UNIVERSITÀ DEGLI STUDI DI ROMA TOR VERGATA
DIPARTIMENTO DI INGEGNERIA CIVILE E INGEGNERIA INFORMATICA

Doctorate
Computer Science, Control and Geoinformation
Ciclo XXVIII

PARALLEL PROGRAMMING TECHNIQUES FOR HETEROGENEOUS EXASCALE COMPUTING PLATFORMS

Alessandro Fanfarillo

PhD Coordinator:
Prof. Giovanni Schiavon

Advisors:
Prof. Valeria Cardellini
Prof. Salvatore Filippone

Rome, March 2016 - Academic Year 2014/2015

To my parents and my uncle Adalberto.

Acknowledgements

My PhD period has been one of the most exciting and satisfying experiences of my life. Such positive adventure would not have been possible without the opportunities, support and guidance provided by Prof. Valeria Cardellini and Prof. Salvatore Filippone. To them all my gratitude and best wishes for the future.

A profound thank you goes to Dr. Damian Rouson, his enthusiasm, positive view, encouragement and tangible support have made a difference in my career.

My sincere thanks go also to Dr. Daniel Nagle, for being my mentor during the six months spent at the National Center for Atmospheric Research in Boulder (Colorado) and for trusting my potential since the very first day.

I thank my labmates at Tor Vergata, in particular Valerio Di Valerio, Matteo Nardelli, Francesco Bianchi and Luca Silvestri.

I thank Dr. Tobias Burnus and the members of the J3/WG5 Fortran standard committee for their encouragement and knowledgeable feedbacks about my work.

I also wish to acknowledge Amazon Web Services, CINECA and NERSC for the grants provided to several projects presented in this dissertation.

A special thanks to the city of Boulder, for being such an amazing place.

My deepest gratitude goes to my parents, for their unconditional love and support.

Abstract

Nowadays, the most powerful supercomputers in the world, needed for solving complex models and simulations of critical scientific problems, are able to perform tens of quadrillion (10^{15}) floating point operations per second (tens of PetaFLOPS). Although such big amount of computational power may seem enough, scientists and engineers always need to solve more accurate models, run broader simulations and analyze huge amount of data in less time. In particular, experiments that are currently impossible, dangerous, or too expensive to be realized, can be accurately simulated by solving complex predictive models on an exascale machine (10^{18} FLOPS). A few examples of studies where the exascale computing can make a difference are: reduction of the carbon footprint of the transportation sector, innovative designs for cost-effective renewable energy resources, efficiency and safety of nuclear energy, reverse engineering of the human brain, design, control and manufacture of advanced materials.

The importance of having an exascale supercomputer has been officially acknowledged on July 29th, 2015 by President Obama, who signed an executive order creating a National Strategic Computing Initiative calling for the accelerated development of an exascale system.

Unfortunately, building an exascale system with the technology we currently use on petascale machines would represent an unaffordable project. Although the cost of the processing units is so inexpensive as to be considered as free, the energy required for moving data (from memories to processors and across the network) and to power-on the entire system (including the cooling system) represents the real limit for reaching the exascale era. Therefore, deep changes in hardware architectures, programming models and parallel algorithms are needed in order to reduce energy requirements and increase compute power.

In this dissertation, we face the challanges related to data transfers on exascale architectures, proposing solutions in the field of heterogeneous architectures (CPUs + Accelerators), parallel programming models and parallel algorithms. In particular, we first explore the potential benefits brought by a hybrid CPUs+GPUs approach for sparse matrix computations, then we implement and analyze the performance of coar-

ray Fortran as parallel programming system for exascale computing. Finally, we merge the world of accelerators and coarray Fortran in order to create a data-aware parallel programming model, suitable for exascale computing.

The implementation of OpenCoarrays, the open-source communication library used by GNU Fortran for supporting coarrays, and its usage on heterogeneous devices, are the most relevant contributions presented in this dissertation.

Publications

Parts of the work presented in this thesis have been published in the following conference, journal and workshop papers.
Journals:

- A. Fanfarillo and D. Rouson. *Leveraging OpenCoarrays to Support Coarray Fortran on IBM Power8E*, ACM SIGPLAN Fortran Forum. Vol. 34. No. 2, May 2015.

Conferences and Workshops:

- V. Cardellini, A. Fanfarillo, S. Filippone. Heterogeneous CAF-based load balancing on Intel Xeon Phi, *6th International Workshop on Accelerators and Hybrid Exascale Systems (AsHES 2016)* (in conjunction with the 30th IEEE International Parallel & Distributed Processing Symposium), Chicago, Illinois, May 2016. Published in *Proceedings of the 30th IEEE International Parallel & Distributed Processing Symposium Workshops*, 2016.

- V. Cardellini, A. Fanfarillo, S. Filippone and D. Rouson. Hybrid coarrays: A PGAS feature for many-core architectures. In *Proceedings of the International Conference on Parallel Computing (ParCo2015)*, Edinburgh, UK, September 2015.

- A. Fanfarillo, T. Burnus, V. Cardellini, S. Filippone, D. Nagle and D. Rouson. OpenCoarrays: Open-source Transport Layers Supporting Coarray Fortran Compilers. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models (PGAS'14)*, Eugene, Oregon, USA. ACM, October 2014.

- A. Fanfarillo, T. Burnus, V. Cardellini, S. Filippone, D. Nagle and D. Rouson. Coarrays in GNU Fortran. In *Proceedings of the 23rd international conference on Parallel architectures and compilation (PACT14)*, Edmonton, Alberta, Canada. pp. 513-514 ACM, August 2014.

- V. Cardellini, A. Fanfarillo and S. Filippone. Sparse matrix computations on clusters with GPGPUs. In *Proceedings of 2014 International Conference on High Performance Computing & Simulation (HPCS'14)*, pp. 23-30. IEEE, July 2014.

- V. Cardellini, A. Fanfarillo and S. Filippone. Heterogeneous sparse matrix computations on hybrid GPU/CPU platforms. In *Proceedings of International Conference on Parallel Computing - ParCo2013*, Garching, Munich, Germany, September 2013.

Posters:

- A. Fanfarillo, T. Burnus, V. Cardellini, S. Filippone, D. Nagle and D. Rouson. Coarrays in GNU Fortran. PGAS booth poster, *The Intl. Conf. on High-Performance Computing, Networking, and Storage Analysis (SC14)*, New Orleans, Lousiana, Nov. 17-20.

- A. Fanfarillo, T. Burnus, V. Cardellini, S. Filippone, D. Nagle and D. Rouson. Coarrays in GNU Fortran. *ACM SRC at the 23rd international conference on Parallel architectures and compilation (PACT14)*, Edmonton, Alberta, Canada. August 2014.

Technical Reports:

- V. Cardellini, A. Fanfarillo, S. Filippone, "Overlapping Communication with Computation in MPI Applications", *Technical Report RR-16.09*, Dip. di Ingegneria Civile e Ingegneria Informatica, Università di Roma "Tor Vergata", Italy, Feb. 2016.
  https://art.torvergata.it/handle/2108/140530

- D. Barbieri, V. Cardellini, A. Fanfarillo, S. Filippone, "Three storage formats for sparse matrices on GPGPUs", *Technical Report RR-15.06*, Dip. di Ingegneria Civile e Ingegneria Informatica, Università di Roma "Tor Vergata", Italy, Feb. 2015.
  https://art.torvergata.it/handle/2108/113393

Submitted papers

Journals:

- D. Barbieri, V. Cardellini, A. Fanfarillo and S. Filippone. Sparse Matrix-Vector Multiplication on GPGPUs. Submitted to *ACM Transactions on Mathematical Software*, under second review.

Contents

List of Tables

List of Figures

List of Abbreviations

CAF Coarray Fortran

GPGPUs General-Purpose computation on Graphics Processing Units

GPU Graphic Processing Unit

HPC High Performance Computing

ILP Instruction-Level Parallelism

MIMD Multiple Instruction Multiple Data

MPI Message Passing Interface

PGAS Partitioned Global Address Space

RDMA Remote Direct Memory Access

SIMD Single Instruction Multiple Data

SPMD Single Program Multiple Data

SpMV Sparse matrix-vector multiplication

1 Introduction

Contents

1.1 Motivation

In order to reach ever increasing performance, computer architectures and programming techniques will change radically in the next future. Gordon Moore, in 1965, observed that the number of transistors of a typical processor chip doubles every 18-24 months. Surprisingly, such statement still holds after 50 years. Anyway, the growth in number of transistors inside the chip does not directly corretale to performance enhancement.

From 1970 to 2000/2005, the growth in number of transistors was accompanied by higher clock rates (due to better miniaturization technology). Such higher clock rates

had two effects: 1) higher performance perceived by the users; 2) higher heat generation. The first effect gave the users the idea that their applications would run faster on the next generation architecture without changing anything in the source code. The second effect stopped that illusion. In fact, in 2005, industries hit the "power wall" and decided to release a microprocessor with two *cores* running at lower frequency. Such strategy allowed to provide theoretical higher performance than the previous architecture and overcome the limits imposed by heat generation. The main issue brought by this new strategy was that applications had to use both cores, in parallel, in order to exploit all of the avaiable computational power. In other words, users must change (radically) the source code of their application if they want to exploit the computational power of the new architectures. This programming revolution has been postponed for a long time by improving the instruction-level parallelism (ILP) inside the processors. A remarkable effect of instruction-level parallelism was shown by the processor Intel Pentium M (Centrino) in 2003. Centrino came out with a lower frequency than a Pentium 4 processor but it provided higher/comparable performance due to the presence of the SSE2 SIMD instructions (vector instructions able to perform multiple arithmetical operations in parallel) and a longer pipeline. Instruction-level parallelism improves the thoughput by executing multiple instructions at the same time and attempts to relieve users from explicit parallel programming, but this has obvious limitations: 1) the pipeline cannot be elongated infinitely; 2) longer vector instructions are hard to exploit for the compiler. Such limitation are also known as the "instruction-level parallelism wall".

Multi-core and many-core devices represent the way for overcoming the power and ILP walls. Anyway, a third restriction limits performance: the so called "memory wall" [2]. Such wall points out the fact that the performance of memories is lower than the performance of microprocessors; in other words, the cores cannot be fed fast enough to exploit the computational power.

We currently live in the so called *petascale era*, which means that current supercomputers are able to compute order of 10^{15} floating point operations per second (FLOPS). It is possible to make a prediction of when we will enter in the exascale era ("exa" means 10^{18}) and what hardware will be available at that time. Currently, people

expect the first exascale machine around 2020 but the conditions to access such a huge amount of computational power will impact all the people involved in designing and using such machines.

1.1.1 Exascale and Its Consequences

In June 2015, the most powerful computer in the world (Tianhe-2) was able to achieve 33.863 petaflops using the LINPACK benchmark; its peak performance was estimated to be around 54.902 petaflops. In order to provide such amount of computational power, Tianhe-2 used 3,120,000 cores and burned 17.8 MW of power. From this data, we can estimate that Tianhe-2 burns 1 nJ (nanoJoule) of energy for a FLOP running the benchmark. If we keep the energy cost constant and try to make an estimation for an exascale machine we obtain 1 GW of power. Both DOE (Department of Energy) and DARPA (Defense Advanced Research Projects Agency) adopted 20MW as upper bound for a reasonable power consumption of an exascale system; thus, an exascale system should burn, at most, 20 pJ per FLOP.

In [3] Shalf et al. define some cost fuctions associated with an exascale system; this analysis points out clearly the major consequences and restrictions imposed by exascale systems.

Cost of Power Even with the least expensive power available in US, the cost for a supercomputer system, including the overheads of cooling and power distribution, will cost around $1M per Megawatt per year to operate the system.

Cost of a FLOP Floating Point Units (FPU) used to be the most expensive component in a system in terms of power and design cost. Nowadays, FPUs consume a small fraction of the area of a modern chip and a much smaller fraction of power consumption. As stated in [3], in 2011 a double-precision FMA (fused multiply add) consumed around 100 pJ/op. By contrast, reading the double precision operands from DRAM costed about 2000 pJ/op. By 2018, Shalf et al. [3] estimated that a floating point operation will consume around 40 pJ/op, but reading data from a classic DRAM will cost 1000 pJ/op, on non-local NUMA domains, and 70 pJ/op on local NUMA domains.

Generally speaking, the actual limit to scaling current memory architectures is the *off-chip memory bandwidth wall*: off-chip bandwidth grows with package pin density, which scales much more slowly than on-die transistor density [4].

Currently, we know that an exascale compute node will be probably equipped with a low capacity on-chip memory that provides high bandwidth and low power consumption. Such memory will be a multi channel DRAM (MCDRAM): a variant of the Hybrid Memory Cube technology [5, 6]. From a more recent estimation, one HMC uses about 10% (1000 pJ per double) of the energy per bit compared to traditional DIMM. Even though the performance and energy improvements are remarkable, the memory capacity of such a new RAM is still limited and not comparable to the traditional DRAM. We will expand this concept in Section 1.1.1.1.

Cost of Moving Data Memory interfaces and communication links on supercomputers are currently dominated by electrical/copper technology. In [7, 8] Miller observes that a conventional electrical line can be modeled as a RC circuit. From such a model, Miller shows that the natural bit rate capacity of the wire (bandwidth), for a constant input voltage, does not improve as we use a smaller wire[1]. It is possible to increase the bit rate by increasing the drive voltage to the wire but this also increases power consumption. From this observations, we can get the following considerations:

- Power consumption increases proportionally to bitrate. Higher bandwidth means higher power consumption.
- Power consumption is distance dependent (quadratically with wire length). Thus, we will probably have very localized high bandwidth networks on the exascale machines.
- Improvements in chip technology (smaller wires) will not improve energy efficiency or bandwidth.

A possible alternative to copper wires are optical fibers. Optical technology consumes about 30-60 pJ/bit whereas copper needs 10 pJ/bit for short distance transmis-

[1] Smaller means shorter and tighter. Keeping the cross-sectional area constant and using a shorter wire decreases power consumption.

sion. A good configuration would be using copper for short links and optical fiber for longer distance transmission. Anyway, without significant improvement in packaging and photonics technologies, it will not be possible to deal with a globally flat bandwidth across the system. Algorithms, software systems and applications will need to be aware of data locality.

1.1.1.1 Memory Subsystem, Cache and Data Movement

As stated in Section 1.1.1, DRAM memory alone will not be usable on exascale architecture because of the unaffordable energy costs required. Furthermore, JEDEC (the standard body that supported DDR memory) will not provide a standard for DDR-5. A valid alternative to this technology is to provide a hybrid memory system that integrates different types of memory with different sizes, performance and energy efficiency. As mentioned in Section 1.1.1, exascale computers will most likely have a very fast (in terms of bandwidth) and energy efficient multi-channel DRAM fused in the CPU chip. Unfortunately, the high bandwidth comes at a cost of lower capacity. Thus, the usual DRAM can be used as support of this fast, on-chip, memory. This memory hierarchy, based only on two levels, can be easily extended to three or more levels. It is possible, for example, to add a further level, more energy efficient than DDR-4, based on NVRAM (non-volatile memory).

The expansion of memory levels presented so far poses additional challenges to programmers. Although developers are quite familiar to optimize their codes in order to fit the problem size into the cache, they will face the same problem for each new memory level on the next architectures. In [1], the authors propose two main approaches to manage this new memory organization: 1) a physical address partitioning scheme in which the entire physical space is split into blocks allowing each memory pool to be individually addressed and 2) a system in which faster memory pools are used to cache slower levels in the memory system. In the former approach, the operating system can decide the location of a memory allocation by mapping the request to a specific physical address, using a virtual memory map or through the generation of pointer to a specific location. This approach allows for a series of specialized memory allocation routines to be provided to applications. The latter is quite self-explanatory:

high-bandwidth memories are used as cache for slower memories. Obviously, this requires a hardware caching mechanism to be added to the memory subsystem.

So far, programmers have used large shared caches to virtualize data movement between in-chip and off-chip memory. As the number of cores per socket increases, a cache coherence system will not be manageable because of the huge amount of off-chip data movement (which implies high energy costs and performance penalties). Programmers will be compelled to explicitly manage data movement; in the best case only regional coherence domains will be automatically managed across a subset of cores. There has been increasing interest in explicit software management of memory, such as the one exposed by the Kepler GPU or next generation Intel Xeon Phi, "Knights Landing" [9]. When data is placed into an explicitly software managed cache, it can be globally visible to other processors on the chip but it is not visible to the cache-coherence protocol (which requires a lot of data movement and communication overhead). Programming languages must adapt to this new model and enable the on-chip parallelism without a cache-coherent model. Even though it is unlikely that cache-coherence will be eliminated completely, the trade-offs between the size of the coherency domain and the magnitude of NUMA (Non-Uniform Memory Access) effects must be carefully considered.

1.1.1.2 Heterogeneous Processors

It is likely that processors on exascale computers will be composed of a collection of several types of processing elements. Figure 1.1 represents a possible exascale node architecture.

The *fat cores* could be the usual latency-oriented CPU cores we find in contemporary desktops or servers, equipped with multiple levels of cache, instruction-level parallelism, deep pipelines. The *thin cores* represent compute units with a less complex design, which require less power and space on the chip. Using a much higher number of smaller cores, a processor will be able to provide higher performance if a sufficiently high degree of parallelism is exposed by the algorithm. Generally speaking, processors of exascale computers will integrate, into the same chip, accelerators (like GPUs, Intel Xeon Phi and DSPs) and CPU cores. Programmers will need to consider

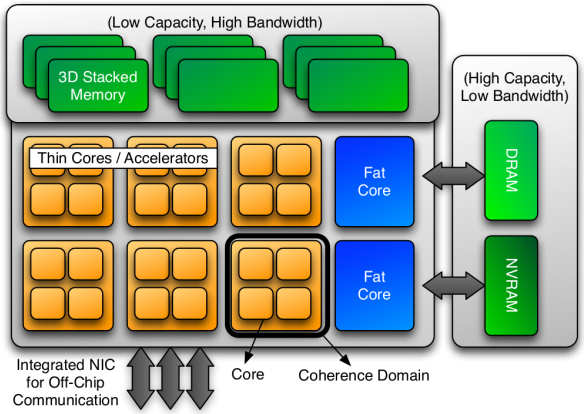


Figure 1.1: Model of exascale node architecture
(reprinted with permission from [1])

to use different classes of cores for different types of tasks: a fat core will provide the highest performance and energy efficiency for algorithms where little or no parallelism is available, while thin cores will provide the highest aggregate processor performance and energy efficiency where parallelism can be exploited. Furthermore, a combination of the two can also be employed: using fat cores and thin cores for the same algorithm but with a different workload distribution.

In 2014, the National Energy Research Scientific Computing Center (NERSC) announced that their next supercomputer, named Cori, will be a Cray system based on a next-generation Intel Many Integrated Core (MIC) architecture; this machine will be a self-hosted architecture, neither a co-processor nor an accelerator. In other words, the concept of accelerator as a separate co-processor will disappear in the foreseeable future; such a deep change in processor architecture will require to write much more hardware-aware code in order to exploit all the available computational power.

In Table 1.1 we report a comparison between an Intel Ivy-Bridge processor and an Intel Xeon Phi Knights-Landing (KNL), thus illustrating the change between the current and future generations of HPC platforms.

From Table 1.1 we are able to confirm several trends and characteristics presented

Features	Edison (Ivy-Bridge)	Cori (Knights-Landing)
Num. physical cores	12 cores per CPU	72 physical cores per CPU
Num. virtual cores	24 virtual cores per CPU	288 virtual cores per CPU
Processor frequency	2.4-3.2 GHz	Much slower than 1 GHz
Num. OPs per cycle	4 double precision	8 double precision
Memory per core	2.5 GB	less than 0.3 GB of fast memory per core and less than 2 GB of slow memory per core
Memory bandwidth	≈ 100 GB/s	Fast memory has $\approx 5\times$ DDR4

Table 1.1: Architectural changes (source: NERSC)
(Edison and Cori are the names of the systems installed or planned at NERSC)

so far:

- high number of physical cores;
- lower frequency;
- two levels of memory, where one is small and fast;
- less memory per core.

