(double)}, \quad (6)$$

where r_{nz} denotes the fixed number of off-diagonal nonzero values per matrix row, each occupying sizeof(double) bytes in memory, with sizeof(int) bytes needed per column index. The last term in (6) corresponds to the two memory loads for accessing loc_$D[k]$ and mythread_x_copy[offset + k] (or x[offset + k]) and the memory store associated with updating loc_$y[k]$. We refer to Listings 3–5 for the implementation details.

Formula (6) has assumed perfect data reuse in the last-level data cache. Our earlier experiences with the same SpMV computation (implemented in sequential C or OpenMP), for the case of a "proper" ordering of the matrix rows ([12]), suggest that (6) is a realistic estimate for the last two UPC implementations (Listings 4 and 5). For these two

implementations, the x values are fetched from the thread-private array mythread_x_copy. In the first transformed UPC implementation (Listing 3), indirectly indexed accesses to the shared array x (of form $x[\text{loc\_}J[k * r_{\text{nz}} + j]]$) will incur additional memory traffic on "remote" threads, caused by the inevitable between-thread data transfers. We have chosen for this case to model the deviation from (6) as a part of the communication cost, to be discussed in Section 5.2.3.

Therefore, the minimum computational time needed per thread, which is the same for all the UPC implementations of this paper, can be estimated as

$$T_{\text{thread}}^{\text{comp}} = \frac{B_{\text{thread}}^{\text{comp}} \cdot \text{BLOCKSIZE} \cdot D_{\text{min}}^{\text{comp}}}{W_{\text{thread}}^{\text{private}}}, \tag{7}$$

where $W_{\text{thread}}^{\text{private}}$ denotes the realistic bandwidth (bytes per second) at which a thread can access its private memory space. This can be found by running a multithreaded STREAM benchmark [13] on one compute node of a target hardware platform, using the intended number of UPC threads per node. The $W_{\text{thread}}^{\text{private}}$ value equals the measured multithreaded STREAM bandwidth divided by the number of threads used. Note that the bandwidth measured by a single-threaded STREAM benchmark cannot be used directly as $W_{\text{thread}}^{\text{private}}$, unless a single UPC thread per compute node is indeed intended. This is because the multithreaded STREAM bandwidth is not linearly proportional to the number of threads used, due to saturation of the memory bandwidth.

5.2. Communication Overhead

5.2.1. Definitions. Before we dive into the details of modeling the various communication costs that are associated with the three transformed UPC implementations, it is important to establish the following definitions:

(i) If a thread accesses a memory location in the globally shared address space with affinity to another thread, a *non-private* memory operation is incurred.

(ii) A nonprivate memory operation, which is between two threads, can belong to one of two categories: *local inter-thread* and *remote inter-thread*. The first category refers to the case where the two involved threads reside on the same compute node, which has a physically shared NUMA (or UMA) memory encompassing all the threads running on the node. The second category refers to the case where the two threads reside on two different nodes, which need to use some interconnect for exchanging data.

(iii) A nonprivate memory operation, in each category, can happen in two modes: either individually or inside a sequence of memory operations accessing a contiguous segment of nonprivate memory. We term the first mode as *individual* and the second mode as *contiguous*.

5.2.2. Cost of Nonprivate Memory Operations. The time needed by one nonprivate memory operation, in the contiguous mode, can be estimated as

$$T_{\text{cntg}}^{\text{local}} = \frac{\text{sizeof (one element)}}{W_{\text{thread}}^{\text{local}}},$$
$$T_{\text{cntg}}^{\text{remote}} = \frac{\text{sizeof (one element)}}{W_{\text{node}}^{\text{remote}}}, \tag{8}$$

where $W_{\text{thread}}^{\text{local}}$ denotes the per-thread bandwidth for contiguous local interthread memory operations, and we assume for simplicity $W_{\text{thread}}^{\text{local}} = W_{\text{thread}}^{\text{private}}$, with the latter being defined in Section 5.1. Correspondingly, $W_{\text{node}}^{\text{remote}}$ denotes the interconnect bandwidth available to a node for contiguous remote (internode) memory operations. The reason for adopting a per-node bandwidth for internode memory operations is because the internode network bandwidth can typically be fully utilized by one thread, unlike the main-memory bandwidth. The value of $W_{\text{node}}^{\text{remote}}$ can be measured by a modified UPC STREAM benchmark or simply a standard MPI ping-pong test, to be discussed in Section 6.2.

The cost of one individual remote interthread memory operation, $T_{\text{indv}}^{\text{remote}}$, is assumed to be dominated by a constant latency overhead, denoted by τ. Specifically, the latency τ is independent of the actual number of bytes involved in one individual remote-memory operation. By the same reason, $W_{\text{node}}^{\text{remote}}$ has no bearing on $T_{\text{indv}}^{\text{remote}}$. The actual value of τ can be measured by a special UPC microbenchmark, to be discussed in Section 6.2. The cost of one individual local interthread memory operation can be estimated by the following formula:

$$T_{\text{indv}}^{\text{local}} = \frac{\text{sizeof (cache line)}}{W_{\text{thread}}^{\text{local}}}. \tag{9}$$

Here, we will again adopt $W_{\text{thread}}^{\text{local}} = W_{\text{thread}}^{\text{private}}$. The reason for having the size of one cache line as the numerator in (9) is that individual local interthread memory operations are considered to be noncontiguously spread in the private memory of the owner thread, thus paying the price of an entire cache line per access (it has been implied that one data element occupies fewer bytes than one cache line).

5.2.3. Communication Time for the First Transformed UPC Implementation. For the UPC implementation in Listing 3, an individual nonprivate memory operation arises when the owner thread of value $x[\text{loc\_}J[k * r_{\text{nz}} + j]]$ is different from the accessing thread. Each such nonprivate memory operation costs either $T_{\text{indv}}^{\text{local}}$ as defined in (9) or $T_{\text{indv}}^{\text{remote}} = \tau$. To quantify the total communication time incurred per thread, we need the following two counts, which can be obtained by letting each thread examine its owned blocks of the shared array J:

(i) $C_{\text{thread}}^{\text{local,indv}}$: number of occurrences when $\&x[\text{loc\_}J[k * r_{\text{nz}} + j]]$ has a different affinity than MYTHREAD and the owner thread resides on the same compute node as MYTHREAD

(ii) $C_{\text{thread}}^{\text{remote,indv}}$: number of occurrences when &x[loc_J[$k * r_{\text{nz}} + j$]] has a different affinity than MYTHREAD and the owner thread resides on a different compute node

Thus, the total communication cost per thread during each SpMV is

$$T_{\text{thread}}^{\text{comm,UPCv1}} = C_{\text{thread}}^{\text{local,indv}} \cdot \frac{\text{sizeof (cache line)}}{W_{\text{thread}}^{\text{private}}} + C_{\text{thread}}^{\text{remote,indv}} \cdot \tau. \tag{10}$$

5.2.4. Communication Time for the Second Transformed UPC Implementation. For the UPC implementation in Listing 4, before computing the SpMV, each thread calls the upc_memget function to transport its needed blocks from the shared array x to the private array mythread_x_copy. To estimate the communication time spent per node, we will use the following formula:

$$T_{\text{node}}^{\text{comm,UPCv2}} = \max_{\forall \text{threads in node}} B_{\text{thread}}^{\text{local}} \cdot \frac{2 \cdot \text{BLOCKSIZE} \cdot \text{sizeof (double)}}{W_{\text{thread}}^{\text{private}}} + \sum_{\forall \text{threads in node}} B_{\text{thread}}^{\text{remote}} \cdot \left(\tau + \frac{\text{BLOCKSIZE} \cdot \text{sizeof (double)}}{W_{\text{node}}^{\text{remote}}} \right), \tag{11}$$

where $B_{\text{thread}}^{\text{local}}$ denotes the number of x blocks residing on the same node as MYTHREAD and having at least one value needed by MYTHREAD, whereas $B_{\text{thread}}^{\text{remote}}$ denotes the number of needed blocks residing on other nodes. The reason for having a factor of 2 in the numerator of the first term on the right-hand side of (11) is due to the private/local memory loads and stores that both take place on the same node. Note that we have consistently assumed $W_{\text{thread}}^{\text{local}} = W_{\text{thread}}^{\text{private}}$. More importantly, we consider that all the threads on the same node concurrently carry out their intranode part of communication, whereas the internode operations of upc_memput are carried out one by one. For communicating each internode block, we have included τ as the "start-up" overhead in addition to the $W_{\text{node}}^{\text{remote}}$-determined cost.

5.2.5. Communication Time for the Third Transformed UPC Implementation. For the UPC implementation in Listing 5, the overhead per thread for preparing the private array mythread_x_copy before the SpMV has four parts: (1) packing all its outgoing messages, (2) calling upc_memput for each outgoing message, (3) copying its own blocks of x to the corresponding positions in mythread_x_copy, and (4) unpacking the incoming messages.

Let us denote by $S_{\text{thread}}^{\text{local,out}}$ the accumulated size of the outgoing messages from MYTHREAD to threads residing on the same node as MYTHREAD; $S_{\text{thread}}^{\text{remote,out}}$ denotes the accumulated size of the outgoing messages towards other nodes. Similarly, $S_{\text{thread}}^{\text{local,in}}$ and $S_{\text{thread}}^{\text{local,in}}$ denote the incoming counterparts. Then, the per-thread overhead of packing the outgoing messages is

$$T_{\text{thread}}^{\text{pack}} = \frac{\left(S_{\text{thread}}^{\text{local,out}} + S_{\text{thread}}^{\text{remote,out}} \right) (2 \cdot \text{sizeof (double)} + \text{sizeof (int)})}{W_{\text{thread}}^{\text{private}}}. \tag{12}$$

We remark that packing each value in an outgoing message requires loading at least sizeof (double)+ sizeof (int) bytes from the private memory and storing sizeof(double) bytes into the message.

Instead of modeling the per-thread overhead related to the upc_memput calls, we choose to model the per-node counterpart as

$$T_{\text{node}}^{\text{memput,UPCv3}} = \max_{\forall \text{threads in node}} \frac{2 \cdot S_{\text{thread}}^{\text{local,out}} \cdot \text{sizeof (double)}}{W_{\text{thread}}^{\text{private}}} + \sum_{\forall \text{threads in node}} \cdot \left(C_{\text{thread}}^{\text{remote,out}} \cdot \tau + \frac{S_{\text{thread}}^{\text{remote,out}} \cdot \text{sizeof (double)}}{W_{\text{node}}^{\text{remote}}} \right), \tag{13}$$

where $C_{\text{thread}}^{\text{remote,out}}$ denotes the number of outgoing internode messages from MYTHREAD. Again, for each internode message, we have included τ as the "start-up" overhead in addition to the $W_{\text{node}}^{\text{remote}}$-determined cost.

The per-thread overhead of copying the private blocks of x into mythread_x_copy is

$$T_{\text{thread}}^{\text{copy}} = \frac{2 \cdot B_{\text{thread}}^{\text{comp}} \cdot \text{BLOCKSIZE} \cdot \text{sizeof (double)}}{W_{\text{thread}}^{\text{private}}}, \tag{14}$$

where we recall that $B_{\text{thread}}^{\text{comp}}$ is defined in (5).

Finally, the per-thread overhead of unpacking the incoming messages is

$$T_{\text{thread}}^{\text{unpack}} = \frac{\left(S_{\text{thread}}^{\text{local,in}} + S_{\text{thread}}^{\text{remote,in}} \right) (\text{sizeof (double)} + \text{sizeof (int)} + \text{sizeof (cache line)})}{W_{\text{thread}}^{\text{private}}}. \tag{15}$$

Note that sizeof(double) + sizeof(double)(int) corresponds to contiguously reading each value from an incoming message, whereas sizeof(cache line) corresponds to the cost of writing the value to a noncontiguous location in the array mythread_x_copy.

5.3. Total Time Usage. Due to the possible imbalance of both computational work and communication overhead among the threads, the total time usage of any of the UPC implementations will be determined by the slowest thread or node. For the first transformed UPC implementation, shown in Listing 3, the total time is determined by the slowest thread:

$$T_{\text{total}}^{\text{UPCv1}} = \max_{\forall \text{threads}} \left(T_{\text{thread}}^{\text{comp}} + T_{\text{thread}}^{\text{comm,UPCv1}} \right). \tag{16}$$

For the second transformed UPC implementation, shown in Listing 4, the total time is determined by the slowest node:

$$T_{\text{total}}^{\text{UPCv2}} = \max_{\forall \text{nodes}} \left(\left(\max_{\forall \text{threads in node}} T_{\text{thread}}^{\text{comp}} \right) + T_{\text{node}}^{\text{comm,UPCv2}} \right). \tag{17}$$

For the third transformed UPC implementation, shown in Listing 5, due to the needed explicit barrier after the upc_memput calls, the total time usage is modeled as

$$T_{\text{total}}^{\text{UPCv3}} = \max_{\forall \text{nodes}} \left(\left(\max_{\forall \text{threads in node}} T_{\text{thread}}^{\text{pack}} \right) + T_{\text{node}}^{\text{memput,UPCv3}} \right) + \max_{\forall \text{threads}} \left(T_{\text{thread}}^{\text{copy}} + T_{\text{thread}}^{\text{unpack}} + T_{\text{thread}}^{\text{comp}} \right). \tag{18}$$

5.4. Remarks. It is important to separate two types of information needed by the above performance models. The hardware-specific information includes $W_{\text{thread}}^{\text{private}}$, $W_{\text{node}}^{\text{remote}}$, τ, and the cache line size of the last-level cache. The first parameter denotes the per-thread rate of contiguously accessing private memory locations. The second parameter is the per-node counterpart for contiguously accessing remote off-node memory locations. Note that we do not distinguish between $W_{\text{thread}}^{\text{private}}$ and intrasocket or intersocket local memory bandwidths, due to very small differences between them. The τ parameter describes the latency for an individual remote-memory access. All the hardware parameters are easily measurable by simple benchmarks, see Section 6.2, or known from hardware specification.

The computation-specific information includes $C_{\text{thread}}^{\text{local,indv}}$, $C_{\text{thread}}^{\text{remote,indv}}$ (Section 5.2.3), $B_{\text{thread}}^{\text{local}}$, $B_{\text{thread}}^{\text{remote}}$ (Section 5.2.4), and $S_{\text{thread}}^{\text{local,out}}$, $S_{\text{thread}}^{\text{local,in}}$, $C_{\text{thread}}^{\text{remote,out}}$, $S_{\text{thread}}^{\text{remote,out}}$, and $S_{\text{thread}}^{\text{remote,in}}$ (Section 5.2.5). These numbers depend on the specific spread of the nonzero values in the sparse matrix. They can be obtained by letting each thread go through its owned blocks of the shared array J and do an appropriate counting. Another important input is the programmer-chosen value of BLOCKSIZE, which controls how all the shared arrays are distributed among the threads, thus determining all the above computation-specific parameters.

6. Experiments

To study the impact of various code transformations described in Section 4 and to validate the corresponding performance models proposed in Section 5, we will use a real-world case of SpMV in this section.

6.1. A 3D Diffusion Equation Solver Based on SpMV. One particular application of SpMV can be found in numerically solving a 3D diffusion equation that is posed on an irregular domain. Typically, an unstructured computational mesh must be used to match the irregular domain. All numerical strategies will involve a time integration process. During time step ℓ, the simplest numerical strategy takes the form of an SpMV:

$$v^{\ell} = M v^{\ell-1}, \tag{19}$$

where vectors v^{ℓ} and $v^{\ell-1}$ denote the numerical solutions on two consecutive time levels, each containing approximate values on some mesh entities (e.g., the centers of all tetrahedrons). The M matrix arises from a numerical discretization of the original diffusion equation. Matrix M is normally time-independent and thus computed once and for all, prior to the time integration process. The unstructured computational mesh will lead to an irregular spread of the nonzeros. Particularly, if a second-order finite volume discretization is applied to a tetrahedral mesh, the number of off-diagonal nonzero values per row of M is up to 16 [14].