Although multi and many core processors represent the way for overcoming the power and ILP walls, having more and more cores within the same chip does not represent a definitive solution. In 1974 Dennard et al. [10] formulated a scaling law (related to MOSFETs) saying that as transistors get smaller their power density stays constant, so that the power use stays in proportion with the area. Since around 2005/2007, Dennard scaling appears to have broken down. The primary reason cited for the breakdown is that at small sizes, current leakage poses greater challenges, and also causes the chip to heat up, which creates a threat of thermal runaway and therefore further increases energy costs. The failure of Dennard's law and the validity of Moore's law will make impossible to power-on all the transistors simultaneously at the nominal voltage, while keeping the chip temperature in the safe operating range. People from the electronics industry refer to the amount of silicon that cannot be powered-on at the nominal

operating voltage, for a given thermal design power (TDP) constraint as *Dark Silicon*. According to recent studies, researchers from different groups have projected that, at 8 nm technology nodes, the amount of Dark Silicon may reach up to 50%-80% of the chip, depending on processor architecture, cooling technology and workload [11]. This fact introduces more challenges as well as more opportunities. When a lot of transistors are easily available (almost for free compared to the cost of energy) but power is very limited, circuit specialization may be the solution. As explained in [12] transistors can be "spent" in order to "buy" power efficiency. For example, a circuit might have many different special-purpose cores that perform one task very efficiently but are dark the rest of the time. In conclusion, in the next future energy constraints will lead to highly heterogeneous processors, equipped with several specialized circuits. Recently, there has been increasing interest in neuromophic computing: the use of electronics circuits (digital and analog) in order to mimic neuro-biological architectures present in the nervous system. It is possible that a portion of the future processors will be devoted to such chips in order to solve machine learning tasks (neural networks) with minimal power requirements.

1.1.1.3 Fault Detection and Tolerance

The presence of billions of hardware components and several levels of software stack will likely represent an increment in number of hardware and software failures. An exascale system, 1000 times faster than a petascale system, will fail, roughly (in the best case), 1000 times more frequently. Unfortunately, smaller transistors are much more error prone. Smaller circuits, carrying smaller charges, are much more subject to transient errors due to environment interferences. One of the major causes for transient errors is cosmic radiation: neutrons flux occasionally interacts with silicon creating a parasite cascade of charged particles. In [13] the authors analyze the effect of cosmic-rays on soft-error rate in computer logic devices in several locations across the United States. They show that computers located on top of mountains experience an order of magnitude higher rate of soft errors compared to sea level. The increase in chip density (predicted by Moore's law) will probably be a limiting factor in processors design because of the cosmic-rays effect. Smaller transistors and wires will age more

rapidly and more unevenly, so that permanent failures (physical hardware failures) will be more frequent. Although hardware vendors can face the increase of fault rates by using more powerful error detection and correction codes, researchers ([14]) estimate that the increase in error rate can be kept under control by using 20% more circuits and energy consumption.

One of the most used techniques for facing failures is checkpointing: save all the work periodically in order to restart from a recent backup after an eventual fault. Even though this technique is quite effective, it requires a lot of I/O operations that usually require tens of minutes. At limit, the risk is that the time for checkpointing is close to the Mean Time Between Failures; in this case, a lot of time is spent for saving the work and only a little for the actual computation. A valid alternative to checkpointing is to create fault tolerant applications. In this scenario, a parallel application can handle the error and execute some actions to terminate cleanly or follow some recovery procedures. The application has to be able to detect errors and access remote data to correct or compensate for the error and its effect. Obviously, such applications will pay this ability in terms of code complexity, energy consumption (mainly because of more frequent communication) and speed.

1.1.2 Parallel Programming Models for Exascale

The High Performance Computing (HPC) world has been dominated for years by the parallel distributed-memory paradigm, which allows to distribute the computation among several compute units, each of which has a private memory space. This approach has been successfully used in the last 30 years, when multi-core architectures were not commonly used. Programming a distributed memory system implies the usage of a mechanism to allow interprocess communication. In the early 80s, the dominant approach for inter-process communication on distributed systems was the message passing model. At that time, every machine/network producer provided its own message passing protocol to its users. In 1992, a group of researchers from academia and industry, decided to design a standardized and portable message-passing system called MPI, for Message Passing Interface. Since then, MPI has been the standard for com-

munication among processes on distributed memory systems and thus, the dominant programming system in HPC.

MPI provides a set of point-to-point and collective communication functions that allow the user to distribute the load among processes and manage the parallel computation. Such set consists of about 128 functions (MPI 1.3) which allow the user to implement very complicated and efficient parallel programs, based on the message-passing paradigm. The drawback of such a big (and powerful) function set is complexity. In order to be used, MPI functions require the user to know about low level details, such as the memory arrangement of multidimensional arrays (row-major vs. column-major order). Because of its high performance and complexity, MPI got the appellation of "the assembly language of parallel computing".

In scientific computing, the most used approach for parallel programming is usually based on the data-parallel model. This strategy consists in dividing the data to be analyzed (used for computation) among the processes and performing communication when needed in order to get the final result. This approach is also called the "Owner-Computes rule": each process performs all the computation involving the data it owns (same code executed on each process, applied on different data). After this computation phase, each process waits in a barrier for everyone to complete in order to exchange the results. People refer to this pattern as "bulk-synchronous" because it recalls the computation/communication phases of the Bulk Synchronous Parallel (BSP) model [15–17]. The data-parallel model and the bulk-synchronous[2] execution style provide very high performance when all the compute units are homogeneous (in terms of speed) and the data partitioning is well balanced. In fact, having all the compute units running at the same speed, on the same amount of data, leads to perfect parallelization and thus, high performance. If some processes fail to complete their portion of work in due time, a large amount of time may be wasted during the wait at the barrier.

Computing on an exascale machine will present many challenges. Because of what we have explained in Section 1.1.1, programs will need to deal with heterogeneous processors, high degree of parallelism and unpredictable behaviors due to node faults

[2] With the term "bulk-synchronous", we refer to the programming style based on separate computation and communication phases, even when the communication is implemented with two-sided message passing functions. It should not be confused with the BSP model.

and/or frequency/voltage throttling. The latter will be probably imposed by power management systems, based on energy optimization techniques like the near-threshold voltage (NTV) operation [18].

The high dynamicity exposed by the exascale machines will certainly break down the assumption of homogeneous components on which scientific applications relied upon so far. For all these reasons, new parallel programming models are required to handle these challenges.

Exascale nodes will be equipped with several, different devices; for example CPUs, Intel Xeon Phi, and DSPs. All these devices have a different parallel programming style and, probably, a different parallel programming language. Currently, the solution to tackle this problem seems to be a MPI+X approach. X could represent OpenMP, OpenACC, CUDA, PGAS, OpenCL or, more generally, what deals with data within an address space that spans multiple cores or even multiple nodes. Such an approach requires high interoperability from the parallel programming environment in order to mix pre-existing hardware and software technologies. Nowadays, OpenMP seems the best candidate for intra-node computation. In [19], the authors report the design choices that a high-performance parallel programming model faces; we report in details those considerations in the following paragraphs.

Scheduling The scheduling problem, critical task considered since the beginning of computer science, plays an even more relevant role in the exascale era. At the hardware level, the mapping between logical and physical threads and whether they can move to different compute units can significantly impact performance. Because of the heterogeneous nature of exascale machines, a pure static resource allocation should be avoided. Again, in the MPI+OpenMP approach (also known as hybrid model), logical threads are statically allocated to nodes but dynamically allocated to cores within nodes. At the software level, the presence of heterogeneous compute units requires an intelligent scheduling able to minimize, as much as possible, the execution time by balancing the amount of work to be assigned to the different units.

Communication As shown in Section 1.1.1, communication takes more time and energy than computation and represents one of the most important limitations for exascale computing. By communication, we mean not only communication across (and within) nodes, but also between memory levels. The latter is traditionally managed implicitly, by cache coherence protocols that ensure a consistent view of the memory to all cores. On an exascale node, equipped with hundreds of cores, it will be difficult to support cache coherence. Therefore, in-chip communication (core-to-core and core-to-memory) will become more software controlled and (perhaps not fully) managed by the user. Compilers should be able to optimize loads and stores by aggregating them or by perfoming them collectively. Communication across compute nodes is usually controlled by software and managed explicitly by the user; it can be one-sided, two-sided or collective. In the one-sided communication, *get* and *put* operations are performed on remote memory; in this case, only one process is aware that communication is taking place. The two-sided approach assumes that both communication locations are involved in the communication (as for send/recv message-passing operations). Collective operations are used for one-to-many or many-to-one communication; they have a crucial impact on application performance, because the lagging of a single node will impact the entire application. Luckily, collective operations can be efficiently implemented in hardware (on NIC) and thus some sort of overlap with computation can be achieved.

Synchronization Two-sided communication is implicitly synchronizing: the (usually blocking) send (or recv) posted by a process requires a correspondent (blocking) recv (or send) on the target process. One-sided communication is not synchronizing and thus a separate synchronization is required to guarantee the correct execution in case of data dependency. As mentioned at the beginning of this section, with a two-sided approach, a jitter in performance on one process impacts not only the performance of the process waiting for data, but also all the other processes related to the waiting process. Furthermore, when the numbers of cores and nodes increase, even small local delays caused by interrupts, operating system daemons, or events of cache and page misses (system noise) can affect global application performance. In [20–22],

the authors analyze this phenomena and propose models and tools for quantifying the effect of system noise. In light of what we have shown so far, one-sided communication should probably replace (although not completely) the two-sided approach on exascale machines. The explicit synchronization needed by one-sided communication should expose fine granularity, allowing the user to express applications in a "event-driven" way through non-blocking communication.

One-sided communication supports the PGAS (Partitioned Global Address Space) programming model well. In this model, data can be either private or exposed (and accessible) to remote processes. Access to private data happens like a local load/store, whereas accessing remote data requires RDMA (Remote Direct Memory Access). Modern communication hardware increasingly supports RDMA, to reduce the amount of copying and overhead that occur during inter-node communication. We describe the PGAS languages (and in particular coarray Fortran) in more details in Chapter 5.

Data Distribution Communication costs depend on where the data is stored when it is not actively used. Lowering this cost means to collocate properly the data and operations on this data. In [19], the authors report two approaches for facing this problem: *data-centric* and *control-centric*. The former represents the usual data parallelism: data distribution is performed first and then computation is assigned accordingly. The latter represents task parallelism: specific tasks are first distributed among the compute units and then data is moved where it is needed. Most parallel programming models encourage a data-centric view for distributed memory and a control-centric view for shared memory.

1.2 Problem Definition

On an exascale platform, data movement or, more generally, communication, represents the most expensive operation in terms of time and energy. At the same time, it represents one of the most critical factors that influences performance, because of the high level of parallelism involved. Finding a trade-off between reducing communication costs and providing high performance is a task that involves many different aspects

of a HPC platform. Computer and network architectures, parallel programming models and systems, parallel algorithms, scheduling and load balancing are only a few of them.

As practical example, let us consider one of the most important (and tough) problems in scientific computing: the fast solution of sparse linear systems. Most problems of mathematical physics need to solve huge sparse linear systems coming from the discretization of differential equations. Unlike dense matrix computations, usually characterized by regular memory access patterns and thus limited by floating-point throughput, sparse matrix computations suffer of memory bandwidth limitations due to much less regular access pattern. Sparse matrix computations have gotten remarkable benefits from the use of GPGPUs, mainly because of the adoption of efficient sparse representations, able to harness a large fraction of the available memory bandwidth (higher than CPUs) on such devices. Unfortunately, scaling from a single compute node to multiple nodes, on a cluster of GPGPUs (General-Purpose computing on Graphics Processing Units), can seriously degrade performance, mainly because of the network bottleneck, but also because of the inappropriate parallel programming model. In fact, improving the performance of the computational kernels increases the overall performance but "moves" the bottleneck of the parallel application from computation to communication. The classic bulk-synchronous approach, where computation and communication are performed as independent phases, is not going to work well in this scenario. Communication must be overlapped with computation in order to mask the performance penalty; it should also avoid implicit synchronization points, that would keep in an idle state the compute units of the processes involved in the communication.

As usually happens in computer science, changes in hardware, like the one shown in Tab. 1.1, have an impact on software and programming models. Algorithms and parallel programming models must adapt to the opportunities and restrictions imposed by the high degree of parallelism, heterogeneity and energy constraints.

If from one hand, high parallelism allows higher performance while respecting the energy restrictions, on the other hand, many slow cores penalize the performance of applications that cannot expose enough parallelism. For architectures depicted in Figures 1.1 and 1.2, heterogeneous computing can be the solution. In fact, applications

can use, at the same time, CPU cores for latency-oriented kernels, accelerators for throughput-oriented kernels and specialized hardware (like neuromorphic chips) for more complex tasks. Even in this scenario, where heterogeneous compute units coexist within the same silicon die, communication plays a crucial role. In fact, partitioning the application in a task-based fashion means dealing with a highly dynamic environment, where several small messages have to be exchanged across processes in order to progress the execution of the whole application (e.g. pipeline scheme).

When dealing with high parallelism, a small perturbation on a single core can have a significant impact on the whole execution flow. If the perturbation is represented by simple *system noise*, the effects translate immediately in a performance penalty whose magnitude depends on several factors. If the perturbation is represented by a core or node failure, the entire application collapses, unless the parallel programming system is equipped with a failure management mechanism. Implementing such a mechanism requires a non-negligible communication overhead that has to be carefully considered.

As we said, communication can be also expressed as data movement. Current software is based on the idea that computing is the most expensive component but, in the exascale era, computing will be cheap and massively parallel, while data movement will dominate performance and energy consumption. This architectural trend will break our existing programming paradigm because the current software tools are focused on equally partitioning computational work. In doing so, they implicitly assume all the processing elements are equidistant to each other and equidistant to their local memories within a node. Such a *compute-centric* approach no longer reflects the underlying machine architecture, where data locality and the underlying topology of the data-path between computing elements are crucial for performance and energy efficiency. A possible way to deal with this issue is to express, as much as possible, *data locality*. Parallel programming models and applications must embrace a *data-centric* approach that takes data layout and topology as the most important criteria for optimization. Such concepts can be seen, at the macroscopic level, in applications that use accelerators (in particular GPUs) installed on a platform like the one depicted in Figure 1.2. The high costs for transferring data between host and accelerators, induced by the PCIe bus, require to keep and use the data on the accelerators as long as possible.

In order to be effective, such a *data-centric* approach must be adopted not only by parallel programming models, but also by applications. Algorithms need to change and minimize communication (possibly avoiding communication) between processors and the memory hierarchy, by redefining the communication patterns.

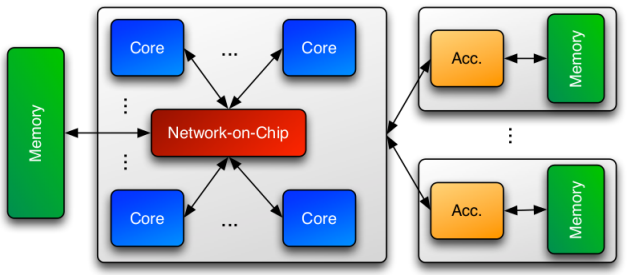


Figure 1.2: Current heterogeneous node with discrete accelerators

1.3 Contributions

The contribution of this thesis is to put together the world of accelerators and the PGAS model in order to tackle the challenges of exascale computing. In particular, this work analyzes the problems related to communication on exascale platforms and presents concrete solutions, considering several critical areas such as: heterogeneous computing, parallel programming models, data locality and load balancing.

So far, scientific applications in heterogeneous clusters relied mainly on a hybrid approach based on the combination of MPI two-sided functions, used for partitioning the work among heterogeneous nodes, and a parallel programming system tied to the accelerators (like CUDA, OpenCL, OpenMP). Even though this approach works quite well on current platforms, it will face several issues on the exascale machines. The contributions of this dissertation are twofold: from a "production" point of view, they provide an alternative approach for implementing scientific applications on heterogeneous clusters, using a parallel programming model suitable for exascale computing. From a research point of view, they allow one to study the behavior and possibilities

brought by coarray Fortran on/with accelerators.

Our contributions cover three main topics: the first comprises sparse matrix-vector computations on heterogeneous clusters of GPGPUs. The second deals with the potential of PGAS languages on exascale platforms and presents OpenCoarrays: the transport layer used by the GNU Fortran compiler for the coarrays support. Finally, we propose a way to merge the first two topics and present some examples on how to use effectively PGAS languages on heterogeneous platforms. In particular, we focus on how to implement dynamic load balancing algorithms and how to use effectively the heterogeneous resources.

From a high-level prospective, this work provides the following contributions:

- we analyze the hybrid CPU+GPU approach for the sparse matrix-vector multiplication (SpMV) kernel and propose load balancing algorithms based on regression models of the heterogeneous devices;
- we report our experience of SpMV on clusters of GPGPUs;
- we investigate the usage of PGAS languages, in particular coarray Fortran, as a parallel programming model for exascale platforms;
- we design, implement and analyze OpenCoarrays; the free coarray Fortran transport layer used by the GNU Fortran compiler. OpenCoarrays has been realized by myself during a six months visiting period at National Center for Atmospheric Research in Boulder, Colorado.
- we propose a new keyword for the Fortran language, useful for expressing data locality on exascale platforms and suitable for both, accelerators and coarrays;
- we demonstrate how coarray Fortran and heterogeneous architectures can be used together effectively.

1.4 Thesis Outline

This thesis is split in two major parts:

Part I. The first part comprises the contributions related to the heterogeneous sparse matrix computations, on a single node and on clusters. Chapter 2 focuses on the related work in this area, describing in details the state-of-the-art of sparse matrix computations on GPUs and hybrid CPUs+GPUs. It also shows the architectural details of a Nvidia GPU, the possible configurations and the alternatives for data exchange with CPU and cluster nodes. Then we describe in Chapter 3 our implementation of hybrid CPU+GPU sparse matrix-vector product, using different sparse matrix formats on CPU and GPU, and propose several load balancing algorithms, based on regression models of the compute units and PCIe bus. Finally, in Chapter 4 we analyze several alternatives for data transfers on GPGPUs clusters.

Part II. The second part comprises the contributions related to PGAS languages, in particular coarray Fortran. Chapter 5 describes the related work in this area, what a PGAS language is and why coarray Fortran can be suitable for the exascale era. We also consider the issues related to the implementation of a PGAS language on top of MPI. In Chapter 6, we present OpenCoarrays, the coarray transport layer used by the GNU Fortran compiler, accompanied with an exhaustive performance comparison with the coarray implementation provided by two commercial compilers, Intel and Cray. In Chapter 7, we propose a new keyword for the Fortran language, capable of expressing data locality and unifying coarray Fortran with accelerators. Although, the keyword has been applied to CUDA GPUs, it has been thought for a more general purpose. We also introduce the Intel Xeon Phi architecture and show some applications using coarray Fortran for heterogeneous computing. Finally, Chapter 8 concludes this thesis and outlines some future work.

Part I

Sparse Matrix Computations on GPGPUs

2

Background and Related Work

Contents

This chapter presents the background and the state-of-the-art of sparse matrix computation on GPGPUs, with a particular focus on sparse matrix-vector multiplication (SpMV). General descriptions of GPGPU architecture, memory model and issues related to cluster environment are also provided.

2.1 Sparse Matrix Computations

Sparse matrices and related computations are one of the centerpieces of scientific computing. Most problems of mathematical physics require the approximate solution of differential equations; to this end, the equations have to be transformed into algebraic equations, or *discretized*. A general feature of most discretization methods, including finite differences, finite elements, and finite volumes [23–25], is that the number of entries in each discretized equation depends on local topological features of the discretization, and not on the global domain size. Thus, many problems deal with sparse matrices; far from being unusual, sparse matrices are extremely common in scientific computing, and the related techniques are extremely important. There is not a unique, exact, definition of what a sparse matrix is but the most famous is the pragmatic definition given by J.H. Wilkinson [26]:

> Matrix $A \in \mathbb{R}^{m \times n}$ is said to be sparse if we can exploit the fact that a part of its entries is equal to zero.

A more rigorous definition may be stated by (implicitly) referring to a class parametrized by the dimension n:

> A matrix $A \in \mathbb{R}^{n \times n}$ is sparse if the number of nonzero entries is $O(n)$.

This means that the average number of nonzero elements per row (per column) is bounded, independently of the number of rows (columns).

The mathematical models based on the discretization of Partial Differential Equations (PDEs) require the solution of linear systems; such solution can be found using two methods: direct or iterative methods. Direct methods are the most common, robust and predictable; in fact, it is possible to know a-priori how many "steps" the algorithm will take in order to return the solution (if it exists). Direct methods are based on the idea of reducing the matrix of a given system to a form that can be solved by backward substitution (e.g. Gaussian elimination). On the other hand, iterative methods approach the solution by generating a sequence of approximations. On sparse systems, most of the time, iterative methods have been shown to get to the solution much faster and with an higher accuracy than direct methods. The iterative methods used today for solving

large, sparse, linear systems are mostly preconditioned Krylov subspace solvers. A *preconditioner* is nothing but a transformation matrix, carefully built in order to transform the coefficient matrix into an equivalent one with a more favorable spectrum; such transformation leads to a faster convergence rate during the iteration. The detailed discussion of Krylov methods is beyond the scope of this work; however, all Krylov methods employ the coefficient matrix A only to perform matrix-vector products $y \leftarrow Ax$. This simple and general fact explains why the Sparse Matrix-Vector multiplication (SpMV) is so important: the faster we perform this simple operation, the sooner we get the solution. Because of the sparse nature of the structures involved, and the linearity of the computation, SpMV is a well-known memory bounded problem. The performance of SpMV is strongly related to the coefficient matrix (sparse pattern) and the computing platform. Its efficient implementation is mostly based on picking the right data structure (sparse representation) for the computer architecture where the iterative solver will be run.

2.2 Storage Formats for Sparse Matrices

Wilkinson's definition given in the previous section suggests that we *must* take advantage of the fact that the sparse matrix has many non zero entries. In other words, with sparse matrices, it is not a good idea to use the explicit storage used for dense matrices. Even though, the transformation from explicit to sparse format is beneficial to memory usage, the non-contiguos data storage impacts the performance. In fact, the performance of sparse matrix kernels is typically much less than that of their dense counterparts, precisely because of the need to retrieve index information and the associated memory traffic. Moreover, whereas normal storage formats allow for sequential and/or blocked accesses to memory in the input and output vectors x and y, sparse storage means that coefficients stored in adjacent positions in the sparse matrix may operate on vector entries that are quite far apart, depending on the *pattern* of nonzeros contained in the matrix.

It is clear that the performance of sparse matrix computations depends on the specific representation chosen for a specific architecture. The following list summarizes

the factors that contribute to overall performance:

- the match between the data structure and the underlying computing architecture, including the possibility of exploiting special hardware instructions;
- the suitability of the data structure to decomposition into independent, load-balanced work units;
- the amount of overhead due to the explicit storage of indices;
- the amount of padding with explicit zeros that may be necessary;
- the interaction between the data structure and the distribution of nonzeros (pattern) within the sparse matrix;
- the relation between the sparsity pattern and the sequence of memory accesses, especially into the x vector.

Many storage formats have been invented over the years but the most widely used and "general purpose" are: COOrdinate (COO), Compressed Sparse Rows (CSR), and Compressed Sparse Columns (CSC). COO is the simplest sparse format: it use three vectors, one for the values, one for the row indices and one for the column indices. Each value of the matrix is accompanied with the row and column number where it appears. It is clear that if the sparse matrix has several elements per row, this format wastes space repeating the same row number for each element. The CSR format addresses this issue by compressing the rows in such a way that the row indices vector becomes a row pointer vector.

Even though these formats behave quite well on most computing platforms with little changes, running a sparse matrix computation on a special architecture (like a GPU) requires different formats, in order to exploit the specific hardware capabilities.

2.3 GPU Architecture and Memory Model

General Purpose Graphics Processing Units (GPGPUs) [27] are currently an established and attractive choice in the world of scientific computing, found in many among

the fastest supercomputers on the Top 500 list. The GPGPU cards produced by Nvidia are currently among the most popular computing platforms; their architectural model is based on a scalable array of multi-threaded streaming multi-processors, each composed by a fixed number of scalar processors, a set of dual-issue instruction fetch units, one on-chip fast memory partitioned into shared memory and L1 cache plus additional special-function hardware. For such Nvidia GPUs, the programming model of choice is CUDA [28–30] (Compute Unified Device Architecture). A CUDA program consists of a host program that runs on the CPU host, and a kernel program that executes on the GPU itself. The host program typically sets up the data and transfers it to and from the GPU/GPGPU, while the kernel program performs the main processing tasks.

In this section, we first discuss some CPU+GPU architectures and future trends, then we explain in details how the CUDA Unified Virtual Address (UVA) mechanism works and how it differs from the new CUDA Unified Memory Access provided by CUDA 6.0. Finally, we briefly describe the internal components of a GPU and how they influence the programming model.

In this section, the reader will notice how the evolution of heterogeneous systems is pointing towards a full integration of CPU cores and accelerators in the same device.

2.3.1 CPU/GPU Configurations

Nowadays, the most common configuration for an heterogeneous node is composed by a discrete (and usually very powerful) GPU connect to the CPUs through a PCI Express bus. This usual (and currently obsolete) configuration is depicted in Fig. 2.1. In this case, the external GPU interacts with the CPU through the PCI Express bus connected to the Northbridge chip, which also contains the memory controller. The PCI Express bus (sometimes called PCIe) can theoretically provide about 500MB/s on each lane. Devices usually provide 1,4,8 or 16 lanes. Since GPUs require the highest bandwidth possible, they are usually plugged into a 16-lane PCIe slot. To sum up, a modern GPU can rely on a bandwidth of about 6 GB/s (including overhead) during the transfers from/to CPU.

Starting with the Intel Nehalem architecture (or AMD Opteron), the memory con-

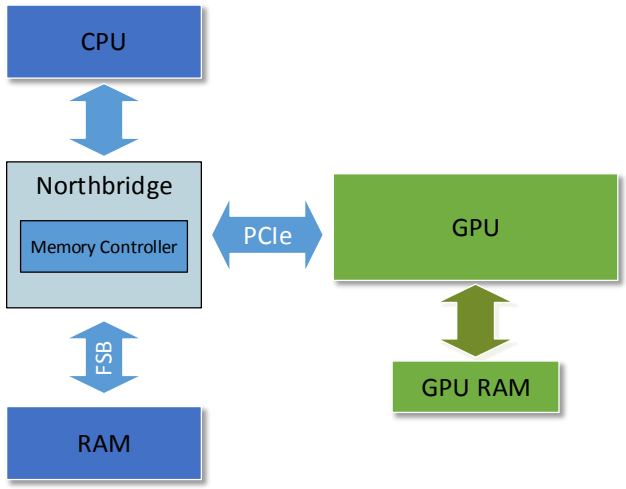


Figure 2.1: CPU/GPU architecture with external memory controller

troller was integrated inside the CPU; this architectural change significantly improved CPU memory performance. Fig. 2.2 shows the new architectural configuration; in this case, the connection between PCIe and CPU relies on an I/O Hub.

A further intergration step was proposed by Intel in its Sandy Bridge class of processors. In this architecture, the I/O Hub, which includes part of the PCIe, was integrated into the CPU. This improvement provided up to 40 PCIe lanes available, but since a GPU can only use up to 16 lanes, the change provided full bandwidth to more than 2 full GPUs connected on the same host.

Fig 2.3 depicts the latest step towards the full integration of CPU and GPUs: the AMD Accelerated Processing Unit (APU) chip. This configuration has huge potential in terms of heterogeneous computing, due to the heterogeneous Unified Memory Access (hUMA) technology, which exposes a unified virtual address space, maintaining cache coherence between the two. Even though we focus mainly on CUDA architectures, the description of the AMD APU architecture is critical for a wide view of what

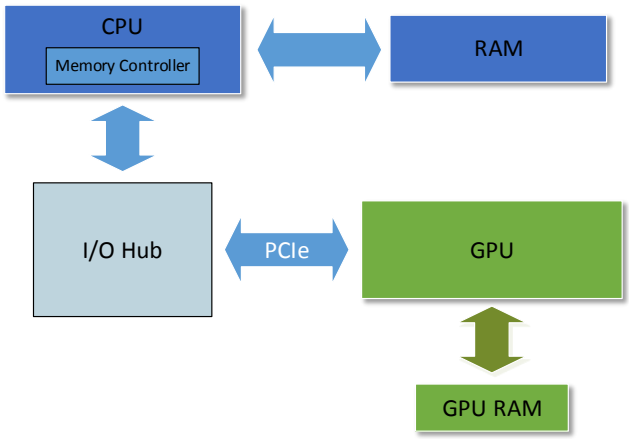


Figure 2.2: CPU/GPU architecture with integrated memory controller

is happening in the world of heterogeneous architectures. Nvidia has also proposed integrated GPUs for laptops and netbooks (like the MCP79 and MCP89). For transfer intensive workloads, such an architecture can outperform a much more powerful discrete GPU. On the other hand, sharing the same chip limits the amount of functionality that can be exposed.

2.3.2 CUDA Virtual Address Space

At the very beginning of the heterogeneous architectures based on CUDA, the memory address spaces of CPU and GPU were completely separate. This required an explicit memory allocation on the GPU and explicit data transfer from/to the GPU. As a regular CPU, the GPU uses a virtual address space in order to protect programs (kernels) from out-of-bounds memory accesses. CUDA 2.2 added a feature called *mapped pinned memory*. Such memory is the page-locked host memory mapped into the CUDA address space. The page tables of CPU and GPU point to the same memory region on the host but the virtual address is different for the CPU and the GPU. Mapped memory is

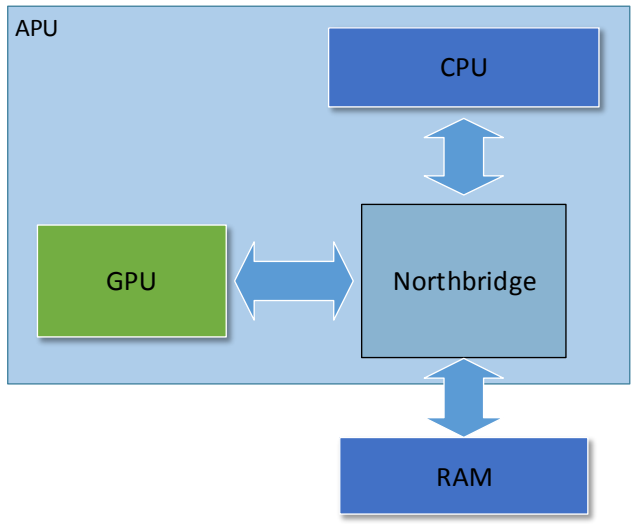


Figure 2.3: AMD's APU architecture

also know as *zero-copy memory*; the name expresses the concept that no explicit memory transfer between host and device needs to be initiated. A transfer across the PCIe will be initiated by the CUDA driver during the first time the mapped memory region is accessed. The implicit transfer will stall kernel execution until it terminates. The name *page-locked* stresses the fact that the memory cannot be swapped out as a usual memory page. Such a restriction allows to use a Direct Memory Access mechanism during the transfer on the PCIe bus. In fact, when regular memory is used, upon a transfer request from host to device, the Nvidia driver allocates a buffer as page-locked memory, copies the data from regular memory to the buffer and then performs the transfer (and frees the page-locked buffer). Such double copy can be completly avoided when the data on the host are stored in a page-locked memory region.

2.3.2.1 Unified Virtual Addressing

CUDA 4.0 added a feature called *unified virtual addressing*. This feature allows CUDA to allocate memory for both, the CPU and GPUs, from the same virtual address space. Technically, the CUDA driver performs large virtual allocations within the CPU address space during the initialization routine. Then, GPUs allocations are mapped onto that memory region when needed. For mapped pinned allocations, the GPU and CPU pointers are the same.

2.3.2.2 Unified Memory

CUDA 6.0 introduced the *managed memory* which is essentially memory allocated on both the CPU and GPU, controlled by the Nvidia driver. Managed memory implements the concept of Unified Memory (also known as Unified Memory Access or UMA): both the host and the device have the same address space and all transfers are implicit and managed by the driver. This feature sounds like the zero-copy memory described in Section 2.3.2, but they differ in when the transfer is triggered. Zero-copy starts the transfer when a memory region is accessed, whereas with Unified Memory, the transfer begins immediately before the launch and right after the termination of a kernel. Nvidia claims that the benefit brought by Unified Memory is twofold: 1) simpler programming and memory model; 2) higher performance through data locality. Allocating managed memory requires only a single invocation to the *cudaMallocManaged()* function and all the memory can be used as if it were regular host memory. Since Unified Memory migrates data on demand between the CPU and GPU, it can offer the performance of local data on the GPU (when the data have not been moved), while providing the ease of use of globally shared data. In [31], Landaverde et al. investigate the performance of the usual zero-copy memory approach with the new Unified Memory; they conclude that the performance improvement claimed by Nvidia about Unified Memory is strongly related to the problem pattern. They also report several cases where Unified Memory is worse than Zero-copy memory. In [32], we observed the same behavior as that reported by Landaverde et al. and we also noticed that Unified Memory does not work correctly when used as MPI buffer for RDMA access.

2.3.3 Nvidia GPU Architecture

The NVIDIA GPGPU architectural model is based on a scalable array of multi-threaded, streaming multi-processors, each composed of a fixed number of scalar processors, one or more instruction fetch units, and an on-chip fast memory with a configurable partitioning between shared memory and L1 cache, plus additional special-function hardware.

The computation is carried on by threads grouped into blocks. More than one block can execute on the same multiprocessor, and each block executes concurrently. During the invocation (also called *grid*) of a kernel, the host program defines the execution configuration, that is:

- how many blocks of threads should be executed;
- the number of threads per block.

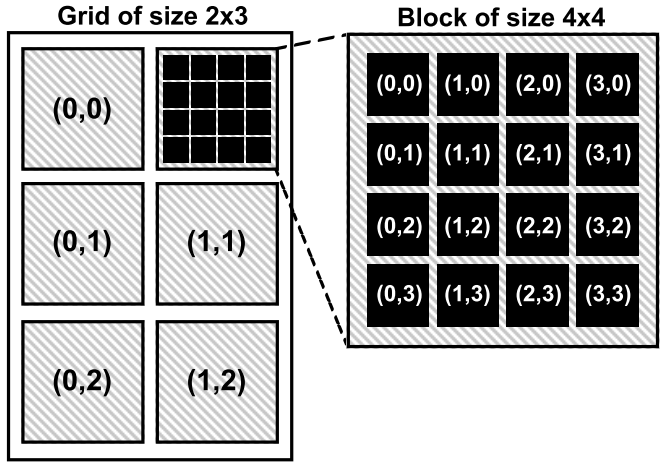


Figure 2.4: A 2D grid of threads

Each thread has an identifier within the block and an identifier of its block within the grid (see Figure 2.4). All threads share the same entry point in the kernel; the thread

ID can then be used to specialize the thread action and coordinate with that of the other threads.

Figures 2.5 and 2.6 describe the underlying Single-Instruction Multiple-Threads (SIMT) architecture. As shown in Figure 2.5, a single host may coexist with multiple

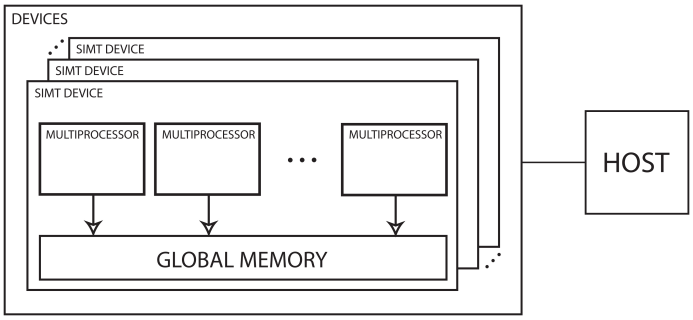


Figure 2.5: SIMT model: host and devices

devices. Each GPU device is made up by an array of multiprocessors and a global memory, divided in channels (or partitions). Memory requests to the same channel are enqueued and each channel provides only a fraction of the whole bandwidth; therefore, to exploit the full device bandwidth, the grid should access all channels.

Multiprocessors execute only vector instructions; a vector instruction specifies the execution on a set of threads (called *warp*) with contiguous identifiers inside the block. The warp size is a characteristic constant of the architecture; its value is currently 32 for NVIDIA GPUs. If threads within a warp execute different branches, the warp will issue a sequence of instructions covering all different control flows, and mask execution on the various threads according to their paths; this phenomenon is called *thread divergence*.

Each grid is executed on a single device; each thread block is enqueued and then scheduled on a multi-processor with enough available resources (in terms of registers, shared memory, and block slots) and retains all its resources until completion. A warp instruction is issued by a *scheduler* on an available vector unit that supports the relevant class of instructions. If a warp scheduler has more than one *dispatcher*, multiple inde-

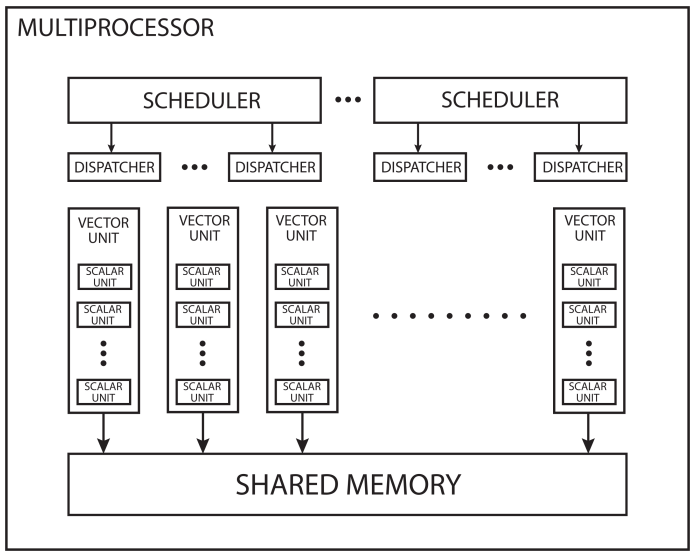


Figure 2.6: SIMT model: a multi-processor

pendent instructions from the same warp can be issued at the same time. Therefore, the throughput of an instruction class depends on the number of schedulers and dispatchers, the number of vector units that supports it, the number of scalar units inside the vector units, and the warp size. Threads belonging to the same thread block share data using the *shared memory* and synchronize their execution waiting on a *barrier*.

To fully exploit the available bandwidth of both shared memory and global memory, some access rules should be followed by the threads of a warp (and, in some cases, of a half warp, i.e., a group of 16 consecutive threads for NVIDIA's GPUs). Shared memory, for example, is divided into banks, each one providing a constant throughput in terms of bytes per clock cycle. For each different architecture there is a specification on the correct access pattern that allows to avoid bank conflicts. Ideally, threads with increasing identifier in the same warp should read sequential elements of either 4 or 8 bytes in memory. Similarly, different access patterns should be followed according to the target GPGPU architecture to exploit the full global memory bandwidth. When

these criteria are met, accesses are called *coalesced*. On Nvidia hardware, a portable pattern which provides coalesced accesses is the following: each thread with index k within the warp ($0 \leq k < warpSize$) should access the element of size D (with D equal to 4, 8 or 16 bytes) at address $D \cdot (Offset \cdot warpSize + k)$.

Performance optimization strategies also reflect the different policies adopted by CPU and GPGPU architectures to hide memory access latency. The GPGPU does not make use of large cache memories, but it rather exploits the concurrency of thousands of threads, whose resources are fully allocated and whose instructions are ready to be dispatched on a multiprocessor.

The main optimization issue to support a GPGPU target then revolves around how an algorithm should be implemented to take advantage of the full throughput of the device. To make good use of the memory access features of the architecture, we need to maximize the regularity of memory accesses to ensure *coalesced* accesses.