Three test problems of increasing resolution will be used in this section. They all arise from modeling the left cardiac ventricle of a healthy male huma. (the 3D diffusion solver can be an integral part of a heart simulator). The three corresponding tetrahedral meshes are generated by the open-source *TetGen* software [15], with the actual size of the meshes being listed in Table 1. Note that we have $r_{\text{nz}} = 16$ for all the three test problems. The tetrahedrons have been reordered in each mesh for achieving good cache behavior associated with a straightforward sequential computation. It is important to notice that all the three meshes are fixed for the following UPC computations, independent of the number of UPC threads used and the value of BLOCKSIZE chosen.

For any computer program implementing the 3D diffusion solver, two arrays are sufficient for containing the two consecutive numerical solutions v^{ℓ} and $v^{\ell-1}$. For the UPC implementations discussed in Section 4, the shared array y corresponds to v^{ℓ} and x to $v^{\ell-1}$ during each time step. The pointers-to-shared y and x need to be swapped before the next time step, fenced between a pair of upc_barrier calls.

6.2. Hardware and Software Platforms. The Abel computer cluster [16] was used to run all the UPC codes and measure their time usage. Each compute node on Abel is equipped with two Intel Xeon E5-2670 2.6 GHz 8-core CPUs and 64 GB of RAM. The interconnect between the nodes is FDR InfiniBand (56 Gbits/s). With a multithreaded STREAM [13] microbenchmark in C, we measured the aggregate memory bandwidth per node as 75 GB/s using 16 threads. This gave

Table 1: Size of the three test problems.

	Test problem 1	Test problem 2	Test problem 3
Number of tetrahedrons, n	6,810,586	13,009,527	25,587,400

$W_{\text{thread}}^{\text{private}}$ = (75/16)GB/s. The internode communication bandwidth, $W_{\text{node}}^{\text{remote}}$ (defined in Section 5.2.2), was measured by a standard MPI ping-pong microbenchmark to be about 6 GB/s.

The Berkeley UPC [17] version 2.24.2 was used for compiling and running all our UPC implementations of SpMV. The compilation procedure involved first a behind-the-scene translation from UPC to C done remotely at Berkeley via HTTP, with the translated C code being then compiled locally on Abel using Intel's icc compiler version 15.0.1. The compilation options were -O3 -wd177 -wd279 -wd1572 -std = gnu99.

In order to measure the cost of an individual remote-memory transfer, τ (defined in Section 5.2.2), we developed a microbenchmark shown in Listing 6. Specifically, v is a shared UPC array created by upc_all_alloc. Each thread then randomly reads entries of v that have affinity with "remote threads," through a thread-private index array mythread_indices. The total time usage, subtracting the time needed to contiguously traverse mythread_indices, can be used to quantify τ. When using two nodes each running 8 UPC threads, we measured the value of τ as 3.4 μs. Varying the number of concurrent threads does not change the measured value of τ very much.

6.3. Time Measurements. Table 2 compares the performance of the naive UPC implementation (Listing 2) against that of the first transformed UPC implementation (Listing 3). Here, we only used one compute node on Abel while varying the number of UPC threads. Each experiment was repeated several times, and the best time measurement is shown in Table 2. Thread binding was always used, as for all the subsequent experiments. Test problem 1 (with 6810586 tetrahedrons) was chosen with the value of BLOCKSIZE being fixed at 65536. We can clearly see from Table 2 that the naive UPC implementation is very ineffective due to using upc_forall and accessing y, D, A, and J through pointers-to-shared.

Table 3 summarizes the time measurements for all the three transformed UPC implementations (denoted by UPCv1, 2, 3) and all the three test problems. It can be seen that UPCv3 has the best performance as expected, followed by UPCv2 with UPCv1 being the slowest. The only exception is that UPCv1 is faster than UPCv2 when running 16 UPC threads on one Abel node. This is because there is no "penalty" of individual remote-memory transfers for UPCv1 in such a scenario, whereas UPCv2 has to transfer all the needed blocks in entirety.

6.4. Validating the Performance Models. We have seen in Section 6.3 that the three transformed implementations have quite different performance behaviors. To shed some light on the causes of the performance differences, we will now use the hardware characteristic parameters obtained in Section 6.2 together with the performance models proposed in Section 5. Specifically, Table 4 compares the actual time measurements against the predicted time usages for Test problem 1 on the Abel cluster. It can be seen that the predictions made by the performance models of Section 5 follow the same trends of the actual time measurements, except for the case of UPCv1 using 128 threads. It is worth noticing that the single-node performance (16 threads) of UPCv2 is correctly predicted to be slower than that of UPCv1, whereas the reverse of the performance relationship when using multiple nodes is also confirmed by the predictions. For small thread counts (16–64), the prediction accuracy is quite good. For larger threads counts, the predictions become less accurate.

For UPCv1, there are four cases where the actual run times are faster than the predictions. These are attributed to the fact that the adopted τ value of 3.4 μs can be a little "pessimistic." Recall from Section 6.2 that the particular τ value was measured by the microbenchmark when it used 8 threads on one node to simultaneously communicate with 8 other threads on another node. In reality, the effective τ value can be smaller than 3.4 μs, if the average number of remotely communicating threads per node over time is fewer than 8. For UPCv3, there are two cases where the actual run times are slightly faster than the predictions. This is due to imbalance between the threads with respect to the per-thread amount of computation and message packing/unpacking. When most of the threads have finished their tasks, the remaining threads will each have access to an effective $W_{\text{thread}}^{\text{private}}$ value that is larger than 1/16 of $W_{\text{node}}^{\text{private}}$. This can result in the time prediction of UPCv3 being a little "pessimistic."

To examine some of the prediction details of UPCv3, we show in Figure 1 the per-thread measurements and predictions of $T_{\text{thread}}^{\text{comp}}$, $T_{\text{thread}}^{\text{unpack}}$, and $T_{\text{thread}}^{\text{pack}}$ (Section 5.2.5), for the particular case of using 32 threads on two nodes. It can be seen that the predictions of the three time components closely match the actual time usages.

As mentioned in Section 4, the three UPC implementations differ in how the interthread communications are handled. To clearly show the difference, we have plotted in the top of Figure 2 the per-thread distribution of communication volumes for the specific case of using 32 threads with BLOCKSIZE set to 65536. We observe that UPCv3 has the lowest communication volume, whereas UPCv2 has the highest. Although UPCv1 induces lower communication volumes than UPCv2, all communications of the former are individual and thus more costly. It is also observed that the communication volumes can vary considerably from thread to thread. The specific variation depends on the spread of the nonzeros, as well as the number of threads used and the value of BLOCKSIZE chosen. The dependency on the last factor is exemplified in the bottom plot of Figure 2. This shows that tuning BLOCKSIZE by the programmer is a viable approach to

```
int nblks = n/BLOCKSIZE + (n% BLOCKSIZE) ?1 : 0;
int mythread_nblks = nblks/THREADS + (MYTHREAD < (nblks% THREADS) ?1 : 0);
shared [BLOCKSIZE] double *v = upc_all_alloc (nblks, BLOCKSIZE * sizeof(double));
double tmp;
int *mythread_indices = (int*)malloc(mythread_nblks * BLOCKSIZE * sizeof(int));
/* let array "mythread_indices" contain random global indices with affinity to "remote threads" */
randomize (mythread_indices, mythread_nblks * BLOCKSIZE);
/* start timing ... */
for (int mb = 0, i = 0; mb < mythread_nblks; mb++)
   for (int k = 0; k < BLOCKSIZE; k++, i++)
      tmp = v[mythread_indices[i]];
/* stop timing ... */
```

Listing 6: A UPC microbenchmark for measuring the latency of individual remote-memory transfers.

Table 2: Time usage (in seconds) of 1000 iterations SpMV for Test problem 1; naive UPC implementation (Listing 2) vs. the first transformed UPC implementation (Listing 3).

	1 thread	2 threads	4 threads	8 threads	16 threads
Naive UPC	895.44	548.57	301.17	173.08	106.10
UPCv1	270.40	159.51	86.37	51.10	28.80

Table 3: Time usage (in seconds) of 1000 iterations SpMV for Test problems 1–3.

	1 node 16 threads	2 nodes 32 threads	4 nodes 64 threads	8 nodes 128 threads	16 nodes 256 threads	32 nodes 512 threads	64 nodes 1024 threads
Test problem 1: 6,810,586 tetrahedrons							
UPCv1	28.80	522.15	443.98	1882.01	551.20	311.54	183.73
UPCv2	39.37	36.70	23.68	18.89	13.61	9.98	9.57
UPCv3	25.01	15.07	8.22	4.65	2.91	2.68	5.56
Test problem 2: 13,009,527 tetrahedrons							
UPCv1	59.14	2525.05	3532.33	3657.95	3078.35	2613.85	1588.67
UPCv2	73.79	69.60	55.33	36.39	24.16	25.06	21.29
UPCv3	46.88	24.97	15.43	10.91	6.25	5.15	7.54
Test problem 3: 25,587,400 tetrahedrons							
UPCv1	115.25	2990.92	1758.94	986.85	1302.52	4653.10	2692.69
UPCv2	154.72	178.14	122.38	81.77	52.99	41.16	44.80
UPCv3	93.30	48.74	26.13	15.37	11.12	7.41	10.16

Table 4: Comparison between actual and predicted time usages (in seconds) of the three transformed UPC implementations for Test problem 1 ($n = 6810586$). The hardware characteristic parameters used for the Abel cluster (16 UPC threads per node) are $W_{\text{thread}}^{\text{private}} = (75/16)$GB/s, $W_{\text{node}}^{\text{remote}} = 6$ GB/s, and $\tau = 3.4\,\mu$s.

THREADS	BLOCKSIZE	$T_{\text{total,actual}}^{\text{UPCv1}}$	$T_{\text{total,predicted}}^{\text{UPCv1}}$	$T_{\text{total,actual}}^{\text{UPCv2}}$	$T_{\text{total,predicted}}^{\text{UPCv2}}$	$T_{\text{total,actual}}^{\text{UPCv3}}$	$T_{\text{total,predicted}}^{\text{UPCv3}}$
16	65536	28.80	26.40	39.37	37.21	25.01	22.95
32	65536	522.15	410.86	36.70	34.30	15.07	14.07
64	65536	443.98	607.08	23.68	20.19	8.22	7.83
128	53200	1882.01	677.99	18.89	12.43	4.65	4.07
256	26600	551.20	679.83	13.61	9.59	2.91	3.06
512	13300	311.54	388.42	9.98	7.83	2.68	2.96
1024	6650	183.73	200.96	9.57	8.15	5.56	3.55

performance optimization. The performance models are essential in this context.

7. Related Work

Many performance studies about UPC programming, e.g., [18–21], selected kernels from the NAS parallel benchmark (NPB) suite [22]. These studies however did not involve irregular fine-grained communication that arises from indirectly indexing the elements of shared arrays. Other published non-NPB benchmarks implemented in UPC, such as reported in [18], had the same limitation. Various UPC implementations of SpMV were studied in [23], but the authors chose to combine a row-wise block

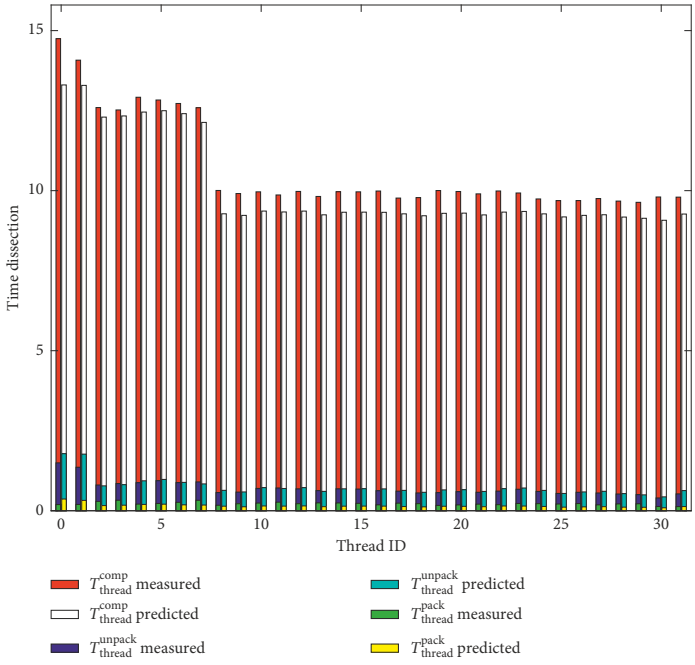
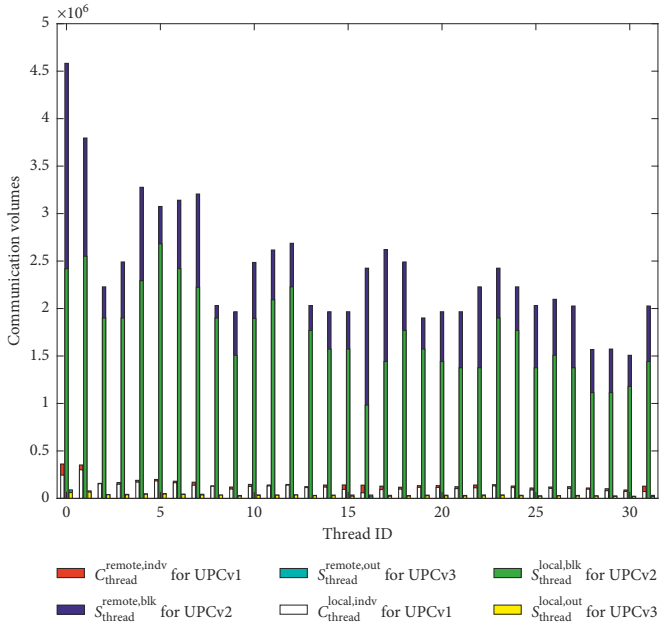


Figure 1: Comparison between per-thread predictions and measurements of T^{comp}_{thread}, T^{unpack}_{thread}, and T^{pack}_{thread} for UPCv3. Test problem 1 ($n = 6810586$), 32 threads spread over two nodes, with BLOCKSIZE = 65536.



(a)

Figure 2: Continued.