2.4 SpMV on GPUs

A significant amount of research has been devoted in the last years to improving the performance of the SpMV kernel on accelerators (in particular GPUs). As we said in the previous section, running a sparse matrix-vector multiplication on a GPU requires a careful selection of the sparse format based on the sparsity pattern and the architecture capabilities. Bell and Garland [33] investigate several well known sparse formats on GPUs including variants of CSR; such formats suffer of load imbalance and non-coalescent memory accesses. They overcome this limitation by presenting a hybrid format (HYB) which combines the strengths of the ELL and COO formats. Choi et al. [34] and Monakov et al. [35] focus their work on new sparse formats for GPUs with a block structure and propose an auto-tuning approach for choosing the right block size parameters. Dang and Schmidt [36] propose a new format called Sliced COO (SCOO) and an efficient CUDA implementation to perform SpMV on the GPU using atomic operations. Atomic operations have been improved in the Nvidia Kepler architecture; this means that, even the same sparse format, on the same sparsity pattern, can show a totally different behavior on a newer architecture because of the new hardware ca-

pabilities. Pichel et al. [37] show the effective advantages obtained by optimizing the SpMV on GPU using reordering techniques on several sparse formats.

2.5 Hybrid GPUs/CPUs Computations

As stated in Section 1.1.1.2, the next generation machines will be equipped with CPUs and accelerators fused in a unique chip. This fact has motivated a substantial amount of research on heterogeneous solutions, where CPUs and accelerators work together for the same purpose, exploting their unique features and strengths. Mittal and Vetter [38] recently presented a survey on CPU-GPU heterogeneous computing techniques where they give the following motivations to heterogeneous computing:

- *Acknowledging and leveraging the unique architectural strengths of processing units*: because of the latency-oriented and throughput-oriented natures of CPU and GPU, respectively, an heterogeneous system can provide high performance for a much wider variety of applications than using a CPU or GPU alone.
- *Matching algorithmic requirements to features of processing units*: in some cases, where data transfers dominate execution time, or branch divergence impacts the execution on all GPU cores, CPUs can provide better performance than GPUs. Different phases of an application may be more suitable for execution on a particular processing unit.
- *Improving resource utilization*: In order to meet the worst-case performance requirements, homogeneous systems are usually over-sized, even though their utilization remains low. Furthermore, during the execution of a GPU kernel, the CPU stays idle with a consequent waste of energy. Using an intelligent resource management, a heterogeneous system can exploit all the processing units in the right way.
- *Reaping the fruits of advancements in CPU design*: the latest performance improvement achieved on modern CPU might prove useless if the CPU is still considered as a simple host for the GPU. In [39, 40], the authors show how, by ap-

plying careful optimizations on both CPU and GPU, CPU can be as fast or even faster than a GPU. Due to this, choosing the right amount of work to delegate to the CPU or the GPU is a critical task in order to achieve high performance.

It is easy to see that such a heterogeneous approach adds further complexity to the already challenging SpMV on GPU. In [41], we analyzed this problem (described in more details in chapter 3) and found the following issues to consider when dealing with a hybrid GPU/CPU approach for SpMV:

- sparse matrix format selection on CPU and GPU;
- accuracy of the floating-point operations executed using different instructions (FMAD - Fused Multiply Add);
- data partitioning/load balancing among heterogeneous compute units.

The first issue has already been analyzed in Section 2.4; in this case, it just needs to be repeated for the CPU. The second issue is related to the possible numerical effect of merging the results of the same code, executed on different hardware, using different floating-point hardware instructions (like the Fused Multiply-Add). Applying a hybrid approach on numerically sensitive models/algorithms might lead to results mismatch with respect to an equivalent homogenous approach or, in the worst cases, to numerical instability. The last point is the most critical for an heterogeneous approach. Because of the heterogeneous nature of the compute units, they express some features better/-worse than others. For SpMV, GPUs are usually much faster than a multi-core CPU; thus, the amount of work to send to GPUs has to be greater than the work to send to CPUs. In [41], we analyze the performance of the various heterogeneous compute units and build a model in order to get the best load balance. “Best” load balance, in this case, represents the partitioning that allows the GPU and CPU to finish the computation at the same time. We will provide more details in Chapter 3. Indarapu et al. [42] present a similar approach based on work division schemes that aim at matching the right workload for the right device. Yang et al. [43] have recently presented a partitioning strategy of sparse matrices based on probabilistic modeling of nonzeros in a row.

The advantages of the proposed strategy lie in its generality and good adaptability for different types of sparse matrices. They also developed a GPGPU/CPU hybrid parallel computing model for SpMV in a heterogeneous computing platform. Matam et al. [44] present a workload division technique for generalized sparse matrix-matrix multiplication (SPGEMM). This task is remarkably hard to accomplish due to the highly irregular computation in SPGEMM and, because of such irregularity, a GPU does not yield a large speedup.

2.6 Cluster of GPGPUs

For what we presented so far, GPUs (and more generally accelerators) appear to be very powerful devices. It is natural to expect that we could get higher performance adding more GPUs. The Scalable Link Interface (SLI) provided by Nvidia allows one to connect up to four GPU inside the same chassis (controlled by the same host). In order to go beyond this limit, HPC people adopted what they learned from the well known MPI+OpenMP hybrid approach: divide the work using MPI processes and perform the computation using OpenMP threads. This approach can be really effective when the communication overhead introduced by the distributed memory approach (MPI) becomes a limiting factor for scalability and performance. On a cluster of GPUs, where there might be more than one GPU per node (up to four), the most common approach is to allocate as many MPI processes per node as GPUs installed. Each process is logically connected to a single GPU and acts as host CPU/GPU and inter-node communication manager. Since MPI is designed to transfer data among host buffers, the user is supposed to manage the CPU/GPU transfer when data have to be sent to another process. As mentioned by Stuart et al. [45], the MPI+CUDA approach is so widely employed that most MPI implementations provide a CUDA-aware support. A MPI CUDA-aware implementation is thus capable of accessing device buffers directly, making the programming effort easier and more efficient in terms of performance. Since 2010, Nvidia has introduced a set of technologies called *GPUDirect* that, on devices with compute capability higher than version 2.0, provide:

- Faster communication with network and storage devices.
- Peer-to-peer transfers between GPUs on the same PCIe buffer without using host memory.
- Remote DMA (RDMA): a GPU buffer can be copied directly over a network to a remote GPU.

The last capability is the most important for a SpMV on GPGPUs cluster because of the highly irregular communication pattern. *GPUDirect* should not be confused with UVA; the latter makes the code simpler by allowing the memory pointer to be used on both the CPU and GPU without any translation. The former, on the other hand, is a technology which is completely transparent to the programmer.

2.6.1 SpMV Issues on GPGPU Clusters

The most important problem when using GPUs on sparse matrix computations is the large overhead imposed by the PCIe bus which connects the CPU to the GPU, whose bandwidth typically becomes the performance bottleneck. In fact, for each iteration of our iterative solver, every process will exchange data with its neighbors in order to complete the computation. This means that, for every iteration, a transfer from GPU to CPU is required in order to proceed to the next step. When performing a copy from the host to the GPU, the CUDA driver uses Direct Memory Access (DMA). This operation causes a double copy: the first from the pageable system buffer to a temporary page-locked buffer, and the second from the page-locked buffer to the GPU. Thus, the copy speed is bounded by the slowest of the PCIe and the system front-side buses. Furthermore, a pageable memory copy involves the CPU, adding further overhead. CUDA provides special functions to allocate host-locked memory, also called *pinned memory*: the operating system guarantees that it will not be paged out [46]. A copy with pinned memory does not need the double access step; moreover, it enables direct use of the host memory inside the CUDA kernels, a working mode called *zero-copy*. The essential communication step is a "halo exchange" [47]. Every sparse matrix can be viewed as a graph representation; for matrices arising from the PDE discretization, this graph

has a natural isomorphism with that describing the topology of the discretized computational domain. When a domain is partitioned into subdomains, each one assigned to a process, the nodes of the graph lying at the boundary of a subdomain are involved in data exchange with the adjacent nodes lying just across the boundary; those adjacent nodes from other subdomains are the "halo" of a given domain, and they correspond to the data items to be exchanged. Note that, in a normal situation, the number of boundary nodes will be much smaller than the total number of nodes in the subdomain, i.e., we have a surface-to-volume effect. To perform the data exchange, each process has to loop through the set of all adjacent subdomains and, for each subdomain, it has to collect and send the values to the boundary nodes, as well as to receive from the other processes the values corresponding to the halo nodes.

Therefore, each communication phase has a packing step, a network send, a network receive, and an unpacking step. The packing and unpacking steps, which we call *gather* and *scatter*, are quite poor in terms of coalesced memory accesses on GPU due to their irregular access pattern.

- A gather operation packs various elements from a source vector into a contiguous target vector:

  ```
  for(i=0;i<n;i++)
     y[i] = x[index[i]];
  ```

  This is a typical operation used in PSBLAS to prepare a buffer `y` to be sent in the data exchange that is inherent to the parallel sparse matrix-vector product.

- The scatter operation is the inverse of the gather one: we use a received buffer to update a set of vector elements in predefined locations, as in the example below:

  ```
  for(i=0;i<n;i++)
     x[index[i]] = y[i] +
         beta*x[index[i]];
  ```

The main problem is the bandwidth bottleneck in moving data between CPU and GPU; the irregular access pattern worsens the situation, since it precludes an effective use of

coalescent accesses; if all data reside in global memory we are in the worst-case scenario. Some help comes from the L2 cache in the NVIDIA Fermi and Tesla architectures, which aims at mitigating the effect of irregular memory accesses and can bring up to one order of magnitude of performance improvement when the random accesses are localized by sorting. In any case, it is essential to somehow minimize the amount of required data transfers.

3

SpMV on Hybrid CPU/GPU Node

Contents

In this chapter, we face the challange of performing sparse matrix-vector multiplications combining CPUs and GPUs. In particular, we propose three data partitioning algorithms based on regression models of CPUs and GPUs. All the results shown in this chapter have been also reported in [41].

As mentioned in Section 2.2, a sparse matrix format provides significant performance difference according to the architecture on which it is implemented. Thus, choosing the right combination of sparse formats to run on CPU and GPU is a critical task in order to achieve the best performance. A more sophisticated problem to solve is how to partition the amount of computation among the heterogeneous compute

units. On homogeneous nodes (and clusters), a uniform data partition is considered the most efficient because the same amount of work, distributed among equivalent compute units, will require about the same amount of time to be processed; this approach maximizes parallelism and thus performance. By definition, on heterogeneous nodes, this assumption does not hold anymore. Thus, a load balancing policy is needed in order to get the right partition for the heterogeneous compute units. By "right" partition, we mean the partition that arranges the data in such a way that the serial parts of the computations will be aligned, i.e., all compute units will take the same time to execute their local matrix-vector product. A more subtle problem on heterogeneous nodes is the difference in accuracy of the floating point operations executed on CPU and GPU due to the fact that the GPU uses, by default, the fused multiply-add (FMA) operation [48]. Such capability computes the product of two numbers and adds that product to a number (i.e., $x \times y + z$) with only one rounding step; thus, the result will in general be different from a product followed by a sum executed with two rounding steps, making FMA more accurate than performing the operations separately. The numerical behavior of these operations is essentially equivalent for the simple matrix-vector product kernel, that is, the error bounds are of the same quality, but it is not possible in general to have bit-identical results when we move from the CPU to the GPU. The right combination of sparse formats, their coupling during the same SpMV, the selection of the right data partition and its implementation, represent hard tasks to accomplish without the support of a robust and flexible framework. In [41], we chose to use the Parallel Sparse BLAS (PSBLAS [49]) as parallel framework. Using multiple Design Patterns [50] (in particular, the State design pattern), PSBLAS is able to handle efficiently several devices (like GPGPUs [51]) and sparse formats without impacting the communication core.

3.1 Software Techniques for Heterogeneous SpMV

PSBLAS uses multiple design pattern in order to provide flexibility and ease of maintenance. One of the most useful design pattern, in the context of heterogeneous computing, is the State design pattern. The State design pattern is a behavioral pattern that

allows the encapsulation of the object state behind an interface that allows the object type to vary at runtime [50, 51]. In the context of sparse matrix computations, it provides a useful and natural solution to switch at runtime among different storage formats for a given sparse matrix. Therefore, the State pattern allows for an easy handling of heterogeneous computing platforms: the application making use of the computational kernels will see a uniform outer data type, but the inner data type can be adjusted according to the specific features of the processing element that the current process is running on. For instance, the code that sets up the matrix-vector product test is basically:

```
if (have_gpu(iam)) then
  amold => agpu
else
  amold => acpu
end if
call a%cscnv(info,mold=amold)
do i=1,ntimes
   call psb_spmm(done,a,xv,dzero,bv,desc_a,info)
end do
```

where the `have_gpu` function will choose, based on the process index `iam`, which processes will perform the computations on GPU or on CPU cores, respectively.

Workload partitioning for SpMV is quite easy to implement: the more powerful the device, the more rows are assigned to it. PSBLAS allows one to statically define the number of rows to assign to each process involved in the computation. Once the partitioning algorithm decided the amount of rows to assign to a specific process, PSBLAS can easily implement such a partition without any change in the source code.

3.2 Sparse Matrix Formats

In our PSBLAS software library, the default sparse format is CSR (Compressed Storage by Rows); to use GPUs, we rely on an auxiliary CUDA kernel library, named SPGPU[1]. The SPGPU library is based on the ELLPACK format and a variant thereof called HLL;

[1] `http://code.google.com/p/spgpu/`

for these formats, we have created a CPU implementation from which we derived an additional GPU-enabled version, as detailed in [51]. The GPU-enabled versions of the data formats are marked with a G in the name, for instance, HLL is the base CPU format while HLG is the GPU-enabled version. In the same vein, we also developed interfaces for the CSR and HYB formats provided by NVIDIA through version 4.1 of the CuSparse library. In Section 3.4.2, we investigate the performance of three sparse formats on the CPU side and four on the GPU side.

3.3 Load Balancing

In order to face the load balancing issue, we propose and compare three partitioning algorithms based on regression models of the two most important factors: CPU/GPU performance and PCIe bandwidth. During the installation phase of PSBLAS, we execute two different benchmarks that analyze separately the two factors. As in [52], we consider groups of CPU cores as a single computational unit; in fact, on multicore platforms, parallel processes interfere with each other through shared memory, so that the speed of individual cores cannot be measured independently. The first benchmark executes 10000 sparse matrix-vector products and returns the total elapsed time, the time spent for each iteration, and the throughput.That benchmark is executed independently on CPU (4 or 6 cores) and GPU, by varying the matrix size, which is expressed in terms of number of matrix rows. The second benchmark is the *bandwidthTest* provided by the CUDA SDK; we run it twice, in order to get the bandwidth from host-to-device and from device-to-host. For our investigation over the data partitioning algorithms, we consider both the cases where the first benchmark produces a substantial amount of data points (40 data points) as well as few data points (10 data points). The data points are used to build a regression model which will be exploited by the data partitioning algorithms to predict the behavior of the compute elements.

3.3.1 Data Partitioning Algorithms

As we mentioned in [41], the goal of a data partitioning algorithm is to find the workload partition that allows each compute unit to complete its computation at the same time. The regression models produced during the installation phase of PSBLAS are used by the partitioning algorithm in order to predict how much time a compute unit will take for a fixed amount of computation (assigned rows). In all the algorithms presented, the PCIe bus effect is modeled by adding to the GPU computation time the time needed to transmit the data to the host/device. The amount of transmitted data is estimated and depends on the largest partition of matrix rows computed by a processing element (usually the GPU).

3.3.1.1 Linear Algorithm

The first algorithm we consider has a pure algebraic nature; it is based on the assumption that the time required by the CPU and GPU to solve a problem varies linearly with respect to the problem size [53]. This assumption is true for the GPU in many cases, but it does not hold at all for the CPU. Indeed, due to the cache effects, the Error Sum of Squares (SSE) of a linear regression on the CPU trend is almost 2 orders of magnitude greater than for the GPU model. The real strength of this algorithm is that, for very "good" CPU trends, the complexity of the algorithm is reduced to a single formula. Approximating the CPU and GPU times by a line expressed in function of problem size, we obtain $t_1 = a_1 \cdot x_1 + b_1$ and $t_2 = a_2 \cdot x_2 + b_2$, where x_1 and x_2 are the amount of data (number of rows of the sparse matrix) assigned to the CPU and GPU, respectively; we denote this total amount of rows as $r = x_1 + x_2$. The optimal data partition is obtained when $t_1 = t_2$; with some simple algebraic manipulation, we easily obtain $x_1 = \frac{(a_2 \cdot r + b_2 - b_1)}{(a_1 + a_2)}$.

The linear algorithm is very simple and works pretty well when the assumption of linear trend holds. Unfortunately, problems arise when we try to model the CPU trend on a wide range of data, since the regression is less accurate and produces significant errors, especially on small values of problem size.

Unlike [53], our algorithm does not consider the case where all computations are

assigned to the GPU, in order to investigate the behavior of hybrid computations. However, as shown later, in some cases, the use of a hybrid approach can be counter-productive.

3.3.1.2 Iterative Algorithm

The second algorithm we consider uses an iterative approach and allows to use different models (besides lines) to represent the CPU behavior. The main idea is to keep a linear equation of the GPU trend and use a piecewise line to represent the CPU trend. At the limit, the piecewise linear function can be the entire set of data gathered during the installation phase, which uses a linear interpolation to return values which fall between two data points. The iterative algorithm produces better results than the linear one but requires to keep in memory (some) data points retrieved during the installation phase. It converges very quickly (about 20 steps) but needs the trend of the GPU time to be lower than the CPU one. Figure 3.1 shows the convergence toward the optimal solution. This algorithm was proposed in [52, 54] but our implementation differs from it since we use the execution time of every processing element rather than its speed.

3.3.1.3 Hybrid Algorithm

The algorithm we propose makes use of a hybrid approach to get good accuracy and require few computations at run-time, thus combining the benefits of the two previous algorithms. The main problem of the linear algorithm is its poor accuracy related to the linear model of the CPU; on the other hand, the iterative algorithm is more accurate, but it needs to keep in memory the CPU data points and requires a certain amount of iterations that may impact performance negatively.

The idea of the hybrid algorithm is to use the iterative algorithm during the installation phase to find the optimal execution time and to define a regression model of the latter. The GPU is still modeled by a single line which is fairly accurate. With the optimal time values we do not need the CPU model anymore and we can get accurate results at run-time by just using two equations (optimal time and GPU equations).

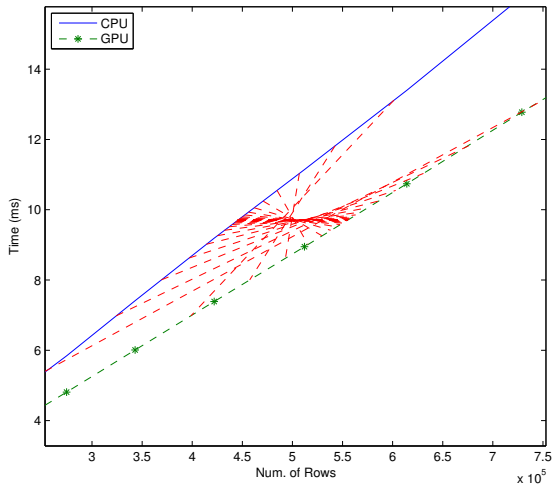


Figure 3.1: Convergence of the iterative algorithm
(convengence steps expressed by red dashed lines)

3.4 Experimental Results

In this section, we first describe the three hybrid CPU/GPU platforms we used in our experiments; then, we discuss the performance results obtained by using the different sparse matrix formats and the data partitioning algorithms over the three platforms.

3.4.1 Hybrid CPU/GPU Platforms

Table 3.1 summarizes the most relevant characteristics of the three hybrid CPU/GPU platforms on which we performed our tests. The AWS platform consists in a single Amazon Web Service (AWS) cluster GPU instance of type CG1; it is equipped with 2 Intel Xeon X5570, quad-core Nehalem architecture with hyperthreading, plus 2 Nvidia Tesla M2050 GPUs and 22 GB of RAM. On this platform we observed an unstable behavior during the execution of the CPU benchmark; such an instability has already been observed in [55]. The PLX platform is a single node of the PLX cluster provided

by the Italian Cineca consortium, which is the largest Italian computing center. It is equipped with 2 six-core Intel Westmere at 2.40 GHz, plus 2 Nvidia Tesla M2070 and 48 GB of RAM. The PLX cluster is ranked at the 266th position in the Top 500 list

Platform	CPU	GPU
AWS	Intel Xeon X5570 (quad-core)	NVIDIA Tesla M2050
PLX	Intel Xeon E5645 (esa-core)	NVIDIA Tesla M2070
Desktop	Intel quad-core Q6600	NVIDIA Geforce GT 520

Table 3.1: Hybrid CPU/GPU platforms

(as of June 2013) and at the 76th position in the Green 500 list. These two platforms were the most widely used ones during the tests, and show quite well how the same data partitioning algorithm can perform differently on similar architectures.

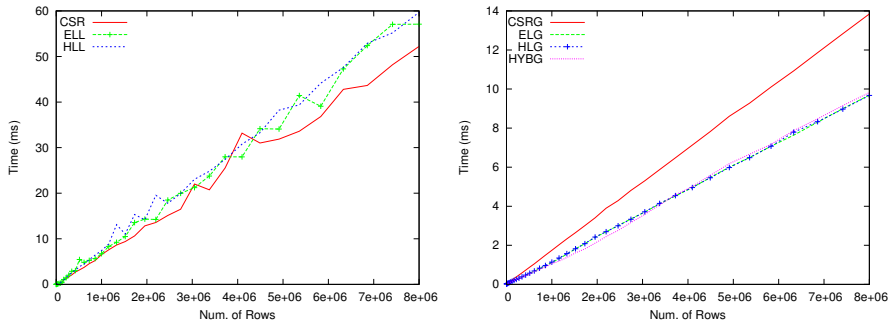
The last platform, named Desktop, represents an unusual solution in the field of scientific computing, since the performances achieved by its CPU and GPU are comparable, that is, the GPU behaves almost as another quad-core socket. While all the three platforms are equipped with Fermi-based cards, the Desktop platform has the lowest performing GPU (with 48 CUDA cores), while the AWS and PLX platforms have very similar GPU cards (with 448 CUDA cores), the AWS card performs slightly better than the one installed on PLX.

In the experiments, we used the benchmark described in Section 3.3. As a performance metric, we report the average execution time for sparse matrix-vector product over 10000 runs.

3.4.2 Performance Analysis

We first analyze the performance of a variety of combinations of sparse matrix formats on the most powerful AWS platform. From Figures 3.2a and 3.2b, we can establish the baseline performance of the various sparse matrix formats on the CPU and GPU processing elements. We observe that, in our test cases, the CSR format is the best one on the CPU; however, it turns out that, at the same time, it is the *worst* format on the GPU, whereas HLG is the best one on the GPU but not on the CPU. Given this result,

in the following, we will use the CSR format on the CPU and the HLG format on the GPU; again, we point out that such a hybrid usage is easily supported by our PSBLAS framework.



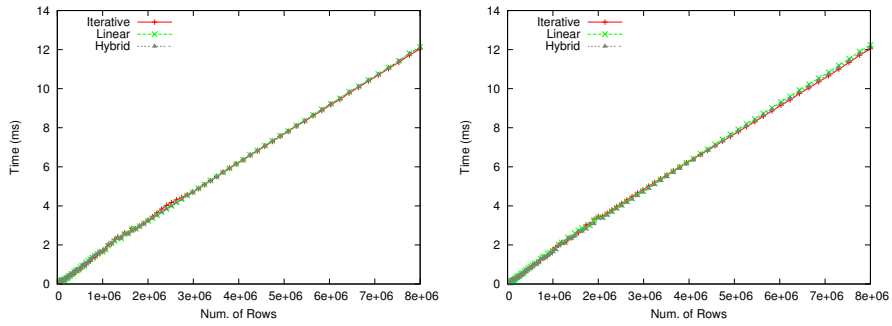
(a) CPU (b) GPU

Figure 3.2: Performance of sparse matrix formats on the AWS platform

We now analyze the performance of the data partitioning algorithms in distributing the workload. Figure 3.3a shows the results of the three algorithms on a typical PLX cluster node with an accurate benchmark execution having 40 data points. With a high sampling rate during the benchmark, the algorithms behavior is practically the same when executing over a stable platform, such as PLX. Figure 3.3b represents the same test case with less data points (only 10) gathered at installation time. We see that, for a stable architecture, we do not get benefits from an accurate preliminary analysis.

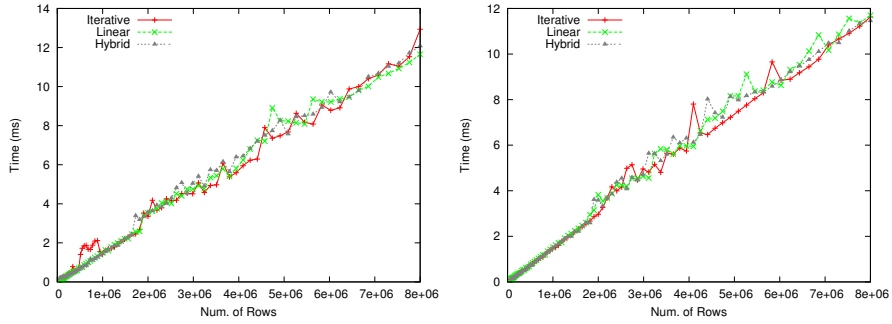
However, results differ on the less stable AWS platform, as shown in Figures 3.4a and 3.4b. On the latter, the iterative algorithm fits the unstable behavior of the CPU, thus producing wrong results. On the other hand, the linear algorithm is shielded from such instability and achieves better results.

The third set of experiments is on the Desktop platform, which represents the lowest performing architecture. As shown in Figure 3.5a, the stability of the Desktop platform in terms of performance variations makes the iterative and hybrid algorithms the worst. The linear algorithm does not encounter any difficulty due to the linear behavior of



(a) 40 data points (b) 10 data points

Figure 3.3: Performance of data partitioning algorithms on PLX platform



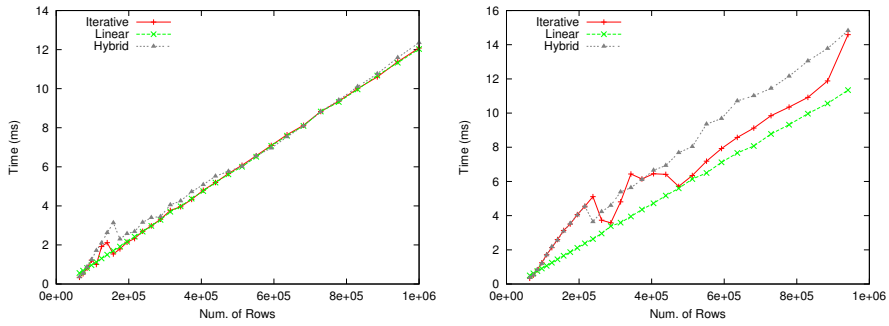
(a) 40 data points (b) 10 data points

Figure 3.4: Performance of data partitioning algorithms on AWS platform

CPU and GPU. Even with an inaccurate sampling (see Figure 3.5b), the same result holds.

To prove the effectiveness of the iterative algorithm, we compare it against the exhaustive optimum search executed over PLX for a matrix with 1000000 rows. As shown in Figure 3.6, we are really close to the real optimum.

A final note regards the presence of multiple processing elements, in particular



(a) 40 data points

(b) 5 data points

Figure 3.5: Performance of data partitioning algorithms on Desktop platform

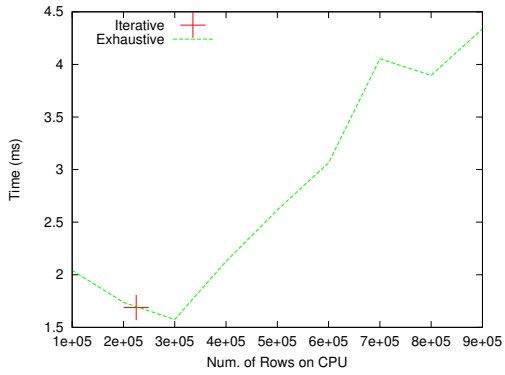


Figure 3.6: Comparison of the iterative algorithm with the exhaustive search

CPUs. Taking advantage of more computational units is strictly related to the architecture and the load balancing algorithm. In most experiments, we observed that there is no significant gain when using more CPU cores. This is mainly due to the impact of the MPI communication overhead, amplified by unbalance among cores. Furthermore, the most important limit to scalability is the PCIe bandwidth. For example, with a matrix with 8000000 rows, we have to transmit, for each iteration, about 320000 bytes (with double precision); we know that, on PLX, the PCIe can reach at most 5840 MB/sec.

With a simple formula, we can estimate the matrix size that saturates the PCIe bandwidth: on the PLX platform, this limit is 10648000 rows (that is, about 387200 bytes for each iteration). For that amount of data, the bus reaches 5440 MB/sec which is very close to its maximum. During our experiments, PSBLAS assigns rows to processes by using a block decomposition strategy which may cause harmful communication overheads. Therefore, our idea is that multiple computational units may produce significant improvements when a more intelligent load distribution is used. A better strategy may be to rely on graph partitioning distribution (provided by PSBLAS as well), which aims at creating domain partitions that require as little communication as possible.

During the experiments, we observed that the use of hybrid computation is not always beneficial. In fact, we get significant benefits only for the Desktop platform, whose CPU and GPU are very similar in terms of performance. On AWS, which has the most powerful GPU among the 3 architectures, it is more convenient to use only the GPU. On PLX, we get a small improvement for small matrices, where the CPU and GPU throughputs are almost the same. As a future development, it may be useful to define an index which expresses the affinity of a specific platform to hybridization.

4

SpMV on Clusters of GPGPUs

Contents

In June 2015, among the ten most powerful computers in the world, five were equipped with accelerators on their compute nodes. In particular, the two most powerful clusters (Tianhe-2 and Titan) belong to this category. In Section 2.6, we already examined some techniques employed for inter-node communication on heterogeneous

clusters. In this chapter, we discuss how to integrate GPU-enabled computational kernels for SpMV into the PSBLAS library. We report our findings on which strategies are more promising in the quest for the optimal compromise among raw performance, speedup, software maintainability, and extensibility. We consider several solutions to implement the data exchange with the GPU, focusing on data access and transfer, and present an experimental evaluation for a cluster system with up to two GPUs per node. In particular, we compare the pinned memory and the OpenMPI approaches, which are the two most used alternatives for multi-GPU communication in a cluster environment. We consider various alternative strategies to realize the data exchange needed by a fully parallel version of the PSBLAS library with CUDA support, where multiple PSBLAS processes, each one using a GPU device for the SpMV kernel computation, inter-communicate. Specifically, the alternative strategies for data transfers from CPU to GPU and vice versa include CUDA Peer-to-Peer, synchronization, scatter and gather kernels, and static index in two versions, namely standard and pinned memory. We also present a strategy which exploits specialized data transfer support available in OpenMPI. We compare the performance of the various data exchange approaches when executing the sparse matrix-vector multiplication in a heterogeneous cluster environment with different configuration scenarios in terms of number of nodes and number of GPUs. In particular, our experimental results demonstrate that OpenMPI turns out to be the best solution for large data transfers when using multi-GPU communication in a cluster environment, while the pinned memory approach is still a good solution for small transfers between GPUs. Most research efforts that propose application optimizations on heterogeneous systems with GPUs (e.g. [56–58]) typically rely on the overlapping of the MPI communication, the CPU-GPU communication, and the CPU-GPU computation.

In particular, in [57], Kreutzer et al. focused on sparse matrix-vector multiplication on GPGPU clusters and considered three alternatives for communication and computation: no overlap of communication and computation, naive overlap of communication and computation by nonblocking MPI, and the use of dedicated host threads for asynchronous MPI communication. The latter approach achieves the best results in their experimental setting, which however differs from ours both in terms of GPU cluster

architecture and set of matrices, such that the performance results are not comparable.

4.1 PSBLAS with NVIDIA GPUs Support

The latest release of the PSBLAS library (version 3.1.2) is a reimplementation in the Fortran 2003 language; the new internals have a full object-oriented (OO) design, as described in [49]. The availability of the OO infrastructure enables an easy implementation of a CUDA "serial" plugin [51]; here "serial" refers to the use of just one PSBLAS process invoking kernels on a single GPU. In this mode, we can test how efficient the computational kernels are without the burden of communication among processes.

spGPU[1] is a set of custom matrix storages and CUDA kernels for sparse linear algebra computing on GPU that we implemented. It includes a new GPU-friendly storage format named ELL-G [51]. It is a variation of the standard ELLPACK (or ELL) format [59] and aims at reducing the memory overhead of the padding zeros that occurs in ELL. Indeed, the ELL format introduces padding with zero coefficients to fill unused locations of the elements array, but its efficiency highly depends on the distribution of nonzero elements. When the number of nonzeros per row varies considerably, the ELL performance degrades due to the overhead of the padding zeros. Since, in the GPU implementation, it is not necessary to execute arithmetic operations on these padding entries, a better solution is to create an additional array of row lengths, so that each thread will only execute on the actual number of nonzero coefficients within the row, at the cost of one more memory access. Other researchers have used a similar solution, for instance in the ELLPACK-R format described in [60].

The PSBLAS object model enables an easy translation among different sparse matrix formats [51] as well as easy extensibility in user code; it is thus possible to wrap the ELL-G format and use it in the PSBLAS context with the following class (in the code, we refer to ELL-G as `elg` because PSBLAS matrix format names are 3 characters long):

[1] `http://code.google.com/p/spgpu/`

```
type, extends(psb_d_ell_sparse_mat) &
      &  :: psb_d_elg_sparse_mat
#ifdef HAVE_GPU
  type(c_ptr) :: deviceMat = c_null_ptr
contains
  procedure, pass(a) :: &
     & d_sp_mv => psb_d_elg_vect_mv
  procedure, pass(a) :: &
     & d_csmm => psb_d_elg_csmm
  generic, public    :: &
     & cp_from => psb_d_elg_cp_from
  procedure, pass(a) :: psb_d_elg_mv_from
  generic, public    :: &
     & mv_from => psb_d_elg_mv_from
  procedure, pass(a) :: &
     &to_gpu  => psb_d_elg_to_gpu
#endif
end type psb_d_elg_sparse_mat
```

The `deviceMat` attribute holds a pointer to a shadow image of the matrix data structure which resides in the GPU memory space. A similar encapsulation is applied to vectors.

spGPU spGPU is a library which implements the computational kernels in CUDA C language as well as the data movement operators invoked from PSBLAS when dealing with the shadow-memory copies of matrices and vectors. Once the data is in the GPU memory, it is possible to invoke the computational kernel, such as that for the sparse matrix-vector product $y \leftarrow \alpha Ax + \beta y$.

4.2 Parallel PSBLAS-GPU Alternatives

In this section, we analyze the possible approaches to implement the data exchange needed for PSBLAS-GPU, highlighting advantages and drawbacks of each alternative.

4.2.1 CUDA Peer-to-Peer

Besides sharing the same Virtual Address Space between CPU and GPU, the Unified Virtual Address (UVA) space [61] introduced in CUDA 4.1 brought the Peer-to-Peer access support. This mechanism allows two GPUs, installed on the same node, to communicate directly without involving the CPU.

Since our programming model uses different processes, Peer-to-Peer cannot be directly used.

In our software architecture, the CPU acts as a front-end for the various computation and we use it to control the scatter/gather operations. Thus, the usage of multiple GPUs with Peer-to-Peer access would require a specialized storage format that extends across multiple GPUs, which we have not designed and implemented so far.

4.2.2 Sync: Brute Force Solution

The simplest solution is to use a `sync` operation to move the vector data between device and host for each scatter/gather operation. The `sync` method is normally intended to seed the device version upon start of a computation, and to recover the results at the end of a (possibly long) chain of operations carried out within the GPU. Since the parallel halo communication is handled by existing CPU-side code, this implementation does not require any specific GPU code beyond the serial data movement. It is clear that such a solution is not optimal, since at each step we will be moving around a much larger set of data than necessary; therefore, this strategy will only establish a minimum baseline performance to be improved upon.

4.2.3 Scatter and Gather Kernels

A better solution is to implement the gather/scatter methods inside the GPU, so as to only transfer the boundary data between host and device. This solution is fairly simple but presents two drawbacks: the irregularity of the memory access patterns and the need to handle arrays of indices. Indeed, the indirect addressing is based on a data structure that is built on the host side. Therefore, to execute the gather/scatter operations we

need to have a copy of the indices on the device side, and this requires the invocation of the `cudaMalloc()`, `cudaMemcpy()`, and `cudaFree()` kernels.

4.2.4 Pinned Memory Version

To avoid the data traffic associated with the indices and the buffers needed for packing and unpacking, one possible strategy is to use the mapped memory. The latter is a particular page-lock host memory allocation provided by CUDA, which can be accessed directly by CUDA kernels.

Using the mapped memory for the indices and the source/destination buffers, we can avoid the time spent on allocating, copying, and deallocating; furthermore, there is a significant improvement of the PCI-E bus utilization. CUDA provides a memory management function called `cudaHostRegister()` which transforms a normal host memory area into a pinned memory area; previous restrictions on its usage have been lifted from CUDA 4.1. On the library side, the use of pinned memory requires to store a couple of new fields in the data structures and to register/deregister them as needed during the application lifetime.

4.2.5 Static Index on GPU (Standard and Pinned Version)

As already seen, a well-known best practice for optimizing codes on NVIDIA GPUs and other accelerators is "send data on GPU and keep it there". Since the index lists in the communication are based on the topology of the discretization mesh, they are quite stable during the application life; the lifetime of a communication descriptor is tied to the lifetime of a discretization mesh, and certainly this covers multiple matrix-vector products, i.e., data exchanges. It is therefore clear that a viable solution would be to keep a copy of the indices in the device memory throughout the life of the descriptor object. An alternative is to keep the indices in pinned memory. In the rest of the paper, we will refer to the two alternatives as *IndexStandard* and *IndexPinned*, respectively.

4.2.6 Open MPI with CUDA Support

A completely different solution can be implemented through the use of MPI derived data types, specifically using the `MPI_Type_indexed` function, which allows the creation of a data type with irregular stride(s) among the components. We can then use a specific derived data type to describe the boundary elements taking part in a data exchange. The upshot would be that the packing and unpacking operations would be performed directly by the MPI library.

This approach becomes interesting when coupled with the support provided by Open MPI [62] for UVA usage [63], so that all pointers within a program have unique addresses, and a new API that allows to check if a pointer is either a CUDA device pointer or a host memory pointer (used by the library to detect if the memory area used in a send/receive is a device area or not). Furthermore, CUDA 4.1 adds the CUDA IPC (InterProcess Communication), which allows a fast communication between GPUs on the same node, as well as between different processes. In addition, CUDA 4.1 provides the ability to register host memory with the CUDA driver, which can improve performance.

In other words, it is possible to delegate *both* packing and movement between host and device memory to the MPI implementation. At the time of this writing, the Open-MPI CUDA-aware support is approaching full maturity. It now fully exploits the capabilities of the newer CUDA versions, such as GPUDirect RDMA available from CUDA 6.0 in Kepler-class GPUs, and is quite stable.

4.3 Experimental Results

The experimental results presented in this section are organized in two different sets. We first analyze in Section 4.3.1 a comparison among the "CUDA native" approaches, which include: Sync, Scatter/Gather (for short, SG), Pinned, and the two variants of Static Index, namely IndexPinned and IndexStandard, which have been described from Section 4.2.2 to Section 4.2.5. Then, in Section 4.3.2, we present a comparison between the best solution coming from the CUDA native comparison and OpenMPI with CUDA

support approach.

The CUDA native experiments have been executed on the Jazz GPU cluster provided by CINECA[2]. Its nodes are based on dual-socket hexa-core Intel Xeon X5650 processors (for a total of 12 cores), with 48 GB RAM; each node has two Nvidia Fermi S2050 devices and is interconnected with QDR InfiniBand (40 Gbit/s). The second set of experiments has been executed on Amazon Web Service (AWS) EC2 GPU instances of type CG1; each node is equipped with two Intel Xeon X5570, quad-core with hyperthread plus two Nvidia Tesla M2050 GPUs. Each AWS CG1 instance is interconnected with a low latency 10 Gbit/s network. Furthermore, a scalability test has been executed on several AWS EC2 G2 instances; each node has an Intel Xeon E5-2670 (Sandy Bridge), 15GB RAM plus one NVIDIA GRID Kepler GK104. Note that both CG1 and G2 instances do not not have an Infiniband network.