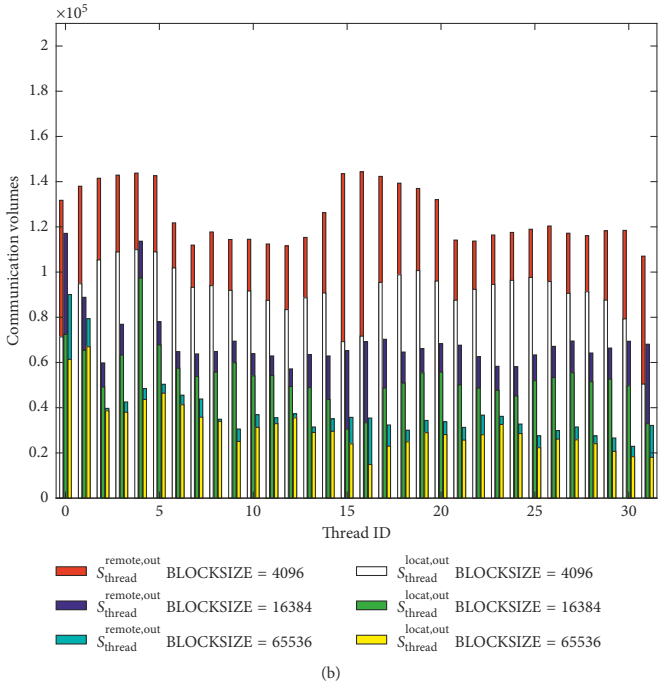


Figure 2: Test problem 1 ($n = 6810586$), 32 threads spread over two nodes. (a) Per-thread communication volumes required by the three transformed UPC implementations with BLOCKSIZE = 65536. (b) Per-thread communication volumes associated with UPCv3 for different values of BLOCKSIZE.

distribution of the sparse matrix with duplicating the entire source vector x on each thread. Such a UPC implementation of SpMV completely avoided the impact of irregular fine-grained communication, which is the main focus of our present paper. The authors of [24] distributed the source vector x among the UPC threads, but their UPC implementation of SpMV explicitly avoided off-node irregular fine-grained communication, for which the needed values of x were transported in batches from the off-node owner threads using, e.g., upc_memget.

An extended suite of UPC STREAM microbenchmarks, including various scenarios of using pointers-to-shared, was proposed by the authors of [21]. They also reported measurements that clearly reveal the additional overhead due to UPC language features. For our purpose of performance modeling, we only found the so-called "random shared read" microbenchmark defined in [21] to be useful for quantifying τ. This prompted us to write our own microbenchmark (Section 6.2) because we have no access to the source code used for [21].

One approach to alleviating the various types of overhead associated with using pointers-to-shared is by specialized UPC compilers [7, 8, 25, 26]. However, indirectly accessing array elements through pointers-to-shared, which will induce irregular fined-grained communication, cannot be handled by any of the specialized UPC compilers. Manual code transformations are thus necessary, such as those proposed in Sections 4.2 and 4.3 in this paper.

It is quite common that performance studies (such as [18]) and performance models (such as [27]) of UPC programs are built upon "single-value statistics" that is accumulated or averaged over all threads. We see this as an unignorable source of inaccuracy, because considerable variations in the communication volume and pattern may exist between threads, as exemplified by Figures 1 and 2. Ignoring such thread-wise imbalances in communication will lead to inaccurate performance predictions, exactly in the same way as ignoring thread-wise imbalances in computation. At the same time, the performance models proposed in Section 5 of our present paper are kept to a minimalistic style, in that we only rely on four easily benchmarked/obtained hardware characteristic parameters: $W_{\text{thread}}^{\text{private}}$, $W_{\text{node}}^{\text{remote}}$, τ, and the last-level cache line size. This stands as a strong contrast to complicated performance modeling approaches such as in [24].

8. Performance Modeling for a 2D Uniform-Mesh Computation

Our performance modeling strategy in Section 5 was originally derived for the case of fine-grained irregular

communication that is due to indirectly indexed accesses to shared arrays. In this section, we will show that the same methodology, as well as the same hardware characteristic parameters, is applicable to other scenarios. As a concrete example, we will use an *existing* UPC implementation that solves a 2D heat diffusion equation $\partial\phi/\partial t = (\partial^2\phi/\partial x^2) + (\partial^2\phi/\partial y^2)$ on a uniform mesh. The UPC code was kindly provided by Dr. Rolf Rabenseifner at HLRS, in connection with a short course on PGAS programming [28].

8.1. Brief Description of the Code

8.1.1. Data Structure. The global 2D solution domain is rectangular, so the UPC threads are arranged as a 2D processing grid, with mprocs rows and nprocs columns (note THREADS equals mprocs ∗ nprocs). Each thread is thus identified by an index pair (iproc,kproc), where iproc = MYTHREAD/nprocs and kproc = MYTHREAD% nprocs. The global 2D domain, of dimension $M \times N$, is evenly divided among the threads. Each thread is responsible for a 2D subdomain of dimension $m \times n$, which includes a surrounding halo layer needed for communication with the neighboring threads. The main data structure consists of the following components:

```
shared[] double * shared xphi[THREADS];
double *phin;
xphi[MYTHREAD] = (shared[]double*)
upc_alloc(m * n * sizeof(double));
phin = (double*) malloc(m * n * sizeof(double));
```

Note that the values in each shared array xphi [MYTHREAD] have affinity to the allocating thread but are accessible by all the other threads. Each thread also allocates a private array named phin to store its portion of the numerical solution for a new time level, to be computed based on the numerical solution for the previous time level, which is assumed to reside in xphi [MYTHREAD].

8.1.2. Halo Data Exchange. The halo data exchange, which is needed between the neighboring threads, is realized by that every thread calls upc_memget on each of the four sides (if a neighboring thread exists). In the vertical direction, the values to be transfered from the upper and lower neighbors already lie contiguously in the memory of the owner threads. There is thus no need to explicitly pack the messages. In the horizontal direction, however, message packing is needed before upc_memget can be invoked towards the left and right neighbors. The following additional data structure is needed with packing and unpacking the horizontal messages:

```
/* scratch arrays for UPC halo exchange of non-
contiguous data */
shared[] double * shared xphivec_coord1first[THREADS];
shared[] double * shared xphivec_coord1last [THREADS];
double *halovec_coord1first, *halovec_coord1last;
xphivec_coord1first[MYTHREAD] = (shared[] double*)
upc_alloc((m − 2) * sizeof(double));
xphivec_coord1last[MYTHREAD] = (shared[] double*)
upc_alloc((m − 2) * sizeof(double));
halovec_coord1first = (double*) malloc((m − 2) * sizeof
(double));
halovec_coord1last = (double*) malloc((m − 2) * sizeof
(double));
```

Consequently, the function for executing the halo data exchange is implemented as follows:

8.1.3. Computation. The computation that is carried out at each time level can be seen in the following time loop:

8.2. Performance Modeling. By slightly changing the formulas in Section 5.2.5, we can model the cost of the different parts involved in the function halo_exchange_intrinsic (Listing 7) as follows:

$$T^{\text{halo,pack}}_{\text{thread}} = T^{\text{halo,unpack}}_{\text{thread}} = \frac{\left(S^{\text{local,horiz}}_{\text{thread}} + S^{\text{remote,horiz}}_{\text{thread}}\right)(\text{sizeof}(\text{double}) + \text{sizeof}(\text{cache line}))}{W^{\text{private}}_{\text{thread}}}, \tag{20}$$

$$T^{\text{halo,memget}}_{\text{node}} = \max_{\forall \text{threads in node}} \frac{2 \cdot S^{\text{local}}_{\text{thread}} \cdot \text{sizeof}(\text{double})}{W^{\text{private}}_{\text{thread}}} + \sum_{\forall \text{threads in node}} \left(C^{\text{remote}}_{\text{thread}} \cdot \tau + \frac{S^{\text{remote}}_{\text{thread}} \cdot \text{sizeof}(\text{double})}{W^{\text{remote}}_{\text{node}}} \right). \tag{21}$$

Comparing (20) with (12) from Section 5.2.5, we can see that the cost due to indirect array indexing (sizeof(int)) is no longer applicable. We have also assumed in (20) that reading/writing from/to noncontiguous locations in the array phi each costs a cache line. The value of $S^{\text{local,horiz}}_{\text{thread}}$ in (20) denotes the total volume of local and remote messages, per thread, to be transfered in the horizontal direction. This can be precisely calculated when the values of m and n, as well as the thread grid layout, are known. Formula (21) is essentially the same as (13) from Section 5.2.5. It is commented that $S^{\text{local}}_{\text{thread}}$ in (21) denotes the total volume of all local messages (in both horizontal and vertical directions) per thread, likewise denotes $S^{\text{remote}}_{\text{thread}}$ the total volume of all remote messages, with $C^{\text{remote}}_{\text{thread}}$ denoting the number of remote messages per thread. Putting them together, the total time spent on halo_exchange_intrinsic is modeled as

```
#define idx(i, k) ((i) * n + (k))
#define rank(ip, kp) ((ip) * nprocs + (kp))
void halo_exchange_intrinsic()
{
  double* phi = (double*) xphi[MYTHREAD];
  int i, k;
  /* packing messages for the horizontal direction */
  if (kproc > 0) {
    double *phivec_coord1first = (double*) xphivec_coord1first[MYTHREAD];
    for (i = 0; i < m - 2; i++)
      phivec_coord1first[i] = phi[idx(i + 1, 1)];
  }
  if (kproc < nprocs - 1) {
    double *phivec_coord1last = (double*) xphivec_coord1last[MYTHREAD];
    for (i = 0; i < m - 2; i++)
      phivec_coord1last[i] = phi[idx(i + 1, n - 2)];
  }
  upc_barrier;
  /* message transfer and unpacking (needed for the horizontal direction) */
  if (kproc > 0) {
    upc_memget(halovec_coord1first, xphivec_coord1last[rank(iproc, kproc - 1)], (m - 2) * sizeof(double));
    for (i = 1; i < m - 1; i++)
      phi[idx(i, 0)] = halovec_coord1first[i - 1];
  if (kproc > nprocs - 1) {
    upc_memget(halovec_coord1last, xphivec_coord1first[rank(iproc,kproc + 1)], (m - 2) * sizeof(double));
    for (i = 1; i < m - 1; i++)
      phi[idx(i, n - 1)] = halovec_coord1last[i - 1];
  }
  if (iproc > 0)
    upc_memget(&phi[idx(0, 1)], &(xphi[rank(iproc - 1, kproc)][idx(m - 2, 1)]), (n - 2) * sizeof(double));
  if (iproc < mprocs - 1)
    upc_memget(&phi[idx(m - 1, 1)], &(xphi[rank(iproc + 1, kproc)][idx(1, 1)]), (n - 2) * sizeof(double));
}
```

Listing 7: The halo data exchange function of an existing 2D heat equation solver.

```
#define idx(i, k) ((i) * n + (k))
for (it = 1; it ≤ itmax; it++) {
  double *phi = (double*) xphi[MYTHREAD];
  int i, k;
  /* communication per time step */
  halo_exchange_intrinsic();
  /* computation per time step */
  for (i = 1; i < m - 1; i++)
    for (k = 1; k < n - 1; k++)
      phin[idx(i, k)] = ((phi[idx(i + 1, k)] + phi[idx(i - 1, k)] - 2 * phi[idx(i, k)]) * dy2i + (phi[idx(i, k + 1)] + phi[idx(i, k - 1)] - 2 * phi
[idx(i, k)]) * dx2i) * dt;
  /* copying the content of phin to phi, checking convergence etc. */
  // ...
}
```

Listing 8: The computational kernel of an existing 2D heat equation solver.

$$T_{2D}^{\text{halo}} = \max_{\forall \text{nodes}} \left(\left(\max_{\forall \text{threads in node}} T_{\text{thread}}^{\text{halo,pack}} \right) + T_{\text{node}}^{\text{halo,memget}} + \left(\max_{\forall \text{threads in node}} T_{\text{thread}}^{\text{halo,unpack}} \right) \right). \tag{22}$$

The time spent on computation during each time step (Listing 8) is modeled as

$$T_{2D}^{\text{comp}} = \frac{3\,(m-2)\,(n-2) \cdot \text{sizeof}\,(\text{double})}{W_{\text{thread}}^{\text{private}}}. \tag{23}$$

Table 5: Comparison of actual and predicted time usages when running 1000 time steps of the 2D heat equation solver. The same hardware characteristic parameters as in Table 4 are used.

THREADS	Partitioning	$T^{\text{halo}}_{\text{2D,actual}}$	$T^{\text{halo}}_{\text{2D,predicted}}$	$T^{\text{comp}}_{\text{2D,actual}}$	$T^{\text{comp}}_{\text{2D,predicted}}$
Mesh size 20000×20000					
16	4×4	0.52	0.33	122.53	122.07
32	4×8	0.44	0.37	61.55	61.04
64	8×8	0.27	0.21	30.78	30.52
128	8×16	0.29	0.21	15.31	15.26
256	16×16	0.18	0.13	7.70	7.63
512	16×32	0.14	0.14	3.85	3.81
Mesh size 40000×40000					
16	4×4	1.55	0.65	489.96	488.28
32	4×8	1.08	0.73	246.25	244.14
64	8×8	0.64	0.42	122.82	122.07
128	8×16	0.64	0.42	61.85	61.04
256	16×16	0.42	0.26	31.01	30.52
512	16×32	0.29	0.26	15.47	15.26

Here we have assumed that every thread has exactly the same amount of computational work. The value of $T^{\text{comp}}_{\text{2D}}$ is estimated on the basis of the minimum amount of traffic between the memory and the last-level cache [29].

Table 5 shows a comparison between the actual time usages of the 2D UPC solver with the predicted values of $T^{\text{halo}}_{\text{2D}}$ and $T^{\text{comp}}_{\text{2D}}$. We have used the same hardware characteristic parameters as in Table 4. It can be seen that the predictions of $T^{\text{comp}}_{\text{2D}}$ agree excellently with the actual measurements, while the prediction accuracy of $T^{\text{halo}}_{\text{2D}}$ is on average 72%.

9. Conclusion

Our starting point is a naive UPC implementation of the SpMV $y = Mx$ in Section 3.2. This naive implementation excessively uses shared arrays, pointers-to-shared, and upc_forall. We have developed three increasingly aggressive code transformations in Section 4 aiming at performance enhancement. The transformations include explicit thread privatization that avoids upc_forall and casts pointers-to-shared to pointers-to-local whenever possible, as well as removing fine-grained irregular communications that are implicitly caused by indirectly indexed accesses to the shared array x. The latter transformation is realized by letting each thread adopt a private mythread_x_copy array that is prepared with explicit one-sided communications (using two different strategies) prior to the SpMV computation. Numerical experiments of a realistic application of SpMV and the associated time measurements reported in Section 6 have demonstrated the performance benefits due to the code transformations. The performance benefits are also justified and quantified by the three performance models proposed in Section 5.

While the code transformations lead to improved performance, the complexity of UPC programming is increased at the same time (trading programmability for performance is by no means specific for our special SpMV example; the textbook of UPC [5] has ample examples). The naive UPC implementation in Section 3.2 shows the easy programmability of UPC that is fully comparable with OpenMP, as discussed in [30]. The first code transformation, in form of explicit thread privatization shown in Section 4.1, may be done by automated code translation. The second and third code transformations, see Sections 4.2 and 4.3, are however more involved. The adoption of one mythread_x_copy array per thread also increases the memory footprint. Despite reduced programmability, all the UPC implementations maintain some advantages over OpenMP in targeting distributed-shared memory systems and promoting data locality. The third code transformation, UPCv3, results in a programming style quite similar to that of MPI. Nevertheless, UPCv3 is easier to code than MPI, because global indices are retained for accessing the entries of array mythread_x_copy. An MPI counterpart, where all arrays are explicitly partitioned among processes, will have to map the global indices to local indices. Moreover, one-sided explicit communications via UPC upc_memget and upc_memput functions are easier to use. Performance advantage of UPC's one-sided communication over the MPI counterpart has also been reported in [31]. On the other hand, persistent advantages of MPI over UPC include better data locality and more flexible data partitionings. A comparison of performance and programmability between UPC and MPI is given in [32] for a realistic fluid dynamic implementation. For a general comparison between OpenMP, UPC, and MPI programming, we refer to [20].

It should be stressed that the SpMV computation is chosen for this paper as an illustrating example of fine-grained irregular communication that may arise in connection with naive UPC programming. The focus is not on the SpMV itself, but on the code transformations and the performance models in general. Moreover, our performance models are to a great extent independent of the UPC programming details, but rather focusing on the incurred communication style, volume, and frequency. The hardware characteristic parameters $W^{\text{private}}_{\text{thread}}$, $W^{\text{remote}}_{\text{node}}$, and the cache line size are equally applicable to similar communications and memory-bandwidth bound computations implemented by other programming models rather than UPC. Even the latency of individual remote-memory accesses, τ, can alternatively be measured by a standard MPI ping-pong

benchmark. Our philosophy is to represent a target hardware system by only four characteristic parameters, whereas the accuracy of the performance prediction relies on an accurate counting of the incurred communication volumes and frequencies. Accurate counting is essential and thus cannot be generalized, because different combinations of the problem size, number of threads, and block size will almost certainly lead to different levels of performance for the same parallel implementation.

Data Availability

The data used to support the findings of this study are included in the tables and figures within the article.

Disclosure

The current affiliation of Martina Prugger is Vanderbilt University, 2201 West End Ave, Nashville, TN 37235, USA.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

This work was performed on hardware resources provided by UNINETT Sigma2—the National Infrastructure for High Performance Computing and Data Storage in Norway, via Project NN2849K. The work was supported by the European Union's Horizon 2020 Research and Innovation Programme under grant agreement no. 671657, the European Union Seventh Framework Programme (grant no. 611183), and the Research Council of Norway (grant nos. 231746/F20, 214113/F20, and 251186/F20). Martina Prugger was supported by the VSC ResearchCenter funded by the Austrian Federal Ministry of Science, Research and Economy (bmwfw) and by the Doctoral Programme Computational Interdisciplinary Modelling (DK CIM) at the University of Innsbruck.

References

[1] G. Almasi, "PGAS (partitioned global address space) languages," in *Encyclopedia of Parallel Computing*, D. Padua, Ed., pp. 1539–1545, Springer, Berlin, Germany, 2011.

[2] D. E. Culler, A. Dusseau, S. C. Goldstein et al., "Parallel programming in split-C," in *Proceedings of 1993 ACM/IEEE Conference on Supercomputing (Supercomputing'93)*, pp. 262–273, Portland, OR, USA, November 1993.

[3] M. de Wael, S. Marr, B. de Fraine, T. van Cutsem, and W. de Meuter, "Partitioned global address space languages," *ACM Computing Surveys*, vol. 47, no. 4, pp. 1–27, 2015.

[4] PGAS—Partitioned Global Address Space, 2016, http://www.pgas.org.

[5] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick, *UPC: Distributed Shared Memory Programming*, John Wiley & Sons, Hoboken, NJ, USA, 2005.

[6] UPC Consortium, "UPC language specifications version 1.3," 2013, http://upc.lbl.gov/docs/user/upc-lang-spec-1.3.pdf.

[7] W.-Y. Chen, *Optimizing partitioned global address space programs for cluster Architectures*, Ph.D. thesis, University of California at Berkeley, Berkeley, CA, USA, 2007.

[8] W.-Y. Chen, C. Iancu, and K. Yelick, "Communication optimizations for fine-grained UPC applications," in *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, St. Louis, MO, USA, September 2005.

[9] R. Grimes, D. Kincaid, and D. Young, "ITPACK 2.0 User's Guide," Technical Report CNA-150, Center for Numerical Analysis, University of Texas, Austin, TX, USA, 1979.

[10] Y. Zheng, "Optimizing UPC programs for multi-core systems," *Scientific Programming*, vol. 18, no. 3-4, pp. 183–191, 2010.

[11] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[12] J. Langguth, N. Wu, J. Chai, and X. Cai, "Parallel performance modeling of irregular applications in cell-centered finite volume methods over unstructured tetrahedral meshes," *Journal of Parallel and Distributed Computing*, vol. 76, pp. 120–131, 2015.

[13] J. D. McCalpin, "STREAM: sustainable memory bandwidth in high performance computers," Technical Report, University of Virginia, Charlottesville, VA, USA, 2007.

[14] J. Langguth, M. Sourouri, G. T. Lines, S. B. Baden, and X. Cai, "Scalable heterogeneous CPU-GPU computations for unstructured tetrahedra l meshes," *IEEE Micro*, vol. 35, no. 4, pp. 6–15, 2015.

[15] H. Si, "TetGen, a delaunay-based quality tetrahedral mesh generator," *ACM Transactions on Mathematical Software*, vol. 41, no. 2, pp. 1–36, 2015.

[16] The Abel computer cluster, 2018, https://www.uio.no/english/services/it/research/hpc/abel/.

[17] Berkeley UPC—Unified Parallel C, 2018, http://upc.lbl.gov.

[18] C. Barton, C. Caşcaval, and J. N. Amaral, "A characterization of shared data access patterns in UPC programs," in *Proceedings of 19th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2006)*, vol. 4382, pp. 111–125, Springer, New Orleans, LA, USA, November 2006.

[19] T. El-Ghazawi and F. Cantonnet, "UPC performance and potential: a NPB experimental study," in *Proceedings of ACM/IEEE SC 2002 Conference (SC'02)*, Baltimore, MD, USA, November 2002.

[20] H. Shan, F. Blagojević, S.-J. Min et al., "A programming model performance study using the NAS parallel benchmarks," *Scientific Programming*, vol. 18, no. 3-4, pp. 153–167, 2010.

[21] Z. Zhang and S. R. Seidel, "Benchmark measurements of current UPC platforms," in *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*, Denver, CO, USA, April 2005.

[22] D. Bailey, E. Barszcz, J. Barton et al., "The NAS parallel benchmarks," Technical Report RNR-94-007, NASA Ames Research Center, Mountain View, CA, USA, 1994.

[23] J. González-Domínguez, Ó. García-López, G. L. Taboada, M. J. Martín, and J. Touriño, "Performance evaluation of sparse matrix products in UPC," *Journal of Supercomputing*, vol. 64, no. 1, pp. 100–109, 2013.

[24] S. Li, C. Hu, J. Zhang, and Y. Zhang, "Automatic tuning of sparse matrix-vector multiplication on multicore clusters," *Science China Information Sciences*, vol. 58, no. 9, pp. 1–14, 2015.

[25] M. Alvanos, *Optimization techniques for fine-grained communication in PGAS environments*, Ph.D. thesis, Universitat Politècnica de Catalunya, Barcelona, Spain, 2013.
[26] F. Cantonnet, T. El-Ghazawi, P. Lorenz, and J. Gaber, "Fast address translation techniques for distributed shared memory compilers," in *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*, Denver, CO, USA, April 2005.
[27] Z. Zhang and S. R. Seidel, "A performance model for fine-grain accesses in UPC," in *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS'06)*, Rhodes Island, Greece, April 2006.
[28] R. Rabenseifner, *Short Course: Introduction to Unified Parallel C (UPC) and Co-array Fortran (CAF)*, HLRS, University of Stuttgart, Stuttgart, Germany, 2015.
[29] H. Stengel, T. Jan, G. Hager, and G. Wellein, "Quantifying performance bottlenecks of stencil computations using the execution-cache-memory model," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, pp. 207–216, ACM, Irvine, CA, USA, June 2015.
[30] A. Marowka, "Execution model of three parallel languages: OpenMP, UPC and CAF," *Scientific Programming*, vol. 13, no. 2, pp. 127–135, 2005.
[31] K. Z. Ibrahim, P. H. Hargrove, C. Iancu, and K. Yelick, "An evaluation of one-sided and two-sided communication paradigms on relaxed-ordering interconnect," in *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium (IPDPS'14)*, pp. 1115–1125, IEEE, Phoenix, AZ, USA, May 2014.
[32] M. Prugger, L. Einkemmer, and A. Ostermann, "Evaluation of the partitioned global address space (PGAS) model for an inviscid Euler solver," *Parallel Computing*, vol. 60, pp. 22–40, 2016.

Paper III

A Newcomer In The PGAS World - UPC++ vs UPC: A Comparative Study

Jérémie Lagravière, Johannes Langguth, Martina Prugger, Phuong Hoai Ha, Xing Cai

Not published yet

III

A Newcomer In The PGAS World - UPC++ vs UPC: A Comparative Study

Jérémie Lagravière[1] Johannes Langguth[1] Martina Prugger [2]
Phuong H. Ha[3] Xing Cai[1,4]

[1]Simula Research Laboratory, P.O. Box 134, NO-1325 Lysaker, Norway

[2] Lopez Laboratory, Vanderbilt University , Nashville, Tennessee, USA. [3]The Arctic University of Norway, NO-9037 Tromsø, Norway

[4]University of Oslo, NO-0316 Oslo, Norway

Correspondence should be addressed to Jérémie Lagravière; jeremie.lagraviere@gmail.com

Abstract

A newcomer in the Partitioned Global Address Space (PGAS) 'world' has arrived in its version 1.0: Unified Parallel C++ (UPC++). UPC++ targets distributed data structures where communication is irregular or fine-grained. The key abstractions are global pointers, asynchronous programming via RPC, futures and promises. UPC++ API for moving non-contiguous data and handling memories with different optimal access methods resemble those used in modern C++. In this study we provide two kernels implemented in UPC++: a sparse-matrix vector multiplication (SpMV) as part of a Partial-Differential Equation solver, and an implementation of the Heat Equation on a 2D-domain. Code listings of these two kernels are available in the article in order to show the differences in programming style between UPC and UPC++. We provide a performance comparison between UPC and UPC++ using single-node, multi-node hardware and many-core hardware (Intel Xeon Phi Knight's Landing).

Keywords: PGAS; APGAS;UPC++; UPC programming language; Fine-grained irregular communication; Sparse matrix-vector multiplication; Performance optimization

1 Introduction & Motivation

In distributed memory parallel systems, MPI has been the *de-facto* standard for a long time. It reliably provides high communication performance, but programming MPI applications is very complex, and this complexity of parallel programming is a key challenge that the HPC research and industry communities face. One of the most prominent approaches to alleviate this problem is the use of Partitioned Global Address Space (PGAS) systems. For more than 20 years [6, 18], the dominating PGAS implementations have been UPC, Coarray Fortran, and SHMEM. Other implementations have also been proposed such as Chapel [11], Titanium, and recently UPC++ [24], whose version 1.0 was released in September 2019. The PGAS programming model constitutes an alternative [16] to using MPI or MPI with OpenMP. In this study we focus on UPC++.

We have multiple goals for this study: we want to extend our previous studies [13, 14] and focus on what could be seen as the future of PGAS by studying the new Version 1.0 of UPC++ PGAS language [24]), released in September 2019. To this end, we compare the performance and programability of UPC and UPC++.

PGAS [2, 8, 10, 19] is a programming model that aims to achieve both good programmer productivity and high computational performance. These goals are usually conflicting in the context of developing parallel code for scientific computations. The main aspect of PGAS is a global address space that is shared among concurrent processes, running on different nodes of a supercomputer that jointly execute a parallel program. Data exchange between the processes is typically performed transparently by a low-level network layer such as GASNet [5], without explicit involvement from the programmer, thus providing good productivity. The shared global address space is logically partitioned such that each partition has affinity to a designated owner process. This awareness of data locality is essential for achieving good performance of parallel programs written in the PGAS model, because the globally shared address space may encompass many physically distributed memory sub-systems.

As mentioned before, UPC and UPC++ are both implementations of the PGAS programming model. UPC stands for *Unified Parallel C* and UPC++ stands for *Unified Parallel C++*. In both cases data is either `shared` or `private`, where shared data is accessible (read or write) by all threads and private data is only accessible by its owner thread. UPC is based on the idea of "communication simplification" in the sense that there is no way for the programmer to distinguish intra-node memory operations (between threads running on the same hardware node) from their inter-node counterparts. This brings a very efficient and simple way of creating programs either in UPC at the very possible expense of performance. By default, UPC++ follows the same pattern, but with the recent addition of *teams* of threads, it offers a native way to distinguish between intra- and inter-node communication.

However, the main difference between UPC and UPC++, on a conceptual level, is that UPC++ implements a sub-category of PGAS, which is called APGAS: "Asynchronous Partitioned Global Address Space". The APGAS model extends the PGAS with ideas from task-based asynchronous execution model, which describes the semantics of an application as a hierarchy of tasks that are dynamically created and scheduled during the execution time [17]. Another language that implements the APGAS paradigm is X10 [26]. Furthermore, for software maintenance reasons, UPC++ is distributed as a library rather than a language, although this has little effect on the actual PGAS programming.

To study UPC++ we have developed two applications based on well-known kernels: the heat equation in 2D and a sparse-matrix vector multiplication using the ELLPACK format. It is designed to simulate diffusion processes over unstructured 3D mesh representing the human cardiac ventricle. In this paper we compare UPC++ to its ancestor UPC on both multi-node and many-core architectures using multiple criteria such as: programmability, performance predictability, scalability, performance (GFLOPS or execution time). After this comparison we discuss the obtained results.

Previous studies [3, 27], made in part by the UPC++ design team, have included comparative aspects about UPC++. Our study focuses on comparing UPC++ to UPC by using the SpMV kernel which, due to its irregularity and communication requirements, poses a challenging benchmark for distributed memory systems.

2 Implementations

2.1 Kernels

In this study we have focused our efforts on the implementations of two kernels in UPC and UPC++. The first kernel is rather simple and is the implementation of the Heat Equation in two dimensions domain, presented in detail in Section 2.1.1. The second kernel is the implementation of a Sparse matrix-vector multiplication, solving a 3D diffusion equation that is posed on an irregular domain modeling the left cardiac ventricle of a healthy male human. This 3D diffusion solver was designed as an integral part of a cardiac electrophysiology simulator and is presented in detail in Section 2.1.2. This section aims also at showing the difference between the programming 'style' of UPC and UPC++.

2.1.1 Heat Equation

The heat equation is one of the most fundamental kernels in scientific computing. Due to its simplicity, it is useful as a benchmark to assess the performance of a parallel system. For this experiment, we solve the 2D heat diffusion equation:

$$\frac{\partial \phi}{\partial t} = \frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2}$$

on a uniform mesh. We employ the finite difference discretization given by:

$$u(x,y) = \frac{u-h}{2h}$$

We use an existing UPC code[1] that was tested in previous work [13]. The UPC++ implementation is derived from this UPC implementation. Both the UPC and UPC++ codes implement a 2D heat equation solver using halo-exchange with a single layer of ghost cells to communicate data between threads. We use X and Y to denote the number of cells in each dimension of the domain. Each processing elements receives a rectangular sub-domain of approximately equal size. In order to ensure the communication of data to adjacent subdomain(s) we use the well-known halo exchange technique [9]. In both our implementations of the 2D Heat Equation the domains receive their initial values before the computation starts. The heat diffusion is then computed by performing N time steps. For each time step, each processing-element performs two tasks: computing the diffusion by updating each cell in its subdomain, and communicating the newly computed results on the domain boundary to adjacent subdomains. All time steps perform the same amount of computation, but due to cache effects and the unpredictability of network communication their execution time can vary. For benchmarking purposes it is important to obtain the average time per time step. Typically, running for $N = 1000$ time steps is sufficient to do so.

2.1.2 Sparse matrix-vector multiplication

A general matrix-vector multiplication is compactly and mathematically defined by the following formula: $y = Mx$. In this study, without loss of generality, we assume that the matrix M is square, having n rows and n columns. Both the input vector and the result vector, respectively denoted by x and y, are of length n.