To evaluate the PSBLAS-GPU performance, we used a relatively simple main program which iterates the sparse matrix-vector multiplication for a specified number of times; the performance metric of interest is the global throughput, expressed in GFLOPS. The sparse matrix is generated from an advection-diffusion equation on the unit cube. The equation is discretized with a simple centered differences strategy, giving rise to a matrix with at most 7 nonzeros per row: the matrix size is expressed in terms of the length of the cube edge, so that the case pde10 corresponds to a 1000×1000 matrix. We already used this sparse matrix collection in [51]. This highly structured matrix allows us to easily generate matrices of different sizes; anyway, for such a regular pattern, there are much more efficient methods than using a general-purpose sparse library.

The availability of two GPUs per node enables different configurations with the same number of processes. However, switching from single to dual occupancy per node is by no means neutral in terms of performance, because with double occupancy, contention for the PCIe bus becomes the main performance bottleneck.

[2] `http://www.cineca.it`

4.3.1 CUDA Native Experiments

During a preliminary test phase, the Sync approach obtained, as expected, the lowest throughput; therefore, it has been discarded for large memory sizes.

4.3.1.1 1 Node with 2 GPUs (1-2)

With 1 node and 2 GPUs (for brevity, 1-2 scenario) the interconnection network is not involved; Figure 4.1a shows the corresponding performance comparison.

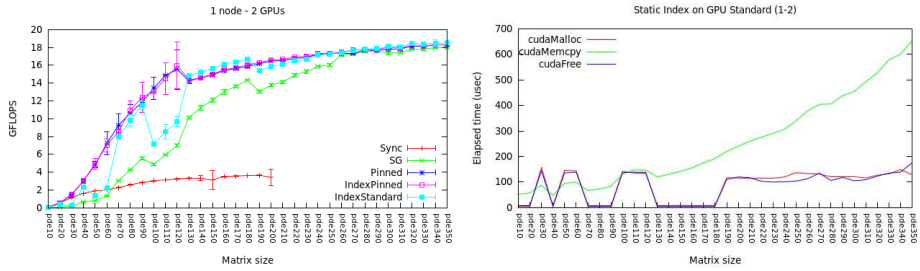


Figure 4.1: Throughput (a) and elapsed time in CUDA functions (b) for the (1-2) scenario on CINECA platform

The poor performance obtained by the Sync approach is obvious and expected because of the involved memory bottleneck. The performance dips exhibited by the Scatter/Gather (SG) and IndexStandard approaches are slightly surprising; they have been found to be reproducible and are ultimately caused by the behavior of the memory management routines `cudaMalloc()`, `cudaMemcpy()` and `cudaFree()`. The SG approach uses them on both the index array and MPI buffer, while indexStandard uses them only on the MPI buffer; thus, IndexStandard has a similar trend to SG but better performance. This explanation is confirmed by the pinned alternatives, showing a much smoother performance curve.

To confirm our explanations, we collected detailed timings with the same number of repetitions as in the matrix-vector case; Figure 4.1b shows the results, clearly indicating that the memory management overhead has a rather complex behavior, probably

related to internal algorithmic and/or hardware thresholds. The sharp rises and falls in Figure 4.1b match perfectly those reported in Figure 4.1a.

4.3.1.2 2 Nodes with 1 GPU (2-1)

In the 2 nodes with 1 GPU scenario (for brevity, 2-1), we use only one core and one GPU from each node; the results shown in Figure 4.2a indicate a better performance obtained by all the approaches with respect to the 1-2 scenario, because of the absence of contention on the PCIe bus.

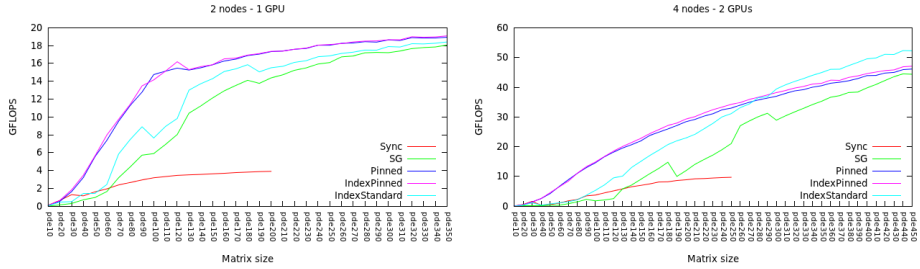


Figure 4.2: Throughput for 2-1 scenario and 4-2 scenario on CINECA platform

This is substantiated by a simple memory bandwidth test, measuring the time needed to transfer a fixed amount of data from CPU to GPU for both pageable and pinned memory. Using 1 node with 2 GPUs and pageable memory, we reach 3035 MB/s, whereas using pinned memory we reach 3614 MB/s. When we employ 2 nodes with 1 GPU, with pageable memory we obtain a throughput of 3457 MB/s, and with pinned memory we reach 5511 MB/s.

4.3.2 CUDA Native vs. MPI-Based Experiments

For the second set of experiments executed on the AWS EC2 GPU cluster, we select the IndexPinned approach as the best performing one from the previous comparison and we name it as Pinned in the corresponding performance curves.

4.3.2.1 1 Node with 2 GPUs MPI (1-2)

With 1 node and 2 GPUs, the results reported in Figure 4.3 show that the OpenMPI support for CUDA IPC works pretty well for sparse matrices having more than 1 million rows.

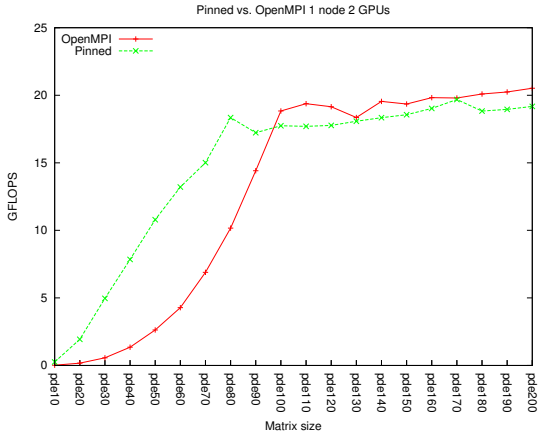


Figure 4.3: Throughput for (1-2) scenario on AWS platform

4.3.2.2 2 Nodes with 1 GPU MPI (2-1)

In the 2 nodes with 1 GPU scenario, we have to consider the impact of network overhead. Since we use two nodes interconnected by a 10 Gbit/s network, as shown in Figure 4.4 we see an expected performance decrease when compared to the 1-2 scenario reported in Figure 4.3. Also, in this scenario, the OpenMPI-based implementation works well for matrix sizes larger than 1 million rows.

4.3.2.3 2 Nodes with 2 GPUs MPI (2-2)

In the 2 nodes with 2 GPUs setting shown in Figure 4.5, we found that the OpenMPI approach shows great instability with respect to the Pinned one. There are two possible

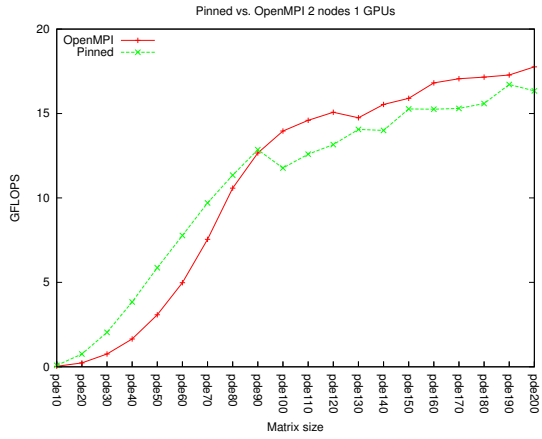


Figure 4.4: Throughput for (2-1) scenario on the AWS platform

motivations for such a strange behavior. The first one is related to the data partition algorithm: for every test, we use a block partition algorithm which operates in a "data blind" way. The quick jumps between high and low performance appear to be related to load imbalance. Indeed, every matrix with a size non divisible by 4 (number of processes in the scenario) is affected by low performance. During the scalability test executed on the G2 instances, we observed the same unstable behavior. This means that the CUDA IPC are not directly related to this instability (CUDA IPC is used only on multi-GPU nodes). However, the same partition algorithm has been applied to the Pinned approach, which does not show the unstable behavior; thus we believe that the OpenMPI CUDA support is more sensitive to load imbalance, although this will require further investigation.

In order to figure out where the instability comes from, we applied the TAU Performance System on our application. For matrices with size not exactly divisible by the number of processes, we pay a penalty during the `MPI_Send()` execution.

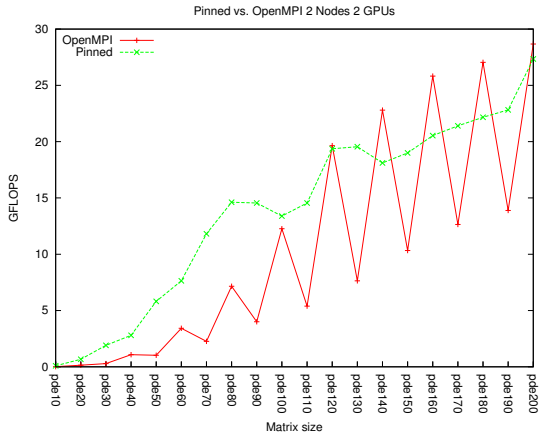


Figure 4.5: Throughput in (2-2) scenario on AWS platform

4.3.2.4 Scalability Test

In order to find out which data exchange approach achieves the best scalability performance, we ran a scalability test on up to 8 EC2 G2 instances, where each instance has only one GPU. In Figure 4.6, the y-axis represents the weak scalability speedup. Regarding the matrices used for the scalability test, for the single node execution, we used the pde160 matrix and we doubled the memory occupation every time we doubled the number of nodes.

Figure 4.6 shows that, without any support for the OpenMPI features, the Index-Pinned approach performs better than the OpenMPI one. This is reasonable because OpenMPI adds the overhead needed to "understand" which kind of memory (CPU or GPU) it handles.

4.3.3 Conclusions

In this chapter, we have analyzed how to integrate GPU-enabled computational kernels into PSBLAS and we have considered several solutions to implement the data exchange

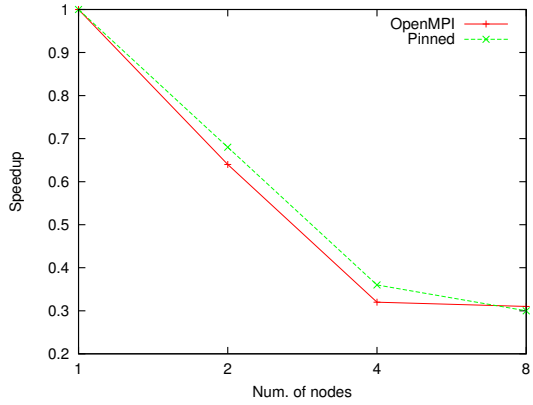


Figure 4.6: Scalability on AWS platform using EC2 G2 instances

with the GPUs. In particular, we have presented a comparison between the two most used alternatives for multi-GPU communication in a cluster environment. The pinned memory approach is still a good solution for small transfers between GPUs. The great improvement of OpenMPI compared to a couple of years ago brings an unexpected performance enhancement: it turns out to be the best solution for large data transfers in every experiment. However, on a bare cluster without any support for the OpenMPI features, we have evidenced that the Pinned version performs better than the OpenMPI version. We also tested the alternative data exchange approaches using a different set of sparse matrices from Tim Davis's collection. The results do not differ from those shown and therefore, for a metter of space, we have not reported them.

In future work, we plan to implement a data-aware partitioning algorithm in order to optimize the load balance among the processes. Furthermore, we plan to test a version which uses the GPUDirect support provided by OpenMPI. We will also consider the recently proposed GPU-aware MPI [64], which supports data communication from GPU to GPU using standard MPI and has been incorporated into MVAPICH2. We have already run some preliminary tests about MVAPICH2 with CUDA-aware support and the results are very promising.

Part II

Partitioned Global Address Space languages: coarray Fortran

5

Background and Related Work

Contents

This chapter presents a background on coarray Fortran and the Partitioned Global Address Space (PGAS) programming model that they support. We also focus on the major characteristics that make coarray Fortran suitable for exascale computing.

5.1 Partitioned Global Address Space Model

As stated in Section 1.1.2, one of the biggest limitations of the classic two-sided approach provided by MPI is the need for a handshake between the sender and receiver of each message. The number of cores in a single exascale node is supposed to increase,

with a consequent increase of the number of messages going off the node. Because moving data off-node has a huge impact on performance and energy consumption, the costs for handshake and implicit synchronization imposed by the two-sided approach will be a huge limiting factor to scalability, performance and energy efficiency.

Some interconnects have the ability to perform a direct access to the memory exposed by another process without involving it in the transfer. In other words, a process *A* can put or get data from the memory exposed by process *B* while process *B* is busy with other tasks than communication (like computation). Therefore, the match between sender and receiver during the message transfer is no longer needed.

With this capability, a new programming paradigm, called PGAS, has evolved in the last few years. The Partitioned Global Address Space model is a parallel programming model that assumes a global memory address space logically partitioned, with a portion of the memory being assigned to a specific processor. The model attempts to combine (and get the best from) the Single Program Multiple Data (SPMD) approach, used in the distributed memory systems, and the semantic of shared memory systems. In the PGAS model, every process has its own memory address space but it can share a portion of its memory to other processes.

There are several cases when a PGAS approach can easily solve difficult message passing situations because of the one-sided semantic. Typical examples of scientific applications that get benefits from a one-sided approach are adaptive mesh refinement and particle trackers [65–67]. Another case where PGAS languages can be effective is when the application needs to perform message-intensive collective communication followed by an intensive computation. A typical case is a 3D-FFT, where the application is a succession of blocking MPI_ALLTOALL and intensive computation (without communication) on the processes. In this scenario, using a PGAS language, it is possible to send data to the destination process as soon as they have been computed, thus overlapping communication and computation and removing the blocking collective operation [68].

In general, whenever the communication is irregular and/or there is space for overlapping communication with computation, PGAS languages can show significant performance improvement.

5.1.1 PGAS Languages

The most relevant PGAS languages at this time are coarray Fortran (CAF) [69,70], Unified Parallel C (UPC) [71], OpenSHMEM [72], Chapel [73], Fortress [74], X10 [75] and Global Arrays [76]. Several of these languages (CAF and UPC in particular), provide interoperability with MPI. Such capability allows users to incrementally modify an existing MPI code by replacing MPI statements with a more efficient PGAS-based approach. This mixture of MPI+PGAS allows one to exploit the best of the two: a regular communication pattern, like a producer-consumer, could be implemented using MPI two-sided routines, whereas an irregular pattern, like a dynamic load balancing algorithm, could be implemented following a PGAS paradigm.

CAF and UPC are provided as part or extension to a well known programming language. Because of that, a substantial part of the complexity of these two PGAS languages is delegated to the compiler. The compiler must understand the syntax, generate the right get/put and synchronization calls and, hopefully, optimize the communication pattern as much as possible. We will provide more details about the CAF support provided by compilers in the next chapter.

Because the power of the PGAS model is based on the efficiency of one-sided operations, particular attention has to be paid to the interconnect. The NIC must support Remote Direct Memory Access on distant nodes and it should provide real asynchronous transfer progression (more details will be given in Section 5.3). Furthermore, using a PGAS language will most likely mean an increased number of small messages flying over the network; for this reason, the interconnect should provide very low latency.

5.2 Coarrays and Exascale

As we already mentioned in Section 5.1, PGAS languages seem to be a promising parallel programming model for the exascale era. In this section, we introduce coarray Fortran and describe the coarray features, already included in the Fortran 2008 standard and proposed by TS-18508 [77], that match the needs of exascale computing. All the features specified by TS-18508 will, most likely, be part of the Fortran 2015 standard.

5.2.1 Introduction to Coarray Fortran

Coarray Fortran (also known as CAF) is a syntactic extension of Fortran 95/2003 which was proposed in the early 1990s by Robert Numrich and John Reid [69] and is now part of the Fortran 2008 standard (ISO/IEC 1539-1:2010) [70]. The main goal of coarrays is to allow Fortran users to create parallel programs without the burden of explicitly invoking communication functions or directives such as with MPI and OpenMP. Currently, coarrays make Fortran the only internationally standardized language with an intrinsic parallel programming model that scales to massively parallel platforms.

A program that uses coarrays is treated as if it were replicated at the start of execution; each replication is called an image. Each image executes asynchronously and explicit synchronization statements are used to maintain program correctness. A typical synchronization function is *sync all*; it can be intended as a barrier for all images. A piece of code contained between synchronization points is called *segment* and a compiler is free to apply all its optimizations inside a segment. An image has an image index, that is, a unique number ranging between one and the number of images (inclusive). In order to identify a specific image at run-time, or determine the total number of images, the `this_image()` and `num_images()` functions are provided. A coarray can be a scalar or an array, static or dynamic, of intrinsic or derived type. In order to access a coarray object on a remote image, the [ ] operator has to be used; if it is not, the object is considered local. A simple Fortran program which uses coarrays is given below:

```
real, dimension(10), codimension[*] :: x, y
integer :: num_img, me

num_img = num_images()
me = this_image()

! Some code here
x(2) = x(3)[7] ! get value from image 7
x(6)[4] = x(1) ! put value on image 4
x(:)[2] = y(:) ! put array on image 2

sync all
```

```
! Remote-to-remote array transfer
if (me == 1) then
  y ( : ) [ num_img ] = x ( : ) [ 4 ]
  sync images(num_img)
elseif (me == num_img) then
  sync images(1)
end if

x(1:10:2) = y(1:10:2)[4] ! strided get from 4
```

In the example above, `x` and `y` are coarray arrays and any image can access these variables on any other image. Note that all the usual Fortran array syntax rules are valid and applicable, except on the codimensions.

The Fortran Standard further provides locks, critical sections and atomic intrinsics. Furthermore, Technical Specification 18508 (TS-18508) specifies the form and establishes the interpretation of features that extend the Fortran 2008 standard. More details about the new features introduced by TS-18508 will be given in Section 5.2.2.

5.2.2 Coarray Features for Exascale

Considering what we said in Sections 1.1.1 and 1.1.2, in order to match the needs of exascale computing, a programming model should be able to:

- be resilient against faults and failures;
- offer fine grain synchronization routines;
- reduce inter-node communication as much as possible;
- manage easily irregular communication patterns in order to handle heterogeneity and unexpected variations.

The coarray definition included in Fortran 2008, as standardized by ISO/IEC 1539-1:2010, defines a simple syntax for accessing data on remote images, synchronization statements and collective allocation and deallocation of memory on all images. Although these features allow one to write a totally functional coarray program, they do

not allow to express more complex and useful mechanisms for synchronization, images organization and failure management.

Technical Specification 18508 proposes the following extensions to the coarray facilities defined in Fortran 2008:

- teams;
- failed images;
- events;
- new intrinsic procedures.

In the next subsections, we describe these extensions and how they satisfy the needs of exascale computing listed above.

5.2.2.1 Teams

The existing coarray definition in Fortran 2008 does not provide for an environment where a subset of the images can easily work on part of an application without affecting other images in the program. Grouping the images of a program into nonoverlapping teams allows one to execute more effectively and independently parts of a larger problem. A class of problems that can benefit of such feature is multiphysics codes (e.g. climate models). Such feature can significantly reduce the amount of off-node communication on an exascale machine, in particular when an entire team of images fits within a single compute node.

5.2.2.2 Failed Images

As the name suggests, this extension provides a mechanism to identify what images have failed during the execution of a program. This obviously affects the resilience of programs running on large systems.

5.2.2.3 Events

The coarray synchronization primitives available in Fortran 2008 do not provide a convenient mechanism for ordering execution segments on different images without requiring that those images arrive at synchronization point before any is allowed to proceed. Events implement a fine grain ordering of execution segments based on a limited implementation of the well known semaphore primitives.

5.2.2.4 New Collectives and Atomics Intrinsics

Fortran 2008 does not provide intrinsic procedures for commonly used collective and atomic memory operations. Such procedures can be highly optimized for the target computational system, providing significantly improved program performance. A typical example of collective operation introduced by TS-18508 is *co_broadcast*. Such intrinsic allows one to broadcast data from a source image to a group of images as one single command. In Fortran 2008, the only way to implement this operation is to run a do-loop on the source image and using a "put" on each target image, one at a time. TS-10508 enriches the available set of atomic intrinsics. In particular, it adds a set of *atomic_fetch_and_op* intrinsics that allow one to implement dynamic load balancing algorithms. A practical demonstration on how these atomic intrinsics can be used for this purpose is given in Sections 7.3 and 7.4.