[1] The code was kindly provided by Dr. Rolf Rabenseifner at HLRS, in connection with a course on PGAS programming [20]

Then, to compute element number i of the result vector y we apply the following general formula (using zero-based indices):

$$y(i) = \sum_{0 \le j < n} M(i,j)x(j). \tag{1}$$

M is called a *sparse* matrix, if most of the $M(i,j)$ values are zero. In this case the above formula becomes unnecessarily expensive from a computational point of view. A more economic formula for computing $y(i)$ in a sparse matrix-vector multiplication (SpMV) is thus:

$$y(i) = \sum_{M(i,j) \neq 0} M(i,j)x(j), \tag{2}$$

This formula takes into account only the nonzero values of matrix M on each row. Because only the nonzero values are used, it is then memory-wise unnecessarily expensive to store all the n^2 values of a sparse matrix. As a result various compact storage formats for sparse matrices have been adopted, such as:

- the coordinate format (COO);
- the compressed sparse row format (CSR);
- the compressed sparse column format (CSC);
- and the EllPack format [12].

For sparse matrices that have a homogeneous number of nonzero values per row, it is usual to use the EllPack storage format. The EllPack storage format usually uses two 2D tables. These two 2D tables are of the same size: having n rows and the number of columns equaling the maximum number of non-zeros per row. The first 2D table contains all the nonzero values of the sparse matrix. Whereas the second 2D table contains the corresponding column indices of the non-zeros. Explicitly stored zeros are used for padding when the rows that have fewer than the maximum number of non-zeros. Additionally, if we assume that all the values on the main diagonal of a sparse matrix M are nonzero, which is applicable to most scientific applications, it is beneficial to split M so that:

$$M = D + A \tag{3}$$

In this formula, D is the main diagonal of M, while A contains the off-diagonal part of M. Then, we can use a modified EllPack format where the main diagonal D is stored as a 1D array of length n. There is no need to store the column indices of these nonzero diagonal values, because their column indices equal the row indices by definition. We suppose that r_{nz} now denotes the maximum number of non-zero off-diagonal values per row. Thus, for storing the values in the off-diagonal part A, it is usual to use two 1D arrays both of length $n \cdot r_{nz}$ (instead of two $n \times r_{nz}$ 2D tables). Therefore, one 1D array contains the off-diagonal values consecutively row by row, whereas the other 1D array contains the corresponding integer column indices [13].

The goal of this chapter is to present a detailed view of the our implementations using UPC++. In addition, for comparison purposes, we also show the corresponding UPC code implementing the same kernel as in UPC++. We want to emphasize the differences in the UPC and UPC++ programming models and their consequences in terms of code.

2.2 Definitions

In our UPC++ implementations of SpMV and the Heat Equation we use repeatedly *global pointers* and communication function such as *rget* or *rput*. In this section we present definitions for these terms and functionalities.

2.2.1 UPC++'s shared arrays and global pointers: PGAS in action

A UPC++ program can allocate global memory in shared segments, which are accessible by all processes. A global pointer points at storage within the global memory, and is declared as follows:

```
upcxx::global_ptr<int> gptr = upcxx::new_<int>(upcxx::rank_me());
```

The call to `upcxx::new_<int>` allocates a new integer in the calling process's shared segment, and returns a global pointer (`upcxx::global_ptr`) to the allocated memory. Each process has its own private pointer (gptr) to an integer in its local shared segment. By contrast, a conventional C++ dynamic allocation (int *mine = new int) will be in private local memory. Note that we use the integer type in this paragraph as an example, but any type T can be allocated using the `upcxx::new_<T>()` function call [23].

In our implementation of both SpMV and the Heat Equation we use shared arrays and global pointers. The declaration and allocation of shared arrays is a multiple step process, as described in Listing 6 (page 9) with a buffer called `share_receive_buffer`.

2.2.2 UPC++: rget and rput

Both UPC++ SpMV codes, i.e. SpMV using block-wise data transfer and SpMV using message condensing and consolidation use the UPC++ function `upcxx::rput`. This function is used to send data from one thread to another asynchronously by default. In our implementation of the UPC++ Heat Equation 2D we use the UPC++ function `upcxx::rget`. Both functions are part of the one-sided communication model that is implemented in UPC++. These operations initiate transfer of the value object to (put) or from (get) the remote process; no coordination is needed with the remote process since it is a one-sided communication. Like many asynchronous communication operations, rget and rput default to returning a future object [25] that becomes ready when the transfer is complete [23].

2.3 Implementations of sparse matrix-vector multiplication

In our study we have implemented multiple versions of the Sparse Matrix-Vector Multiplication (SpMV) both in UPC and UPC++. The difference between these versions lies essentially in the communication methodology. Thus, in this section we describe 4 implementations of SpMV using a modified ELLPack format, as described in Section 2.1.2.

These 4 versions are:

- UPC SpMV using Block-wise data transfer: Section 2.3.2, Listing 3, based on our previous study [13];
- UPC++ SpMV using Block-wise data transfer: Section 2.3.2, Listing 4;
- UPC SpMV using message condensing and consolidation: Section 2.3.3, Listing 5, based on our previous study [13];
- UPC++ SpMV using message condensing and consolidation: Section2.3.3, Listings 7, 6.

Note that in UPC `upcxx::rank_me` corresponds to `MYTHREAD`, meaning 'id number of calling thread' and `upcxx::rank_n` corresponds to THREADS, meaning 'total number of threads used for the current run'.

For both UPC and UPC++ the code can be summarized as follows: first, we read and distribute the data in memory, then we prepare the data (particularly preparing data exchange buffers), and after that the actual SpMV computation occurs.

2.3.1 Explicit thread privatization

In both UPC and UPC++ codes presented in the following sections, we have used as much as possible a strategy that we call *explicit thread privatization*. The goal of this technique is to ensure that each thread accesses (read and write) its own local data. Doing so means that no implicit communication is triggered and no-illegal memory access is performed, thus accessing the data thanks to *explicit thread privatization* delivers the best possible performance. To achieve such a goal, we use the well-known technique of casting *pointers-to-shared* to *pointers-to-local*. To ensure that we create pointers that point to data with affinity to the calling thread (`MYTHREAD` in UPC, `upcxx:rank_me` in UPC++), we use the `BLOCKSIZE` to point to the local data in the computation loop. This technique is illustrated in the Listing 1 for UPC and in Listing 2 for UPC++.

```
for (int mb=0; mb<mythread_nblks; mb++) {
  int offset = (mb*THREADS+MYTHREAD)*BLOCKSIZE;
  /* casting shared pointers to local pointers */
  double *loc_y = (double*) (y+offset);
  //...
}
```

Listing 1: Explicit thread privatization in UPC (Complete code in Listing 3)

```
for (mb=0; mb<mythread_nblks; mb++) {
  offset = (mb*upcxx::rank_n()S+upcxx::rank_me())*BLOCKSIZE;
  /* casting shared pointers to local pointers */
  double *loc_y = (double*) (y[upcxx::rank_me()]+offset);
  //...
}
```

Listing 2: Explicit thread privatization in UPC++

2.3.2 Block-Wise Data Transfer between Threads

In this section, we present a summarized view of UPC and UPC++ SpMV implementing block-wise data transfer, as seen in Listings 3 and 4. The UPC SpMV using block-wise data transfer is the same as the one we used in our previous study [13]. The UPC++ SpMV using block-wise data transfer is derived from this aforementioned UPC version, in the sense that it implements the same logic of computation and communication. To the exception that in UPC we used `upc_memget`, which gets data from another thread's shared memory , for communication and for UPC++ we used `upcxx::rput`, which sends data to another thread's shared memory.

In these versions of UPC and UPC++ SpMV, the communication of data is done block-wise. This means that for UPC if `Thread A` requires one element `X` to `Thread B`, a block of data of size `BLOCKSIZE` containing element `X` will be sent from `Thread B` to `Thread A`. In other words, each needed block is transported in its *entirety*, independent of the actual number of values needed in that block. For UPC++

the operation is initiated from the sending thread as we have used a one-sided `put` function whereas in UPC we have used a one-sided `get` function.

```
//Reading the data set and data distribution over THREADS then prepare the data for computation
double *mythread_x_copy = (double*) malloc(n*sizeof(double));
/* Prep-work: check for each block of x whether it has values needed by MYTHREAD; make a private boolean
     array 'block_is_needed' of length nblks */
// ....
/* Transport the needed blocks of x into mythread_x_copy */
for (int b=0; b<nblks; b++)
   if (block_is_needed[b])
     upc_memget(&mythread_x_copy[b*BLOCKSIZE],&x[b*BLOCKSIZE],
                min(BLOCKSIZE,n-b*BLOCKSIZE)*sizeof(double));
/* SpMV: each thread only goes through its designated blocks */
int mythread_nblks = nblks/THREADS+(MYTHREAD<(nblks%THREADS)?1:0);
for (int mb=0; mb<mythread_nblks; mb++) {
  int offset = (mb*THREADS+MYTHREAD)*BLOCKSIZE;
  /* casting shared pointers to local pointers */
  double *loc_y = (double*) (y+offset);double *loc_D = (double*) (D+offset);
  double *loc_A = (double*) (A+offset*r_nz);int *loc_J = (int*) (J+offset*r_nz);
  /* computation per block */
  for (int k=0; k<min(BLOCKSIZE,n-offset); k++) {
    double tmp = 0.0;
    for (int j=0; j<r_nz; j++)
      tmp += loc_A[k*r_nz+j]*mythread_x_copy[loc_J[k*r_nz+j]];
    loc_y[k] = loc_D[k]*mythread_x_copy[offset+k] + tmp;
  }
}
```

Listing 3: UPC implementation of SpMV by block-wise communication [13]

```
//Reading the data set and data distribution over THREADS then prepare the data for computation
double *mythread_x_copy = (double*) malloc(n*sizeof(double));
/* Prep-work: check for each block of x whether it has values needed by upcxx::rank_me(); make a private
     boolean array 'block_is_needed' of length nblks */
//Creating a future in order to store all communication requests
upcxx::future<> futureComms = upcxx::make_future();
/* SpMV: each thread only goes through its designated blocks */
int mythread_nblks = nblks/upcxx::rank_n()+(upcxx::rank_me()<(nblks%upcxx::rank_n())?1:0);
for (int mb=0; mb<mythread_nblks; mb++) {
  int offset = (mb*upcxx::rank_n()+upcxx::rank_me())*BLOCKSIZE;
  /* casting shared pointers to local pointers */
  double *loc_y = (double*) (y[upcxx::rank_me()]+offset); double *loc_D = (double*) (D[upcxx::rank_me()
      ]+offset);
  double *loc_A = (double*) (A[upcxx::rank_me()]+offset*r_nz);
  int * loc_J = (int*)I[upcxx::rank_me()].local()+offset*r_nz;
  /* computation per block */
  for (int k=0; k<min(BLOCKSIZE,n-offset); k++) {
    double tmp = 0.0;
    for (int j=0; j<r_nz; j++)
      tmp += loc_A[k*r_nz+j]*mythread_x_copy[loc_J[k*r_nz+j]];
    loc_y[k] = loc_D[k]*mythread_x_copy[offset+k] + tmp;
  }
/* Transport the needed blocks of x into mythread_x_copy */
for (int targetThread=0;targetThreads<numberOfTargets, targetThread++)
        for (int b=0; b<nblks; b++)
            if (block_is_needed[b])
                futureComms = upcxx::when_all(futureComm, upcxx::rput(&mythread_x_copy[b*BLOCKSIZE],
          x[targetThread]+(b*BLOCKSIZE), min(BLOCKSIZE,n-b*BLOCKSIZE)));
//Waiting for all communication to be completed
futureComms.wait();
```

Listing 4: UPC++ SpMV implementation of 'Block-Wise Data Transfer between Threads'

In the UPC++ implementation (Listing 4) we use a *future* [25], called `futureComms` in conjunction with `upcxx::when_all`. Unlike standard C++ futures, UPC++ futures and promises are used to manage asynchronous dependencies within a thread and not for direct communication between threads or

processes. A future thus represents the consumer side of a non-blocking operation. Each non-blocking operation has an associated promise object, which is created either explicitly by the user or implicitly by the runtime when the non-blocking operation is invoked. A promise represents the producer side of the operation, and it is through the promise that the results of the operation are supplied and its dependencies fulfilled. A user can pass a promise to a UPC++ communication operation, which registers a dependency on the promise and subsequently fulfills the dependency when the operation completes. The same promise can be passed to multiple communication operations, and through a single wait call on the associated future, the user can be notified when all the operations have completed [3].

The future conjoining begins by invoking `upcxx::make_future` to construct a trivially ready future `futureComms`. A new future, returned by `upcxx::rput` is passed to the `upcxx::when_all` function, in combination with the previous future, `futureComms`. The `upcxx::when_all` constructs a new future representing readiness of all its arguments, and returns a future with a concatenated results tuple of the arguments. By setting `futureComms` to the future returned by `upcxx::when_all`, we can extend the conjoined futures. Once all the processes are linked into the conjoined futures, we simply wait on the final future, i.e. the `futureComms.wait` call [23].

In the UPC++ implementation (Listing 4, 7) we have used Futures in addition to `upcxx::when_all`, `upcxx::rput` and `<upcxx::future>.wait`, which implement one-sided, asynchronous, bulk communication. And in UPC we have used, one-sided, bulk communication using `upc_memget`. By default, all communications in UPC++ are asynchronous, which, in terms of code corresponds to use of at least one instruction for sending or receiving data and one instruction to wait for the completion of the communication. In our UPC++ implementation, we have chosen to distinguish the sending of data performed by `upcxx::rput` and the waiting for completion performed by `futureComms.wait`. This means that multiple communications are launched in a non-blocking manner without having to wait for one-another in order to start transmitting data. This is done at the expense and with a measured risk of bandwidth (between CPU and RAM) and network (The test system in our experiments uses *Infiniband*, as described in section 3).

2.3.3 Message condensing and consolidation

In this versions of UPC and UPC++ SpMV, we implement a different communication technique. Contrary to the block-wise data transfer presented in section 2.3.2, where each communication request is of a full block size (`BLOCKSIZE`), here, we employ a communication strategy where each thread communicates the strict amount of required data to other threads. In other words, the length of a message from thread `A` to thread `B` equals the number of *unique* values in the `k` blocks owned by `A` that are needed by `B`, times the number of bytes required to represent each value (8 byte double in our experiments). All the between-thread messages are thus condensed and consolidated.

Listing 6 (page 9) and Listing 7 (page 10) present respectively the preparation step and the computation step of the UPC++ version of SpMV using message condensing and consolidation. Listing 5 (page 9) presents the preparation and computation steps of the UPC version of SpMV using message condensing and consolidation.

This communication patterns and the use of the UPC++ function `upcxx::rput` leads to the necessity of a *Shared Receive Buffer* (SRB). This SRB will be accessed by each thread that needs to communicate data, and then the receiving thread will get the data from this buffer after the all threads are done communicating. In both UPC and UPC++ implementations we have the necessity for the SRB. In UPC++ the implementation of the SRB is different and slightly more complicated than in UPC. SRBs in UPC++ require the use of a two-dimensional vector and the corresponding declarations and function calls to pop-

ulate it and broadcast the information across all threads (`upcxx::broadcast`).

```
/* Allocation of the five shared arrays x, y, D, A, J */
// ...
/* Allocation of an additional private x array per thread */
double *mythread_x_copy = (double*) malloc(n*sizeof(double));

/* Preparation step: create and fill the thread-private arrays of int *mythread_num_send_values, int *
    mythread_num_recv_values, int **mythread_send_value_list, int **mythread_recv_value_list, double **
    mythread_send_buffers. Also, shared_recv_buffers is prepared. */
// ....

/* Communication procedure starts */
int T,k,mb,offset;
double *local_x_ptr = (double *)(x+MYTHREAD*BLOCKSIZE);
for (T=0; T<THREADS; T++)
  if (mythread_num_send_values[T]>0) /* pack outgoing messages */
    for (k=0; k<mythread_num_send_values[T]; k++)
      mythread_send_buffers[T][k] = local_x_ptr[mythread_send_value_list[T][k]];

for (T=0; T<THREADS; T++)
  if (mythread_num_send_values[T]>0) /* send out messages */
    upc_memput(shared_recv_buffers[T*THREADS+MYTHREAD], mythread_send_buffers[T],
               mythread_num_send_values[T]*sizeof(double));

upc_barrier;

int mythread_nblks = nblks/THREADS+(MYTHREAD<(nblks%THREADS)?1:0);
for (mb=0; mb<mythread_nblks; mb++) {/* copy own x-blocks */
  offset = (mb*THREADS+MYTHREAD)*BLOCKSIZE;
  memcpy(&mythread_x_copy[offset], (double *)(x+offset), min(BLOCKSIZE,n-offset)*sizeof(double));
}

for (T=0; T<THREADS; T++)
  if (mythread_num_recv_values[T]>0) {/* unpack incoming messages */
    double *local_buffer_ptr = (double *)shared_recv_buffers[MYTHREAD*THREADS+T];
    for (k=0; k<mythread_num_recv_values[T]; k++)
      mythread_x_copy[mythread_recv_value_list[T][k]] = local_buffer_ptr[k];
  }
/* Communication procedure ends */

/* SpMV: each thread only goes through its designated blocks */
for (mb=0; mb<mythread_nblks; mb++) {
  offset = (mb*THREADS+MYTHREAD)*BLOCKSIZE;
  /* casting shared pointers to local pointers */
  double *loc_y = (double*) (y+offset);
  double *loc_D = (double*) (D+offset);
  double *loc_A = (double*) (A+offset*r_nz);
  int *loc_J = (int*) (J+offset*r_nz);
  /* computation per block */
  for (k=0; k<min(BLOCKSIZE,n-offset); k++) {
    double tmp = 0.0;
    for (int j=0; j<r_nz; j++)
      tmp += loc_A[k*r_nz+j]*mythread_x_copy[loc_J[k*r_nz+j]];
    loc_y[k] = loc_D[k]*mythread_x_copy[offset+k] + tmp;
  }
}
```

Listing 5: An improved UPC implementation of SpMV by message condensing and consolidation [13]

```
/* Allocation of the five shared arrays x, y, D, A, J */
// Preparation Begins
/* Allocation of an additional private x array per thread */
double *mythread_x_copy = (double*) malloc(n*sizeof(double));

/* Preparation step: create and fill the thread-private arrays of int *mythread_num_send_values, int *
    mythread_num_recv_values, int **mythread_send_value_list, int **mythread_recv_value_list, double **
    mythread_send_buffers.*/
```

```
//Preparation of the shared_receive_buffer
vector<std::vector<global_ptr<double> > >shared_receive_buffer(upcxx::rank_n());
std::vector<global_ptr<double>> temp(upcxx::rank_n());
for(int i = 0; i < upcxx::rank_n(); i++)
 temp[i] = upcxx::new_array<double>(localReceiveCount[i]);

shared_receive_buffer[upcxx::rank_me()] = temp;

for(int i = 0; i<upcxx::rank_n();i++)
 shared_receive_buffer[i] = upcxx::broadcast(shared_receive_buffer[i], i).wait();
//Creating a "empty" future, used to store communication requests
upcxx::future<> futureComms = upcxx::make_future();
```

Listing 6: An improved UPC++ implementation of SpMV by message condensing and consolidation: Preparation step

```
//computation starts
/* SpMV: each thread only goes through its designated blocks */
for (mb=0; mb<mythread_nblks; mb++) {
  offset = (mb*upcxx::rank_n()S+upcxx::rank_me())*BLOCKSIZE;
  /* casting shared pointers to local pointers */
  double *loc_y = (double*) (y[upcxx::rank_me()]+offset);
  double *loc_D = (double*) (D[upcxx::rank_me()]+offset);
  double *loc_A = (double*) (A[upcxx::rank_me()]+offset*r_nz);
  int * loc_J = (int*)I[upcxx::rank_me()].local()+offset*r_nz;
  /* computation per block */
  for (k=0; k<min(BLOCKSIZE,n-offset); k++) {
    double tmp = 0.0;
    for (int j=0; j<r_nz; j++)
      tmp += loc_A[k*r_nz+j]*mythread_x_copy[loc_J[k*r_nz+j]];
    loc_y[k] = loc_D[k]*mythread_x_copy[offset+k] + tmp;
  }
}

/* Communication procedure starts */
int T,k,mb,offset;
double *local_x_ptr = (double *)(x+upcxx::rank_me()*BLOCKSIZE);
for (T=0; T<upcxx::rank_n()S; T++)
  if (mythread_num_send_values[T]>0) /* pack outgoing messages */
    for (k=0; k<mythread_num_send_values[T]; k++)
      mythreadsend_buffers[T][k] = local_x_ptr[mythreadsend_value_list[T][k]];

for (T=0; T<upcxx::rank_n()S; T++) {
  if (mythread_num_send_values[T]>0) /* send out messages */
 futureComms.when_all(futureComms, upcxx::rput( mythread_send_buffers[T], shared_recv_buffers[T][upcxx::
     rank_me()], mythread_num_send_values[T]));
}
//waiting for all communication to be completed
futureComms.wait();
upc_barrier;
//Receiving data, unpacking it into mythread_x_copy
for (T=0; T<upcxx::rank_n()S; T++)
  if (mythreadnum_recv_values[T]>0) {/* unpack incoming messages */
    double *local_buffer_ptr = (double *)shared_recv_buffers[upcxx::rank_me()*upcxx::rank_n()S+T];
    for (k=0; k<mythread_num_recv_values[T]; k++)
      mythread_x_copy[mythread_recv_value_list[T][k]] = local_buffer_ptr[T][k];
  }
/* Communication procedure ends */
```

Listing 7: An improved UPC++ implementation of SpMV by message condensing and consolidation: Computation step

2.4 Implementations of heat equation 2D

The UPC code used in this section is identical to the one used in our previous study [13]. The UPC code was kindly provided by Dr. Rolf Rabenseifner at HLRS, in connection with a short course on PGAS programming [20]. The UPC++ code for Heat Equation 2D is inspired from the UPC code in that it uses identical policies for data distribution, computation and data communication through halo data exchange. Thus, in this section we propose the following implementations of a solver for the Heat Equation on a 2-dimensional domain:

- UPC implementation is presented in the following listings:
  - "Scratch" arrays for data exchange in Listing 8 (page 11)
  - Halo data exchange in Listing 10 (page 12)
- UPC++ implementation is presented in the following listings:
  - "Scratch" arrays for data exchange in Listing 9 (page 12)
  - Halo data exchange in Listing 11 (page 13)

In our implementations of UPC and UPC++ Heat Equation, the global 2D solution domain is rectangular, so the UPC and UPC++ threads are arranged as a 2D processing grid, with `mprocs` rows and `nprocs` columns. (Note `upcxx::rank_n` equals `mprocs*nprocs`.) Each thread is thus identified by an index pair `(iproc,kproc)`, where `iproc = upcxx::rank_me / nprocs` and `kproc = upcxx::rank_me % nprocs`. The global 2D domain, of dimension $M \times N$, is evenly divided among the threads. Each thread is responsible for a 2D sub-domain of dimension $m \times n$, which includes a surrounding halo layer needed for communication with the neighboring threads. Reminder: In UPC++ `upcxx::rank_me` corresponds to `MYTHREAD` in UPC and, in UPC++, `upcxx::rank_n` corresponds to `THREADS`, in UPC.

In both UPC and UPC++ implementations we use a halo data exchange, which is needed between the neighboring threads. The halo data exchange performs communication, where each thread calls `upc_memget` for UPC or `upcxx::rget` for UPC++, on each of the four sides of its subdomain (if a neighboring thread exists). In the vertical direction, the values to be transferred from the upper and lower neighbors already lie contiguously in the memory of the owner threads. There is thus no need to explicitly pack the messages. In the horizontal direction, however, message packing is needed before `upc_memget` can be invoked towards the left and right neighbors.

Listing 8 presents additional data structure which is needed with packing and unpacking the horizontal messages. Listing 9, presents the same idea implemented in UPC++. This short example of code also shows the additional instructions required by the use of UPC++: the declaration of shared arrays and their allocation in memory requires more code than in UPC.

```
shared [] double * shared xphivec_coord1first[THREADS];
shared [] double * shared xphivec_coord1last [THREADS];
double *halovec_coord1first, *halovec_coord1last;

xphivec_coord1first[MYTHREAD] =
  (shared [] double *) upc_alloc((m-2)*sizeof(double));
xphivec_coord1last[MYTHREAD]  =
  (shared [] double *) upc_alloc((m-2)*sizeof(double));
halovec_coord1first = (double *) malloc((m-2)*sizeof(double));
halovec_coord1last  = (double *) malloc((m-2)*sizeof(double));
```

Listing 8: Scratch arrays for UPC halo exchange of non-contiguous data [13]

```
std::vector<upcxx::global_ptr<double>> xphivec_coord1first;
std::vector<upcxx::global_ptr<double>> xphivec_coord1last;
xphivec_coord1first.resize(upcxx::rank_n());
xphivec_coord1last.resize(upcxx::rank_n());
xphivec_coord1first[upcxx::rank_me()] = upcxx::new_array<double>(m-2);
xphivec_coord1last[upcxx::rank_me()] = upcxx::new_array<double>(m-2);
for (int i = 0; i < upcxx::rank_n(); i++) {
 xphivec_coord1first[i] = upcxx::broadcast(xphivec_coord1first[i], i).wait();
 xphivec_coord1last[i] = upcxx::broadcast(xphivec_coord1last[i], i).wait();
}

phivec_coord1first = (double *) xphivec_coord1first[upcxx::rank_me()].local();
phivec_coord1last  = (double *) xphivec_coord1last[upcxx::rank_me()].local();

double *halovec_coord1first = (double *) calloc((m-2), sizeof(double));
double *halovec_coord1last  = (double *) calloc((m-2),  sizeof(double));
```

Listing 9: Scratch arrays for UPC++ halo exchange of non-contiguous data

The communication implementation through halo exchange is different in UPC and UPC++. Listing 10 presents the communication performed in UPC using `upc_memget`, and performing a packing of the horizontal data beforehand. Listing 11 presents the implementation of halo data exchange using `upcxx::rget` and performing a packing of the horizontal data beforehand.

In the UPC++ version it is important to notice that we store futures `upcxx::future` in a `std::vector` called `getRequests`, this vector is then used in the function `waitForGetRequestsToEnd`. This function calls the UPC++ `wait` function on each `upcxx::future` contained in the vector `getRequests`. This is different technique to wait for all communication to complete compared to the one used in Listing 7 (page 10), where we used the combination of `upcxx::when_all` and `upcxx::wait`.

The communication technique used in UPC and UPC++ relies on the same halo data exchange model, however, the amount of code needed in UPC++ is greater than the one needed in UPC. As explained earlier in this paper, this difference in the amount of code is directly related to the way shared arrays are declared in UPC and UPC++ and the fact that communication in UPC++ is done in two steps: `upcxx::rget` and the `wait` statement whereas in UPC a single function call to `upc_memget` is needed.

```
#define idx(i,k) ((i)*n+(k))
#define rank(ip,kp) ((ip)*nprocs+(kp))
void halo_exchange_intrinsic() {
  double* phi = (double *) xphi[MYTHREAD]; int i,k;
  /* packing messages for the horizontal direction */
  if (kproc>0) {
    double *phivec_coord1first = (double *) xphivec_coord1first[MYTHREAD];
    for (i=0; i<m-2; i++)       { phivec_coord1first[i] = phi[idx(i+1,1)]; }
  }
  if (kproc<nprocs-1) {
    double *phivec_coord1last = (double *) xphivec_coord1last[MYTHREAD];
    for (i=0; i<m-2; i++)       { phivec_coord1last[i] = phi[idx(i+1,n-2)]; }
  }
  upc_barrier;
  /* message transfer and unpacking (needed for the horizontal direction) */
  if (kproc>0) {
    upc_memget(halovec_coord1first, xphivec_coord1last[rank(iproc,kproc-1)], (m-2)*sizeof(double));
    for (i=1; i<m-1; i++)       { phi[idx(i,0)] = halovec_coord1first[i-1]; }
  }
  if (kproc<nprocs-1) {
    upc_memget(halovec_coord1last, xphivec_coord1first[rank(iproc,kproc+1)], (m-2)*sizeof(double));
    for (i=1; i<m-1; i++)       { phi[idx(i,n-1)] = halovec_coord1last[i-1]; }
  }
  if (iproc>0)
    upc_memget(&phi[idx(0,1)], &(xphi[rank(iproc-1,kproc)][idx(m-2,1)]), (n-2)*sizeof(double));
  if (iproc<mprocs-1)
    upc_memget(&phi[idx(m-1,1)], &(xphi[rank(iproc+1,kproc)][idx(1,1)]), (n-2)*sizeof(double));
}
```

Listing 10: The halo data exchange function of an existing UPC 2D heat equation solver [13]

```
//Standard CPP vector containing UPC++ Futures
std::vector<upcxx::future<>> getRequests(4)
void halo_exchange_intrinsic() {
 int i,k;
 /* packing messages for the horizontal direction */
 if (kproc>0)
  for (i=0;i<m-2;i++) { phivec_coord1first[i] = phi[idx(i+1,1)]; }

 if (kproc<nprocs-1)
  for (i=0;i<m-2;i++) { phivec_coord1last[i] = phi[idx(i+1,n-2)]; }

 upcxx::barrier();
 /* message transfer and unpacking (needed for the horizontal direction) */
 if (iproc>0) {
  getRequests[0] = upcxx::rget(xphi[getRank(iproc-1,kproc)]+idx(m-2,1), &phi[idx(0,1)], n-2); }
 if (iproc<mprocs-1) {
  getRequests[1] = upcxx::rget(xphi[getRank(iproc+1,kproc)]+idx(1,1), &phi[idx(m-1,1)], n-2); }
 if (kproc>0) {
  getRequests[2] = upcxx::rget(xphivec_coord1last[getRank(iproc,kproc-1)], halovec_coord1first, m-2);
  for (i=1;i<m-1;i++) {phi[idx(i,0)] = halovec_coord1first[i-1];}}

 if (kproc<nprocs-1) {
  getRequests[3] = upcxx::rget(xphivec_coord1first[getRank(iproc,kproc+1)], halovec_coord1last, m-2);
  for (i=1;i<m-1;i++) { phi[idx(i,n-1)] = halovec_coord1last[i-1];}}
 waitForGetRequestsToEnd();
}
void waitForGetRequestsToEnd() {
 if (iproc>0)            { getRequests[0].wait(); }
 if (iproc<mprocs-1)     { getRequests[1].wait(); }
 if (kproc>0)            { getRequests[2].wait(); }
 if (kproc<nprocs-1)     { getRequests[3].wait(); }
}
```

Listing 11: The halo data exchange function of our UPC++ implementation for 2D heat equation solver

3 Experimental Setup

Our performance measurements focus on many-core and multi-node hardware architectures.