5.2.3 Coarray Support

Coarrays were first implemented in the Cray Fortran compiler; nowadays, this implementation is considered the most reliable and efficient. Since the inclusion of coarrays in the Fortran standard, the number of compilers implementing them has increased: besides the Cray Fortran compiler, the Intel ifort, Rice compiler [78], OpenUH compiler [79], and the g95 compiler support coarrays. The Cray compiler only runs on proprietary architectures, while Intel's compiler is only available for Linux and Windows and requires the usage of the Intel Cluster Toolkit. These technical limitations, in conjunction with the cost of a proprietary compiler, limit the widespread usage of coarrays.

The availability of an open-source and widely used compiler that supports coarrays represents a useful and demanded contribution for the parallel computing programmers.

Currently, there are three free compilers that support coarrays: Rice Compiler [78], OpenUH [79] and g95. Unfortunately, they do not support several standard Fortran features and are not widely used by the Fortran user community. g95 provides a free coarray support only for the shared memory configuration; the multi-node coarray support is privative. Furthermore, it does not support many of Fortran 2003/2008 features such as support for OOP. In Chapter 6, we present the first open-source implementation of Fortran parallel programming model on the most widely used free compiler: GNU Fortran.

5.3 MPI as PGAS Transport Layer

The Message Passing Interface (MPI) execution model, thanks to its high performance, portability and standardization is a de-facto standard in the High Performance Computing world and it is installed and tuned on all supercomputers. The MPI standard has evolved from the initial version of 1994 and currently incorporates direct remote memory access (RMA) through one-sided functions, multi-threading support, non-blocking and sparse collective communication operations and dynamic process management. Such new features make MPI-3.0 a good candidate for being the transport layer of PGAS languages [80, 81]. In particular, the asynchronous communication required by the PGAS model can be easily implemented on top of the RMA one-sided functions of MPI-3.0. Although these operations map very well on the RDMA read and write operations provided by HPC network fabrics (like Cray Gemini [82], IBM Blue Gene/Q [83] and Infiniband [84]), the synchronization models associated with the MPI one-sided operations are somewhat complicated. The MPI standard provides two synchronization modes: active and passive. In the active mode, the target process participates in the synchronization; on the other hand, in the passive mode, the target process does not participate in the synchronization. In the latter case, all the processes accessing the memory exposed by a remote process have to synchronize amongst themselves, with-

out participation of the target process. From the point of view of providing support for a PGAS language, the passive mode is the most suitable; in fact, it allows to overlap communication with computation, reducing the synchronization penalty.

Implementing passive MPI one-sided functions, even on network interfaces able to perform RDMA operations in hardware (overlapping communication with computation), may require the MPI implementation to check for transfer completion in order to *progress* the communication.

In [85], the authors provide a detailed description of what "Progress" and "Overlap" mean and how they impact the performance of MPI applications.

Overlap is a characteristic related to the network layer; it represents the capability of the NIC to take care of the data transfer without the direct involvement of the host processor, allowing the CPU to be dedicated to computation.

Progress is a characteristic related to MPI, which is placed several levels higher than the network. The MPI standard defines a Progress Rule for asynchronous communication operations. Unfortunately, there are two views of this rule, which lead to two different behaviors, both compliant with the standard. The strict interpretation of the Progress Rule is that once a nonblocking communication operation has been posted, a matching operation will allow the operation to make progress regardless of whether the application makes further library calls. In short, this interpretation mandates non-local progress semantics for all non-blocking communication operations once they have been enabled. The weak interpretation allows a compliant implementation to require the application to make library calls in order to have outstanding communication operations progress.

In general, it is possible to support overlap without supporting independent MPI progress. For example, an InfiniBand network is usually capable of performing Remote Direct Memory Access (RDMA) operations and fully overlap communication and computation for contiguous PUT/GET operations. However, with such operations, the target address has to be known. If the transfer of the target address depends on the user making an MPI library call, then progress is not independent. Furthermore, if the application requires a non-contiguous transfer or an accumulate operation, the target process has to unpack the (contiguous) data stream received from the network into the

non-contiguous target locations.

It is also possible to have independent MPI progress without overlap. An example of this configuration is the implementation of MPI for ASCI Red [86]; in this case, the network provides interrupt-driven progress, but the data transfer has to be managed entirely by the host processor.

Asynchronous message progress is a very intricate and controversial topic in high-performance computing [85, 87, 88]. With the current available high-performance networks, there are essentially three strategies for making progress: manual progress, thread-based progress, and communication offload.

The manual progress gives complete control and responsability to the programmer for implementing message progress. The programmer has to explicitly invoke functions like MPI_Test and/or MPI_Iprobe inside the code in order to make the communication progress. Although this solution increases code complexity, it is quite used in several cases because the MPI implementation is much more faster without the burden of the full asynchronous support. Unfortunately, it is very unlikely that the user invokes a progress function like *MPI_Test* or *MPI_Iprobe* at the right moment. It will probably come too early, where there is nothing to progress, or too late, wasting the opportunity to overlap.

The thread-based progress has been often considered the most effective because it enables fully asynchronous progress without any user interaction. There are essentially two possible models for implementing the thread-based progress: 1) polling based; 2) interrupt-based. In the first model, a communication thread is dedicated to each MPI process, in order to handle incoming messages from other processes. Each thread polls the MPI progress engine and gets the data immediately. On the other hand, the interrupt-based approach uses hardware interrupts to wake up a thread and make communication progress exactly at the right time. Even though this approach does not waste resources by continuously polling the network, it suffers from substantial overhead introducted by the OS intervention.

Offloading the protocol handling to the communication hardware (hardware offload), is a very powerful alternative to ensure fully asynchronous progress. Anyway, delegating more work to the communication hardware (into the NIC) can easily become

a performance bottleneck, due to the lower performance of the embedded processors compared to regular CPUs.

In [88], Hoefler et al. describe and analyze the thread-based approach. They conclude that thread-based progress based on polling (bypassing the OS) is beneficial only when separate computation cores are available for the progression thread. Using an interrupt-based approach (passing through the operating system) might be helpful in the case of oversubscribed nodes (the progress and user threads share the same core). Anyway, passing through the operating system raises two concerns: 1) it seems unclear how big the interrupt latency and overheads are on a modern systems; 2) the scheduler has to schedule the progress thread right after the interrupts arrive so as to achive asynchronous progress. This second issue can be faced by using real-time functionalities in the Linux kernel.

In [89], Si et al. propose to use dedicated communication processes (called ghost processes) for managing inter-node data transfers using two-sided communication. Once the data is received on the ghost process(es), it is delivered to the destination process using the MPI-3 shared memory capability. This mechanism ensures asynchronous progress on every process without incurring in any issue related to the use of threads.

In [90], we report the most significant contributions about computation/communication overlapping in MPI and provide technical explanation of how this overlap can be achieved on modern supercomputers.

In Chapter 6, we present OpenCoarrays, accompanied with a deep performance analysis of the two versions available, LIBCAF_MPI and LIBCAF_GASNet, implemented on top of MPI-3.0 and GASNet, respectively. At the time of writing, LIBCAF_MPI provides the widest coverage of the coarray features included in the Fortran 2008 standard and in Technical Specification 18508. Message progression is totally delegated to the MPI implementation, using a communication thread, and/or hardware offload.

6

OpenCoarrays

Contents

OpenCoarrays is an open-source software project for developing, porting and tuning transport layers that support coarray Fortran compilers. It targets compilers that conform to the coarray parallel programming feature set specified in the Fortran 2008 standard. It also supports several features proposed for Fortran 2015 in the draft Technical Specification TS-18508 "Additional Parallel Features in Fortran" [77]. OpenCoarrays uses a 3-clause BSD-style open-source license to facilitate its incorporation into free and proprietary compiler software and it is currently used by the GNU Fortran compiler.

At the time of this writing, OpenCoarrays is composed by three parts: 1) compiler wrapper; 2) run-time library; 3) executable file launcher.

The compiler wrapper checks if the actual compiler supports OpenCoarrays (currently only GCC 5 and above), in this case it simply passes the source code to the actual compiler without any modification. Otherwise, the wrapper transforms the coarray syntax into OpenCoarrays procedure calls before invoking the actual compiler on the transformed code. More details about the compiler wrapper are provided in Section 6.1.

The run-time library supports compiler communication and synchronization requests by invoking a lower-level communication library (MPI by default). Two run-time libraries, one based on MPI and one based on GASNet [91], were realized by myself during a six months visiting period at National Center for Atmospheric Research in Boulder, Colorado. More details about the run-time libraries are provided in Section 6.2.

The file launcher passes execution to the chosen communication library's parallel program launcher (mpirun by default).

In the rest of this chapter, we present the OpenCoarrays run-time library used by the GNU Fortran compiler and report a detailed performance comparison against the

most relevant, proprietary, compilers supporting coarray Fortran: the Intel and Cray compilers.

6.1 OpenCoarrays Compiler Wrapper

The main aim of the compiler wrapper is to support CAF even on compilers that provide limited or no support for CAF. To do so, if the compiler wrapper detects a non-OpenCoarrays-aware compiler, the source code is parsed and any occurrence of a coarray operation is replaced by a specific procedure call implemented in a Fortran 2008 module. These procedures adapt the arguments coming from the source code and invoke the run-time library using the OpenCoarrays ABI. This simple approach allows one to invoke the OpenCoarrays functions from any compiler compliant with Fortran 2008.

Currently, only a subset of the coarray functionalities are supported by the wrapper module. Anyway, this approach allowed us to run CAF programs on Intel Xeon Phi, using coarray features not supported by the Intel compiler (atomics defined in TS-18508). More details about this work are reported in Chapter 7.

6.2 OpenCoarrays Run-Time Library

OpenCoarrays defines an application binary interface (ABI) that translates high-level communication and synchronization requests into low-level calls to a user-specified communication run-time library. This design decision liberates compiler teams from hardwiring communication-library choice into their compilers and it frees Fortran programmers to express parallel algorithms once, and reuse identical CAF source with whichever communication library is most efficient for a given hardware platform.

The run-time libraries provided by OpenCoarrays are named "LIBCAF_", followed by the specific communication layer used to implement the library. From now on, we will refer to the OpenCoarrays run-time library based on MPI and GASNet as LIBCAF_MPI and LIBCAF_GASNet, respectively.

In the rest of the chapter we present an in-depth performance comparison of several coarray implementations on various hardware platforms, using OpenCoarrays along with the GNU Fortran compiler. All the results shown in this chapter have been presented in our work [92].

6.2.1 GNU Fortran and LIBCAF

GNU Fortran (GFortran) is a free, efficient and widely used compiler. Starting in 2012, GFortran supported the coarray syntax and the single image execution but it did not provide a multi-image support. The main idea was to delegate the communication effort to an external library (LIBCAF) and to be agnostic to the actual implementation of the library calls. This means that GFortran generates function calls to the external library whenever encounters coarray's operations. Having an external library allows it to switch coarray implementations without modifying the compiler code.

LIBCAF_MPI Since the very first release of OpenCoarrays (August 2014), the widest coverage of coarray features was provided by a MPI based run-time library (LIBCAF_MPI). Because of one-sided nature of coarrays, 99% of the run-time library uses MPI one-sided communication routines with passive synchronization.

LIBCAF_MPI currently supports:

- coarray scalar and array transfers (efficient strided transfers);
- synchronization (sync all, sync images, sync memory);
- atomics;
- critical;
- locks;
- events;
- collectives (co_sum, co_max, co_min, etc...).

Despite the good matching of coarray one-sided semantics and MPI one-sided routines, it should be noted that the behavior of some MPI routines may differ from the CAF counterpart. A typical example is the difference between *MPI_Get* and getting data from a remote coarray variable. For *MPI_Get*, the function calls returns before the data arrives; the programmer can only assume that the operation has completed after a synchronization call (like *MPI_Win_Flush*). For coarrays, the Fortran semantics related to a variable assignment has to be respected; this means that the programmer can assume that the data has arrived as soon as the read operation returns.

LIBCAF_GASNet LIBCAF_GASNet is still an experimental version. It is intented for an expert usage but it provides higher performance than LIBCAF_MPI. GASNet stands for Global Address Space Networking and is provided by UC Berkeley [91]. GASNet provides efficient remote memory access operations, native network communication interfaces and useful features like Active Messages and Strided Transfers (still under development). The major limitation of GASNet for a coarray implementation consists in the explicit declaration of the total amount of remote memory required by the program. Thus, the user has to know, before launching the program, how much memory is required for coarrays. Since a coarray can be static or dynamic, a good estimation of such amount of memory may not be easy to guess. A memory underestimation may generate sudden errors due to memory overflow and overestimations may require the usage of more compute nodes than needed.

Currently, LIBCAF_GASNet supports only coarray scalar and array transfers (including efficient strided) and all the synchronization routines.

In this work, we provide only a partial analysis of this version.

6.3 Coarray Comparison

In this section, we present a comparison between the GFortran coarray implementation and the one provided by the proprietary compilers from Cray and Intel. LIBCAF_MPI is the most deeply analyzed version; however we also provide some results about the LIBCAF_GASNet version on the Cray machines.

6.3.1 Test Suite

In order to compare LIBCAF with the other compilers, we ran several test cases:

- EPCC CAF Micro-benchmark suite;
- Burgers Solver;
- CAF Himeno;
- 3D Distributed Transpose;

6.3.1.1 EPCC CAF Micro-benchmark Suite

The EPCC CAF Micro-benchmark suite [93] has been provided by University of Edinburgh and its source code is freely available on the web. It measures the performance (latency and bandwidth) of the basic coarray operations (get, put, strided get, strided put, sync) and of a typical communication pattern: the halo exchange. Every basic operation is analyzed in two different scenarios: single point-to-point and multiple point-to-point. In the first case, image 1 interacts only with image n; every other image waits for the end of the test. In this scenario, there is no network contention to expect; hence, it represents a best case scenario. During the multiple point-to-point, image i interacts only with image i+n/2; this test case models what actually happens in real parallel applications. In this section, we present only a comparison on these two scenarii; we do not report results on the performance of synchronization operation and halo exchange. The synchronization time has a smaller impact on overall performance than the transfer time and the performance of halo exchange is considered in the real application tests like CAF Himeno.

6.3.1.2 Burgers Solver

The Burgers Solver has been provided by Damian Rouson and the source code is freely available [94]. This benchmark case has a 87% weak scaling efficiency on 16384 cores and linear scaling (sometimes super-linear). It uses coarrays mainly for scalar transfers

and the destination images of that transfers are neighbour images which are usually placed on the same node.

6.3.1.3 CAF Himeno

CAF Himeno has been initially written as a parallel application by Prof. Ryutaro Himeno using OpenMP and MPI. Afterwards, it has been translated to an early Cray implementation of coarrays by Bill Long (Cray) and finally the Cray version has been translated to standard coarrays by Dan Nagle. The test implements a 3D Poisson relaxation which uses the Jacobi method. It uses coarrays for strided array transfers using the usual Fortran array syntax.

6.3.1.4 3D Distributed Transpose

This program has been provided by Bob Rogallo. It implements a distributed 3D transpose and it is available in coarray and MPI versions; a comparison between these two versions has been provided.

6.3.2 Hardware and Software

In order to compare the Cray and Intel implementations with our new coarray support, we ran each test case on several HPC cluster provided by several organizations. The Intel CAF implementation needs to have the Intel Cluster Toolkit installed on the machine, while the Cray compiler works only on a proprietary architecture. GFortran, based on a MPI communication library like LIBCAF_MPI, can be executed on any machine able to compile gcc and any standard MPI implementation. The hardware available for our analysis is the following:

- Eurora: Linux Cluster, 16 cores per node, Infiniband QDR 4x QLogic (CINECA);
- PLX: IBM Dataplex, 12 cores per node, Infiniband QDR 4x QLogic (CINECA);
- Yellowstone/Caldera: IBM Dataplex, 16 cores per node (2 GB/core), FDR Infiniband Mellanox 13.6 GBps (NCAR);

- Janus: Dell, 12 cores per node, Infiniband Mellanox (CU-Boulder);
- Hopper: Cray XE6, 24 cores per node (1.3 GB/core), 3-D Torus Cray Gemini (NERSC);
- Edison: Cray XC30, 24 cores per node, Dragonfly Cray Aries (NERSC);

In this work, we present only the results collected from Yellowstone and Hopper/Edison, mainly because they provide the best configuration for using Intel and Cray coarrays.

In particular, the comparison between GFortran and Intel has been run on Yellowstone and the comparison between GFortran and Cray on Hopper/Edison.

In order to validate, the results we ran the tests on the remaining machines listed above.

6.3.2.1 On Hopper and Edison

Hopper allows us to run coarray programs only with Cray and GFortran.

For the Cray compiler, we used the 8.2.1 version with the -O3 flag, loaded the craype-hugepages2M module and set the XT_SYMMETRIC_HEAP_SIZE environment variable properly (this variable will set the size of the symmetric heap used by the runtime system). For GFortran, we used the GCC 4.10 development version (GCC trunk) and Mpich/7.0.0 (installed on the machine) for LIBCAF_MPI. The only flag applied on GFortran was -Ofast (for optimization). The GCC-4.10 experimental is almost the same version present nowadays on the gcc-trunk (no performance changes).

Edison has been used only for the EPCC CAF micro-benchmark; we used the Cray compiler version 8.3.0 with the same configuration set on Hopper.

6.3.2.2 On Yellowstone

Yellowstone allows us to run only coarray programs compiled with Intel and GFortran.

We used the Intel compiler 14.0.2 and IntelMPI 4.0.3 need for coarray support. The following flags have been applied during the compilation: -Ofast -coarray -switch no_launch. For gfortran, we used the GCC 5.0 development version (same used for

Cray) and MPICH IBM (optimized for Mellanox IB) for LIBCAF_MPI. Also in this case, the only flag applied on GFortran was -Ofast.

6.4 Results

In this section, we present an exhaustive set of tests performed on single and multiple nodes of Yellowstone and Hopper/Edison. Single node means that the network which connects the cluster nodes is not involved; such a configuration can be useful to understand the performance of the communication libraries in a shared memory environment.

6.4.1 EPCC CAF - GFortran vs. Intel

The EPCC CAF micro-benchmark suite provides latency and bandwidth for several block sizes during the basic CAF operations (get, put, strided get, strided put). EPCC tests two scenarios: single point-to-point, where there is no network contention involved, and multiple point-to-point, for a more realistic test case. In this section, we present the results of GFortran vs. Intel on single and multiple node only for the put and strided put operations (get has almost the same results). This particular comparison, GFortran vs. Intel, can be run only on Yellowstone/Caldera.

6.4.1.1 Single point-to-point Put on single node

Since Yellowstone has 16 cores per node, only a single node has been used.

Figures 6.1 and 6.2 show that, on a single node, Intel is better than MPI for quantities less than or equal to 4 doubles (32 bytes). After that point the latency assumes a linear trend for Intel and stays constant for GFortran. The bandwidth, after 4 doubles, has exactly an inverse behavior: linear for GFortran and constant for Intel. In other words, for small transfers (in particular scalars) within the same node, without contention, Intel has better performance than GFortran.

Figure 6.3 and 6.4 show that increasing the block size does not change the trends observed on small block sizes.