For the many-core experiments we run our UPC and UPC++ codes on a machine equipped with one Intel Xeon Phi 7250 (Knights Landing, or KNL) processor, which is equipped with 16GB of high speed MCDRAM. Due to the hardware structure of the KNL processor we can select the way MCDRAM is seen (addressed and accessed) by the operating system and the programs. For our experiments we chose the *flat mode*. When the Knights Landing processor is booted in *flat mode*, the entirety of the MCDRAM is used as addressable memory. MCDRAM as addressable memory shares the physical address space with DDR4, and is also cached by the L2 cache. With respect to Non Uniform Memory Access (NUMA) architecture, the MCDRAM portion of the addressable memory is exposed as a separate NUMA node without cores, with another NUMA node containing the DDR4 memory [7]. We use *numactl* to explicitly place data in the MCDRAM NUMA node. KNL processors can alternatively be booted in *cache mode*, which uses the MCDRAM as cache which is thus transparent for the operating system, and in *hybrid mode* which used 8GB as cache and 8GB as addressable memory. In previous work we found that for problems which fit entirely within the 16GB of MCDRAM, which is the case for our test instances, the performance of *cache mode* is only marginally lower than that of *flat mode* [15]. On this machine we used

Intel Compiler version 17.0.0 to compile UPC and UPC++ compilers and runtimes environments, as well as our programs.

The Abel computer cluster [1] was used to run all the UPC and UPC++ codes on multi-node and measure their time usage. Each compute node on Abel is equipped with two Intel Xeon E5-2670 2.6 GHz 8-core CPUs and 64 GB of RAM. The interconnect between the nodes is FDR InfiniBand (56 Gbits/s). In our previous study [13], we measured the memory bandwidth per node on Abel and obtained 75 GB/s; we also measured the inter-node communication bandwidth and obtained about 6 GB/s. On this supercomputer we have used nodes providing access to 16 physical cores, and for all our runs we always have used the maximum amount of physical cores per node. In the following, when speaking of cores we mean *physical core*. We have not used any kind of purely logical cores offered by Intel's HyperThreading technology.

The Berkeley UPC [4] version 2.24.2 was used for compiling and running all our UPC implementations of SpMV and Heat Equation. The compilation procedure involved first a behind-the-scene translation from UPC to C done remotely at Berkeley via HTTP, with the translated C code being then compiled locally on Abel using Intel's `icc` compiler version 15.0.1. The compilation options are `-O3 -std=gnu99`.

The Berkeley UPC++ [22] version 1.0 (2019.9.0) was used for compiling and running all our UPC++ implementation of SpMV and Heat Equation. UPC++ relies on a local compiler and MPI installation (i.e. located on the supercomputer). MPI is used to spawn UPC++ process on each node. GNU G++ version 7.2.0 was used to compile C++ code, and OpenMPI version 3.1.2 was used for process spawning.

Both UPC and UPC++ use GASNet(-EX) as an under-layer to ensure communication between threads, processes or cores and nodes. This also means, that UPC and UPC++ are APIs for GASNet. GASNet is a language-independent networking middleware layer that provides network-independent, high-performance communication primitives including Remote Memory Access (RMA) and Active Messages (AM). It has been used to implement parallel programming models and libraries such as UPC, UPC++, Co-Array Fortran, Legion, Chapel, and many others. The interface is primarily intended as a compilation target and for use by runtime library writers (as opposed to end users), and the primary goals are high performance, interface portability, and expressiveness. GASNet stands for "Global-Address Space Networking" [5].

In terms of technology, having GASNet as an under-layer means that UPC and UPC++ have to stay synchronized with the GASNet project in order to benefit from the latest version of it. It also means for the user that a GASNet version change requires recompiling of both the UPC or UPC++ compiler and runtime environment, as well as all the programs. The same process is required when new features are implemented in UPC or UPC++: a full recompiling of the whole tool-chain and all the programs.

It is important to specify that UPC++ relies on a newer version of GASNet called GASNet-EX. Also, newest versions of UPC, not used in this paper, rely on GASNet-EX. In [5], the authors give a detailed presentation of GASNet-EX as well as a short definition of GASNet-EX's "philosophy": "GASNet-EX is the next generation of the GASNet-1 communication system, continuing our commitment to provide portable, high-performance, production-quality, open-source software. The GASNet-EX upgrade is being done over the next several years as part of the U.S. Department of Energy's Exascale Computing Program (ECP). The GASNet interfaces are being redesigned to accommodate the emerging needs of exascale supercomputing, providing communication services to a variety of programming models on current and future HPC architectures. This work builds on fifteen years of lessons learned with GASNet-1, and is informed and motivated by the evolving needs of distributed runtime systems." A set of improvements between GASNet and GASNet-EX are also presented in [5]:

- Retains GASNet-1's wide portability (laptops to supercomputers)

- Provides backwards compatibility for the dozens of GASNet-1 clients,
- including multiple UPC and CAF/Fortran08 compilers
- Focus remains on one-sided RMA and Active Messages
- Reduces CPU and memory overheads
- Improves many-core and multi-threading support
- “Immediate mode” injection to avoid stalls due to back-pressure
- Explicit handling of local-completion (source buffer lifetime)
- New AM interfaces, e.g. to reduce buffer copies between layers
- Vector-Index-Strided for non-contiguous point-to-point RMA
- Remote Atomics, implemented with NIC offload where available
- Subset teams and non-blocking collectives

4 Results

In this chapter, we focus on presenting the results obtained with our implementations of UPC++ SpMV (see Section 2.3, page 5) and UPC++ Heat Equation 2D (see Section 2.4, page 11). First we will show that it is possible to get reproducible performance with UPC++. We then discuss the importance of the `BLOCKSIZE` on the data distribution and the obtained performance. Then, we will focus on comparing performance obtained in UPC++ with that of UPC. The UPC results come from our previous study [13].

In this section we use named implementations of UPC++, the correspondence with the implementations presented in section 2 is as follows:

- ”UPC++SpMV Version 1” corresponds to ”UPC++ SpMV with Block-Wise Data Transfer between Threads” presented in Section 2.3.2 (see page 6)
- ”UPC++SpMV Version 2” corresponds to ”UPC++ SpMV with Message condensing and consolidation” presented in Section 2.3.3 (see page 8)
- ”UPC Heat Equation” corresponds to the implementation of UPC the Heat Equation in 2D presented in Section 2.4 (see page 11)
- ”UPC++ Heat Equation” corresponds to the implementation of the UPC++ Heat Equation in 2D presented in Section 2.4 (see page 11)

Threads (node)	UPC SpMV version 1	UPC SpMV version 2	UPC++ SpMV version 1	UPC++SpMV version 2
16 (1)	6.22	9.20	5.78	10.11
32 (2)	6.75	15.20	8.05	17.86
64 (4)	11.46	27.72	12.97	30.99
128 (8)	18.62	48.35	17.18	50.55
256 (16)	24.14	73.50	26.10	80.54
512 (32)	29.57	103.77	27.43	105.01
1024 (64)	28.41	41.61	26.84	192.00

Table I: Performance results in GFLOPS for UPC++ version 1 and 2 versus UPC SpMV version 1 and 2. Performance obtained on Abel super-computer (described in section 3 see page 13), computing on 16 to 1024 threads (1 to 64 nodes). Using human heart 3D representation, using 6.8 millions tetrahedrons ($n = 6810586$). For graphical representation see Figure 2.

4.1 Data distribution: reproducibility and influence on performance

We focus on the reproducibility of the obtained performance with UPC++ since it is a crucial feature of a new language such as UPC++, both for verifying its usefulness in high performance computing and for ensuring the validity of our results. In addition, being able to expect repeatable performance from UPC++ is a prerequisite for designing a performance model capable of predicting obtainable performance.

In Figure 1, we show results measured on the Abel supercomputer using UPC++ SpMV version 2 running on two different counts of threads: 64 (4 nodes) and 128 threads (8 nodes), processing an instance representing a healthy human heart composed of 6.8 million tetrahedrons. For each thread count we have used different values of `BLOCKSIZE` in order to investigate two aspects: first whether the `BLOCKSIZE` has a strong impact on performance, and second that when running the same UPC++ SpMV program multiple times with the same parameters `Number of threads - BLOCKSIZE` we obtain stable and repeatable performance.

Figure 1 shows both that we obtain performance stability and that the `BLOCKSIZE` has a strong impact on performance. This impact in performance is related to the fact that the chosen `BLOCKSIZE` in our implementations affects directly the data distribution. Consequently, for this particular data set (6.8 millions tetrahedrons), the observed performance varies significantly depending on the chosen value of `BLOCKSIZE`: for 128 threads (8 nodes) performance go from slightly more than 50 GFLOPS down to slightly less than 10 GFLOPS when using either 65536 or 524288 as `BLOCKSIZE`. For 64 threads, going from a block size of 1024 to 8192 helps to improve the performance, however, when the block size is bigger than 8192 we observe a slight drop in the obtained performance.

4.2 Scalability and Performance

Concerning multi-node results for UPC SpMV and UPC++ SpMV: Table I and Figure 2 present the obtained performance for UPC and UPC++ SpMV versions 1 and 2 using again the Abel supercomputer. Additionally, we present the obtained speedups for UPC and UPC++ SpMV versions 1 and 2 in Table II. We obtain speed-ups strictly superior to 1 for all versions, and all amounts of threads (except for 1024 threads (64 nodes) for UPC SpMV version 1 and 2 and UPC++ version 1). Also, the obtained speed-up on the Abel supercomputer for all versions of UPC and UPC++ SpMV is always strictly inferior to 2.00.

Based on these results we can say that we achieve strong scaling in UPC SpMV and UPC++ SpMV

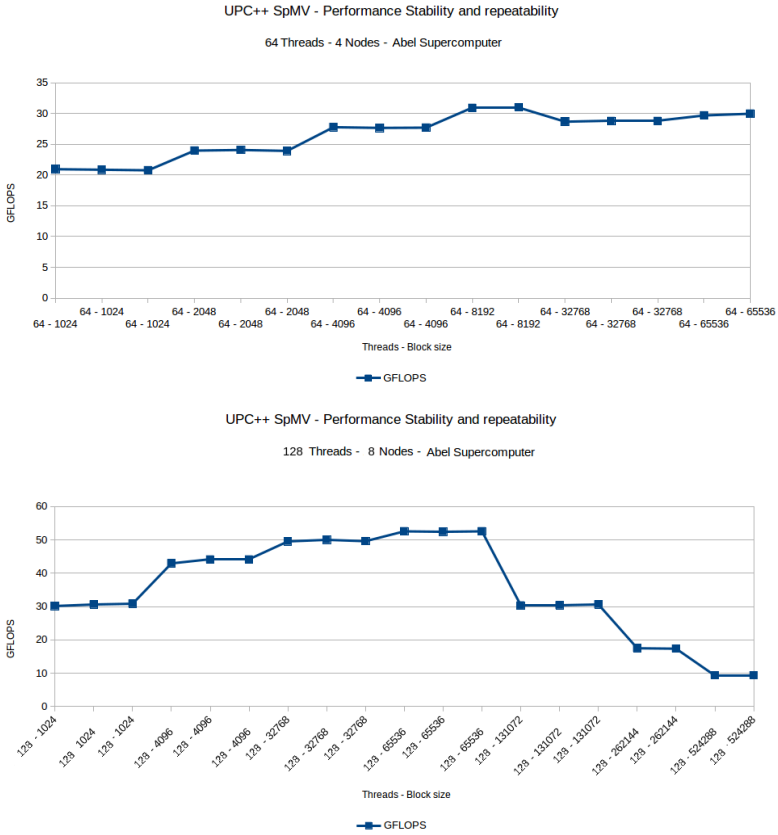


Figure 1: Performance stability and repeatability and importance of the `BLOCKSIZE` on the obtained performance. Using human heart 3D representation, using 6.8 millions tetrahedrons ($n = 6810586$). Using 64 and 128 threads spread over 4 and 8 nodes respectively.

version 1 and 2 using the Abel supercomputer for the multi-node experiments (except for UPC version 1 and 2 and UPC++ version 1 for 1024 threads (64 nodes)). The obtained speedups can be called *strong scaling* in the sense that for the same total amount of data (fixed problem size) we get higher performance when using more processing elements from 16 to 512 cores (1 to 32 nodes), when using 1024 threads only UPC++ version 2 continues to scale up: 1.83 speed-up (see Table II, page 19) between UPC++ version 2 using 1024 threads (64 nodes) compared to performance obtained with UPC++ version 2 using 512 threads (32 nodes).

In Figure 2, it is clear that message condensing and consolidation (which corresponds to UPC SpMV version 2 and UPC++ SpMV version 2) delivers far better performance than block wise data transfer (which corresponds to UPC version 1 and UPC++ version 1).

This is to be expected as the versions 2 of UPC and UPC++ SpMV use a far more sophisticated communication pattern, leading to messages sent between threads/cores/nodes of a size that corresponds

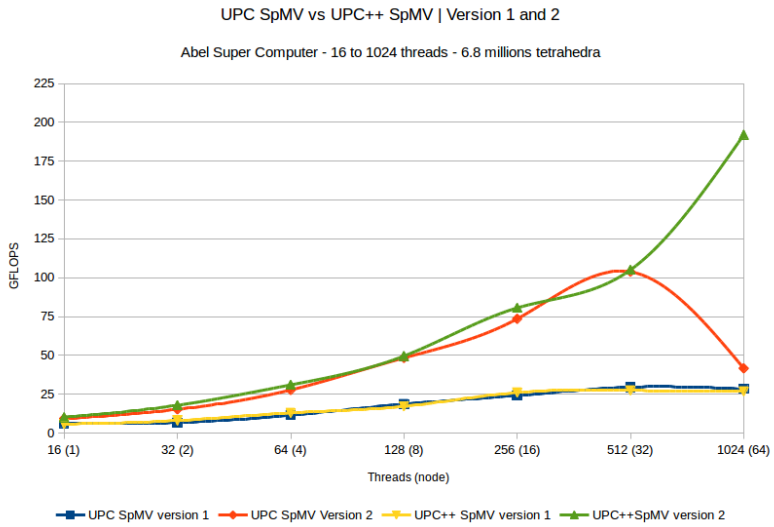


Figure 2: Multi-node performance for SpMV using Abel Supercomputer (abel.uio.no). Using 1 to 64 nodes, 16 to 1024 threads. Performance are expressed in GFLOPS (higher is better). Using human heart 3D representation, using 6.8 millions tetrahedrons ($n = 6810586$). Table I contains numerical results represented in this graph.

exactly to the amount of data we need to transfer. Whereas in the block-wise data transfer versions (UPC SpMV and UPC++ SpMV versions 1), messages are always of size `BLOCKSIZE` even if only 1 element (C-type "double", 8 bytes on 64 bits architecture) needs to be sent. Thus, communication containing only one element that is needed will be of size $1 \times 8(bytes)$ in UPC and UPC++ SpMV versions 2, and of size $1 \times BLOCKSIZE \times 8(bytes)$ in UPC and UPC++ SpMV versions 1. As the communication pattern is the only main difference in UPC and UPC++ versions 1 and 2, it is possible to say that the communication pattern is the main source of difference in performance. In addition, we have shown in our previous study that this performance can be predicted using our performance model [13], and we show in this study that on multi-node the heaviest factor influencing performance is the inter-node communication through the Infiniband network.

Table III (Page 19) and Figure 3 show the obtained performance for UPC SpMV and UPC++ SpMV versions 1 and 2 on the Intel Xeon Phi Knight's Landing (as presented in Section 3, page 13). Globally, UPC and UPC++ SpMV in versions 2 perform better than UPC and UPC++ SpMV versions 1. As explained for the multi-node results, the main difference between these two versions is the communication pattern. In this single-node context running on a many-core architecture, the difference in the communication pattern has a huge influence on performance. As shown earlier in this section and in section 2.3 (page 5) and in [13], block-wise data transfer, as implemented in UPC and UPC++ SpMV versions 1, generate far more data traffic than message condensing and consolidation, as implemented in UPC and UPC++ versions 2. This difference in the amount of data exchanged between cores on a many-core such as Intel Xeon Phi Knight's Landing architecture is large enough to lead to bottlenecks between the processing elements and the MCDRAM (see Section 3, page 13).

Threads (node)	UPC SpMV version 1	UPC SpMV version 2	UPC++ SpMV version 1	UPC++SpMV version 2
16 (1)	-	-	-	-
32 / 16	1.08	1.65	1.39	1.77
64 / 32	1.70	1.82	1.61	1.73
128 / 64	1.62	1.74	1.32	1.60
256 / 128	1.30	1.52	1.52	1.63
512 / 256	1.22	1.41	1.05	1.30
1024 / 512	0.96	0.40	0.98	1.83

Table II: Performance results in speed-up for UPC++ version 1 and 2 versus UPC SpMV version 1 and 2. Performance obtained on Abel supercomputer (described in Section 3 see page 13), computing on 16 to 1024 threads (1 to 64 nodes). Using the human heart 3D representation with 6.8 millions tetrahedrons ($n = 6810586$). As an example on how to read this table: For UPC SpMV version 2 the speed-up between using 128 threads and 64 threads is 1.74.

Additionally, since the Intel Knight's Landing's hardware architecture uses four-way *HyperThreading*, i.e. each of the 68 physical cores appears to the operating system as four logical cores. Since the logical cores do not add computational resources, we cannot expect linear scaling from using them, as observed in [21]. This explains why when using 136 cores we only get a slight increase in performance. When using 272 cores we observe a performance drop for all versions of UPC and UPC++ SpMV.

Table IV and Figure 4, present the obtained results for our implementations of UPC and UPC++ Heat Equation using a 20000x20000 domain running on Abel supercomputer using 16 to 1024 cores (1 to 64 nodes). Both the UPC and UPC++ Heat Equation versions adopt a similar way of communicating data and a similar computation kernel, as presented in Section 2.4, page 2.4. With that in mind we observe that UPC++ performs as well as UPC. In this case, the data per processor is fixed independently of the amount of cores that is used, thus, as for our implementations of SpMV, we can observe that we achieve weak scaling in both UPC and UPC++, at least from 1 to 32 nodes.

The obtained performance is extremely close between the UPC and UPC++ implementations of the Heat Equation, from 1 to 32 nodes. However when using 64 nodes (1024 cores) UPC++ has a strong advantage over UPC. We observed this behavior of UPC having trouble to perform with more than 512 threads before [13, 14].

Threads	UPC SpMV version 1	UPC SpMV version 2	UPC++ SpMV version 1	UPC++ SpMV version 2
16	4.76	5.76	5.00	5.71
32	8.57	10.77	8.43	12.08
68	13.17	19.95	13.50	21.59
136	14.62	23.61	14.60	23.57
272	13.39	18.72	14.08	21.16

Table III: Performance results in GFLOPS for UPC++ version 1 and 2 versus UPC SpMV version 1 and 2. Performance obtained on Intel Xeon Phi Knight's Landing many-core processor (Described in section 3 see page 13), computing on 16 to 272 threads. Using human heart 3D representation, using 6.8 millions tetrahedrons ($n = 6810586$). For graphical representation see Figure 3.

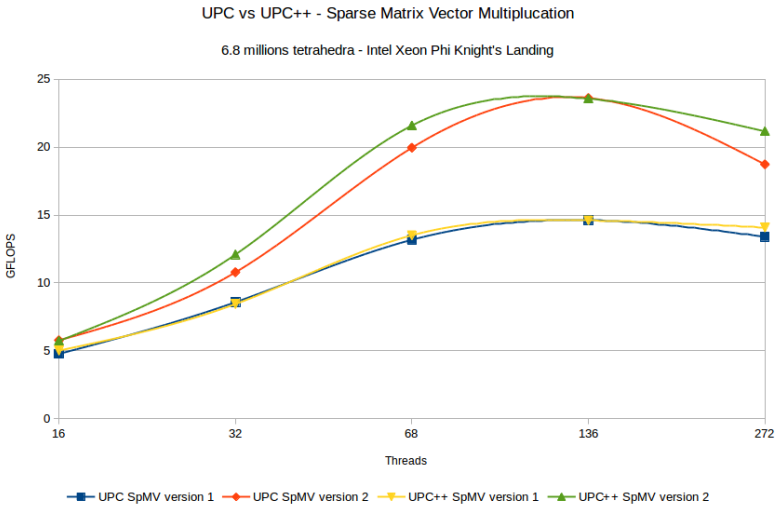


Figure 3: Single-node performance for SpMV using Intel Xeon Phi Knight's Landing (described in section 3, see page 13). Using human heart 3D representation, using 6.8 millions tetrahedrons ($n = 6810586$).

5 Discussion

In Section 2 (page 3), we have presented our UPC and UPC++ implementations of SpMV and the Heat Equation in 2D. It is important to note the significant differences in both programming model and consequences in terms of code between UPC and UPC++. For instance, the way the shared arrays are accessed in UPC is done relatively transparently in the sense that a shared array of size `N` will be accessible with indices ranging from `0` to `N-1`. However, in UPC++ the same array will be accessible in a 2-dimensional way, the first dimension explicitly points at the thread ID, the second dimension corresponds to a position in the array for the portion allocated to the chosen thread.

In other words, UPC++ does not try to maintain the same ease of programming at the expense of performance as UPC. As shown in previous work [13], accessing shared arrays in UPC can be extremely costly as it generates *hidden* communication: hidden in two senses, first the programmer does not explicitly implement the communication and the developer has no way to know whether this communication will be between cores, sockets or nodes. In other words, in UPC communication can be implicit and yield extremely bad performance whereas in UPC++ all communication is done explicitly. However, UPC++ performs also communication without having the developer know whether the communication is between cores, sockets or nodes (unless *teams* are used). This simply means that the UPC and/or UPC++ programmer has to pay attention to both data distribution and the use of communication whether it is implicit or explicit.

This leads to an increase in the *effort in programming* that UPC++ developers have to deploy in order to get satisfying performance on single-node, multi-node or many-core hardware architectures with UPC++ it is no longer possible to claim that this implementation of the PGAS programming model means a better *ease of programming* compared to the de-facto standard MPI+OpenMP. However, by offering a lot of features related to one-sided asynchronous communication, the use of C++ and *Promises* and

Threads (node)	UPC Heat Equation	UPC++ Heat Equation
16 (1)	123.04	123.03
32 (2)	64.99	64.00
64 (4)	31.05	33.00
128 (8)	15.60	14.20
256 (16)	7.88	8.90
512 (32)	3.99	3.85
1024 (64)	3.26	0.52

Table IV: Performance results in seconds for UPC++ Heat Equation 2D versus UPC Heat Equation 2D. Domain size `20000x20000`, 1000 iterations. Performance obtained on Abel super-computer (described in section 3 see page 13), computing on 16 to 1024 threads (1 to 64 nodes). For graphical representation see Figure 4.

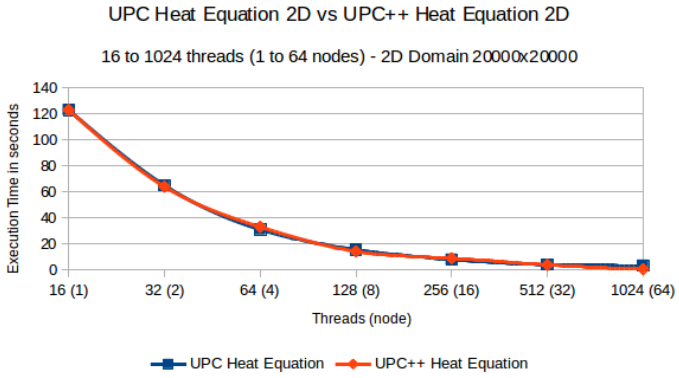


Figure 4: Multi-node performance for Heat Equation 2D, domain size 20000x20000, 1000 iterations using Abel Supercomputer (abel.uio.no). Using 1 to 64 nodes, 16 to 1024 threads. Performance are expressed in seconds (execution time) (lower is better). Table IV contains numerical results represented in this graph.

Futures in order to enforce the use of asynchronous programming, UPC++ represents an alternative to MPI+OpeMP. Moreover, UPC++ offers these functionalities and yields performance as we explain below and as we have shown in this paper.

Our goal was also to verify and show whether UPC++ can run and yield good performance on multi-node and many-core architectures and whether this performance is comparable to previously obtained performance in UPC. Thus, in Section 4 (page 15), we have presented the obtained performance for our implementations of UPC and UPC++ SpMV on both multi-node and many-core hardware architecture and we have presented the results for our implementations of UPC and UPC++ Heat Equation on multi-node. Globally UPC++ perform as well as UPC for both considered kernels. This is encouraging as a basis for future work using more advanced UPC++'s features such as asynchronous remote procedure calls and implementation of communication overlapping with computation.

6 Conclusion

In this study, we have investigated and provided insights into the UPC++ programming model and its performance. We have measured the computational performance of two kernels (SpMV and Heat Equation 2D) both on a multi-node supercomputer and a many-core processor. In addition, we have compared UPC++ to UPC by running identical kernels. We have shown that both on multi-node and many-core architecture UPC++ can compete with UPC and yields better performance on the largest amounts of cores and nodes that we have used (1024 cores, 64 nodes).

We have provided a detailed view of the difference of programming styles between UPC and UPC++ and the consequences in terms of code. UPC and UPC++, despite their similar names, are two extremely different implementations of the PGAS programming model, and in that sense, switching from UPC to UPC++ should not be considered lightly.

For future studies, there are many tracks to investigate, such as using more UPC++ features such as asynchronous remote procedure calls, or using UPC++ in conjunction with CUDA, and using UPC++ to implement different kind of kernels as we have focused our studies mainly on memory-bound and communication-bound problems.

References

[1] The Abel computer cluster. www.uio.no/english/services/it/research/hpc/abel/, 2018.

[2] G. Almasi. PGAS (partitioned global address space) languages. In D. Padua, editor, *Encyclopedia of Parallel Computing*, pages 1539–1545. Springer, 2011.

[3] J. Bachan, S. B. Baden, S. Hofmeyr, M. Jacquelin, A. Kamil, D. Bonachea, P. H. Hargrove, and H. Ahmed. UPC++": A High-Performance Communication Framework for Asynchronous Computation. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 963–973. IEEE, 2019.

[4] Berkeley UPC – Unified Parallel C. upc.lbl.gov, 2018.

[5] D. Bonachea and P. H. Hargrove. GASNet-EX: A High-Performance, Portable Communication Library for Exascale. In *Proceedings of Languages and Compilers for Parallel Computing (LCPC'18)*, volume 11882 of *Lecture Notes in Computer Science*. Springer International Publishing, October 2018.

[6] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and language specification. Technical report, Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, 1999.

[7] Colfax Research. MCDRAM as High-Bandwidth Memory (HBM) in Knights Landing Processors: Developer's Guide. https://colfaxresearch.com/knl-mcdram/, 2017. [Online; accessed 20-November-2019].

[8] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in Split-C. In *Supercomputing '93. Proceedings*, pages 262–273, Nov 1993.

[9] A. Davies and J. Mushtaq. The domain decomposition boundary element method on a network of transputers. *WIT Transactions on Modelling and Simulation*, 15, 1970.

[10] M. de Wael, S. Marr, B. de Fraine, T. van Cutsem, and W. de Meuter. Partitioned global address space languages. *ACM Computing Surveys*, 47(4), 2015.

[11] S. J. Deitz, B. L. Chamberlain, and M. B. Hribar. Chapel: Cascade High-Productivity Language An Overview of the Chapel Parallel Programming Model. *Cray User Group*, 2006.

[12] R. Grimes, D. Kincaid, and D. Young. ITPACK 2.0 User's Guide. Technical Report CNA-150, Center for Numerical Analysis, University of Texas, 1979.

[13] J. Lagravière, J. Langguth, M. Prugger, L. Einkemmer, P. H. Ha, and X. Cai. Performance Optimization and Modeling of Fine-Grained Irregular Communication in UPC. *Scientific Programming*, Article ID 6825728, 2019.

[14] J. Lagraviere, J. Langguth, M. Sourouri, P. H. Ha, and X. Cai. On the performance and energy efficiency of the pgas programming model on multicore architectures. In *2016 International Conference on High Performance Computing & Simulation (HPCS)*, pages 800–807. IEEE, 2016.

[15] J. Langguth, C. Jarvis, and X. Cai. Porting tissue-scale cardiac simulations to the knights landing platform. In *International Conference on High Performance Computing*, pages 376–388. Springer, 2017.

[16] D. A. Mallón, G. L. Taboada, C. Teijeiro, J. Touriño, B. B. Fraguela, A. Gómez, R. Doallo, and J. C. Mouriño. Performance Evaluation of MPI, UPC and OpenMP on Multicore Architectures. In M. Ropo, J. Westerholm, and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 174–184, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[17] S. M. Martin. *MATE, a Unified Model for Communication-Tolerant Scientific Applications*. PhD thesis, UC San Diego, 2018.

[18] R. W. Numrich and J. Reid. Co-Array Fortran for parallel programming. In *ACM Sigplan Fortran Forum*, volume 17, pages 1–31. ACM, 1998.

[19] PGAS - Partitioned Global Address Space. `http://www.pgas.org`, 2016.

[20] R. Rabenseifner. Introduction to Unified Parallel C (UPC) and Co-array Fortran (CAF), 2015. Short course at HLRS, University of Stuttgart, April 23-April 24.

[21] R. A. Tau Leng, J. Hsieh, V. Mashayekhi, and R. Rooholamini. An empirical study of hyper-threading in high performance computing clusters. *Linux HPC Revolution*, 45, 2002.

[22] UPC++. UPC++ Official Website. `bitbucket.org/berkeleylab/upcxx/`. Online; accessed 28-October-2019.

[23] UPC++. UPC++ Programmer's Guide, Revision 2019.9.0. `https://bitbucket.org/berkeleylab/upcxx/downloads/upcxx-guide-2019.9.0.pdf`. Online; accessed 28-October-2019.

[24] UPC++. UPC++ Version Changelog. `bitbucket.org/berkeleylab/upcxx/wiki/ChangeLog`. Online; accessed 28-October-2019.

[25] Wikipedia. Futures and promises — Wikipedia, The Free Encyclopedia. `en.wikipedia.org/wiki/Futures_and_promises`, 2019. [Online; accessed 28-October-2019].

[26] X10. Performance and Productivity at Scale. `x10-lang.org/`, 2019. [Online; accessed 11-December-2019].

[27] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, and K. Yelick. UPC++: A PGAS Extension for C++. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 1105–1114, May 2014.

Conflicts of interest: The authors declare that there is no conflict of interest regarding the publication of this paper.

Acknowledgements: This work was performed on hardware resources provided by UNINETT Sigma2 – the National Infrastructure for High Performance Computing and Data Storage in Norway, via Project NN2849K. The work was supported by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 671657, the European Union Seventh Framework Programme (grant No. 611183) and the Research Council of Norway (grants No. 231746/F20, No. 214113/F20 & No. 251186/F20).