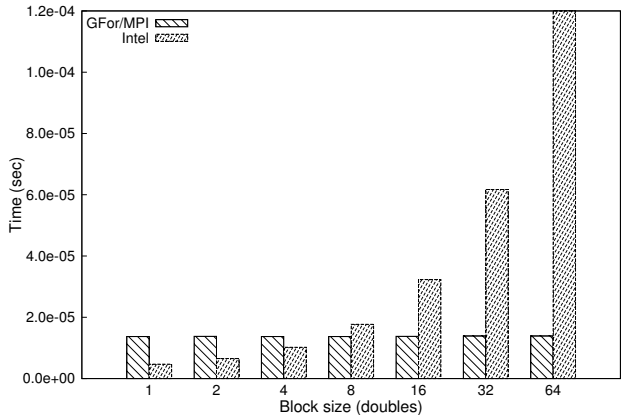


Figure 6.1: Latency Put small block size - Yellowstone 16 cores

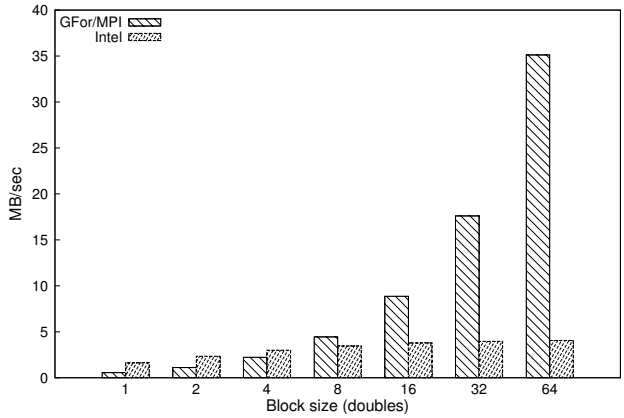


Figure 6.2: Bandwidth Put small block size - Yellowstone 16 cores

6.4.1.2 Multi point-to-point Put on single node

In this configuration, image i interacts only with image i+n/2 (where n is the total number of images). For this case, we show only the bandwidth in Figure 6.5.

This test case shows that Intel is less affected by network contention than GFortran. In this particular case, Intel has a bigger bandwidth than GFortran, for values less than

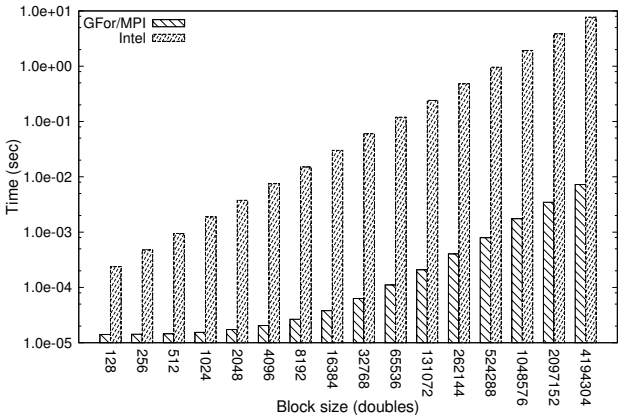


Figure 6.3: Latency Put big block size - Yellowstone 16 cores

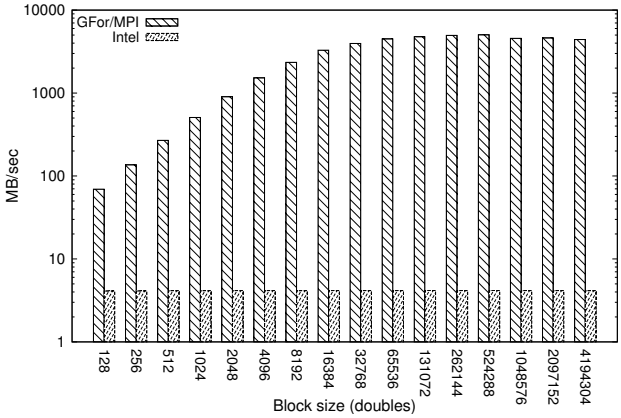


Figure 6.4: Bandwidth Put big block size - Yellowstone 16 cores

or equal to 8 doubles (64 bytes).

A good way to see this phenomenon is to chart the bandwidth difference between single and multiple point-to-point. Figure 6.6 shows this comparison, and we can see that Intel has a constant behavior. Intel is not very sensitive to network contention, which means that the network is not the bottleneck for Intel. For GFortran, we see a quite different behavior and its trend means that GFortran is limited by the network.

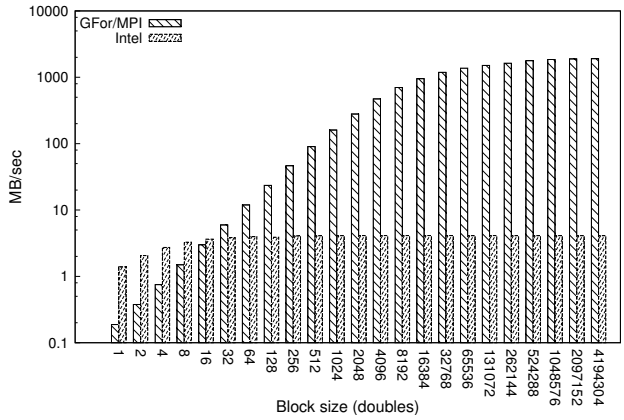


Figure 6.5: Bandwidth multi pt2pt Put - Yellowstone 16 cores

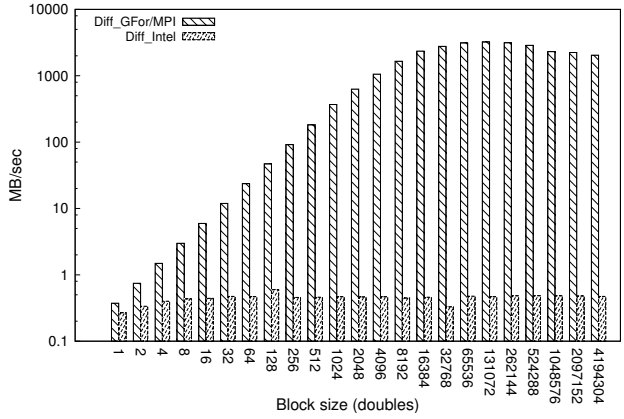


Figure 6.6: Bandwidth difference between single and multi - Yellowstone 16 cores

6.4.1.3 Single point-to-point Strided Put on single node

By strided transfer, we mean a non-contiguous array transfer. This kind of transfer is common in several scientific applications and its performance is therefore crucial for the performance of the entire application. This test is also very useful to understand the behavior of Intel. LIBCAF_MPI provides the support for efficient strided transfer;

which can be disabled by sending array elements one at time. Figure 6.7 shows the performance of strided transfer for LIBCAF_MPI and Intel; GFor/MPI_NS represents the performance of LIBCAF_MPI without strided transfer support (sending element-by-element). The most interesting fact is that, even with a contiguous array (stride = 1), Intel has the same performance as that of LIBCAF_MPI without strided transfer support. This fact shows that Intel sends array objects element-by-element even for contiguous arrays.

LIBCAF_MPI uses the classic MPI Data Type in order to implement an efficient strided transfer. This approach is very easy to implement but it is not very efficient either in terms of memory or time.

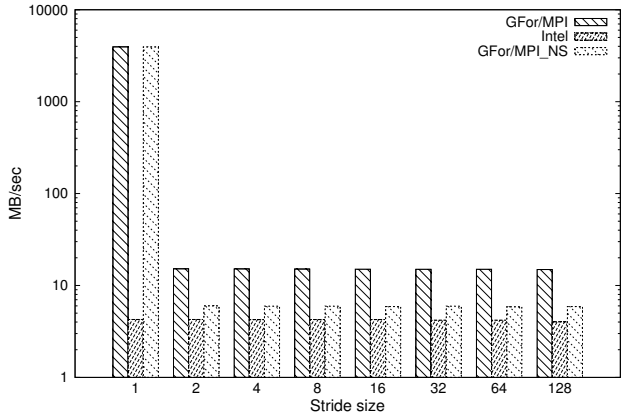


Figure 6.7: Strided Put on single node - Yellowstone 16 cores

6.4.1.4 Single point-to-point Put on multiple nodes

In this configuration, we ran the benchmark on 32 cores. The network which connects the cluster nodes is thus involved in the transfer. In this case, something unexpected happens: in Figure 6.8, Intel shows about 428 seconds of latency for transferring 512 doubles (4 KBytes) through the network. That strange behavior is related to the element-wise transfer.

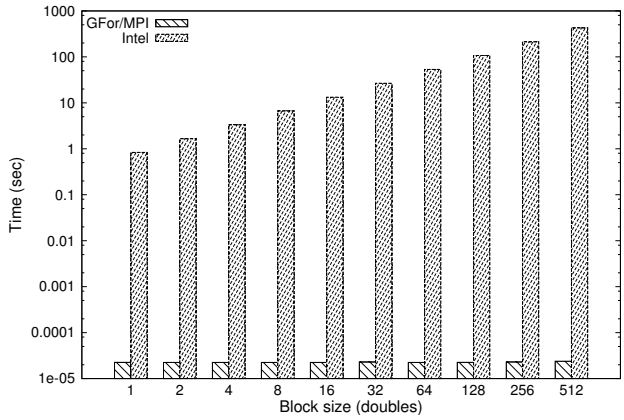


Figure 6.8: Latency on 2 compute nodes - Yellowstone 32 cores

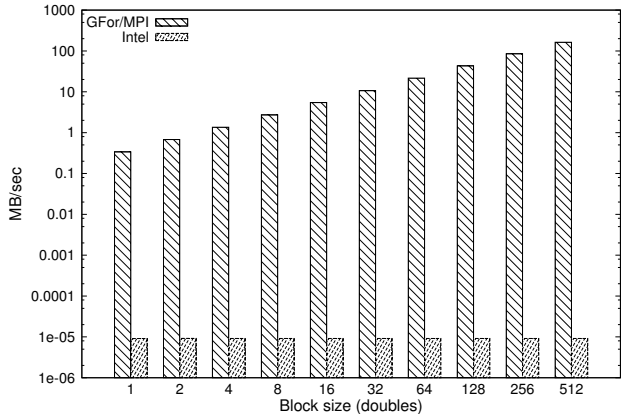


Figure 6.9: Bandwidth single Put on 2 compute nodes - Yellowstone 32 cores

6.4.2 EPCC CAF - GFortran vs. Cray

In this section, we present the results of GFortran vs. Cray. This particular comparison can be carried out only on Hopper and Edison. For this case, we report only the bandwidth results and we report also the GASNet results for EPCC and Distributed Transpose tests. Furthermore, we report only the results for the Get and Strided Get

operations. [1]

6.4.2.1 Single point-to-point Get on single node

Since Hopper and Edison have 24 cores on each compute node, only a single node has been used. Figure 6.10 shows that for small transfers, GASNet performs better than Cray on Hopper. For big transfers, Figure 6.11 shows that Cray is usually (but not always) better than the two LIBCAF implementations. Figure 6.12 shows that, on Edison, LIBCAF_GASNet performs better than Cray for small transfers (like on Hopper) but with a bigger gap. Figure 6.13 shows that, for big sizes, MPI performs better that Cray and GASNet.

On single node, on Edison, Cray is always outperformed by one of the LIBCAF implementations.

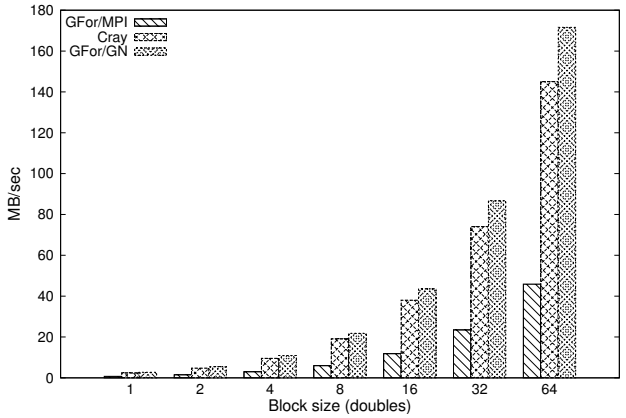


Figure 6.10: Bandwidth for small block sizes - Hopper 24 cores

[1] The performance of Get is similar to that of Put; anyway, we do not show the results for Put because of RMA problems when data package size exceeds a protocol-type-switching threshold.

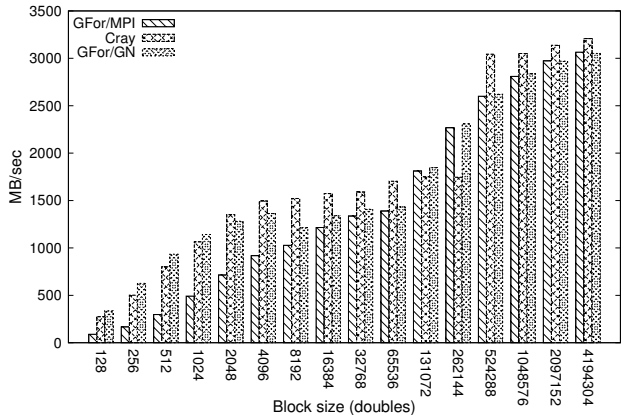


Figure 6.11: Bandwidth for big block sizes - Hopper 24 cores

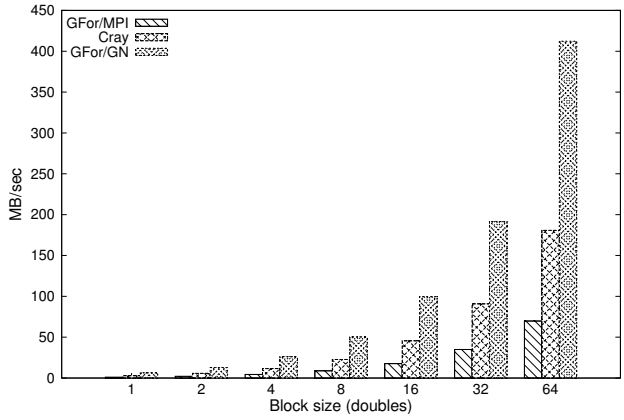


Figure 6.12: Bandwidth for small block sizes - Edison 24 cores

6.4.2.2 Multi point-to-point Get on single node

We analyze the behavior of LIBCAF_MPI, LIBCAF_GASNet and Cray with contention on the underlying network layer. Figure 6.14 shows that, in almost every case, GASNet evidences better performance than Cray when we have contention on the underlying network layer on Hopper. In Figure 6.15, we see that LIBCAF_GASNet has better

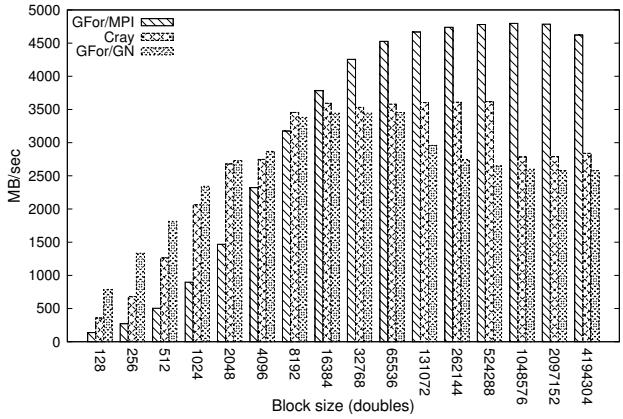


Figure 6.13: Bandwidth for big block sizes - Edison 24 cores

performance than Cray for small block sizes on Edison.

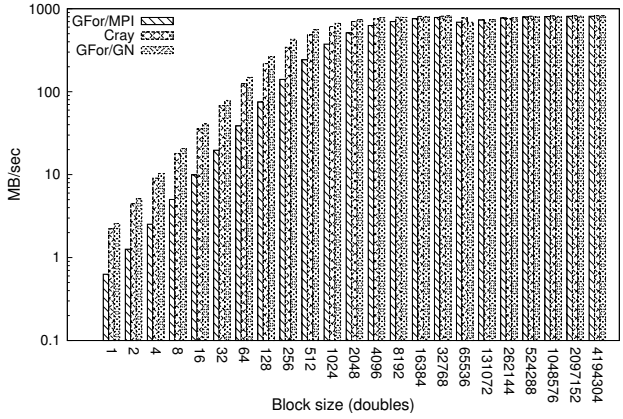


Figure 6.14: Bandwidth for multi pt2pt Get - Hopper 24 cores

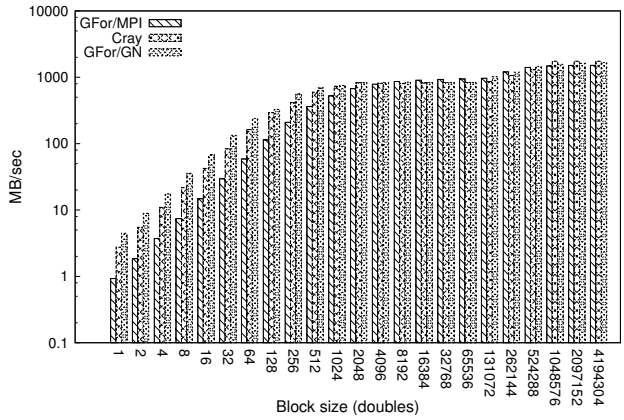


Figure 6.15: Bandwidth for multi pt2pt Get - Edison 24 cores

6.4.2.3 Single point-to-point Strided Get on single node

In Figure 6.16, we show the performance of the single point-to-point strided get on a single compute node of Hopper. In this case, Cray always shows the best performance. GASNet provides an experimental strided transfer support which is not yet optimized. The single stride has a size of 32768 doubles. In Figure 6.17, we see that LIBCAF_MPI is very effective on Edison. In fact, for several stride sizes, LIBCAF_MPI is better than Cray.

6.4.2.4 Single point-to-point Get on multiple nodes

Figures 6.18 and 6.19 show that on Hopper, on multiple nodes, Cray performs better than LIBCAF in almost every situation. GASNet shows a good behavior for small block sizes. Figures 6.20 and 6.21 show that, on Edison, LIBCAF_GASNet performs almost always better than Cray.

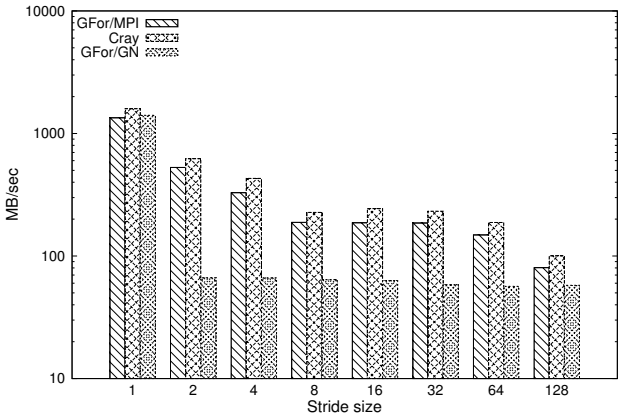


Figure 6.16: Strided Get on single node - Hopper 24 cores

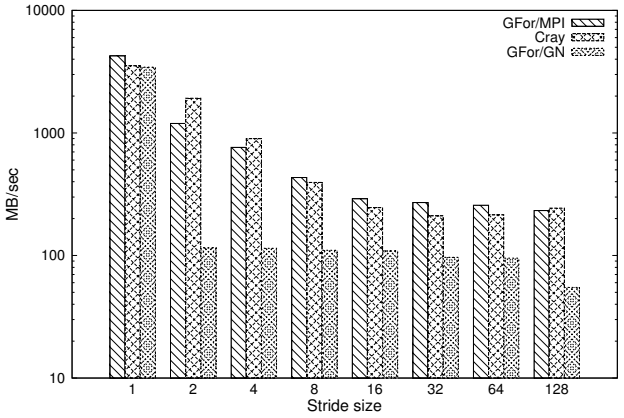


Figure 6.17: Strided Get on single node - Edison 24 cores

6.4.2.5 Multi point-to-point Get on multiple nodes

Figures 6.22 and 6.23 show the performance of Cray and LIBCAF when network contention occurs on multiple nodes. On Hopper, LIBCAF_GASNet has the best performance for small transfers (less than 512 doubles); on Edison, LIBCAF_GASNet shows good performance for transfers smaller than 16 doubles.

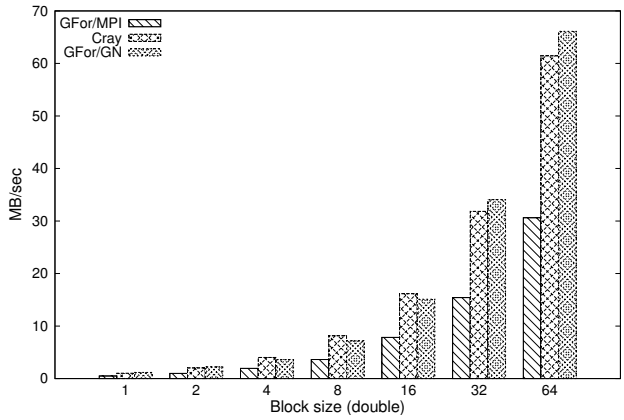


Figure 6.18: Bandwidth for small block sizes - Hopper 48 cores

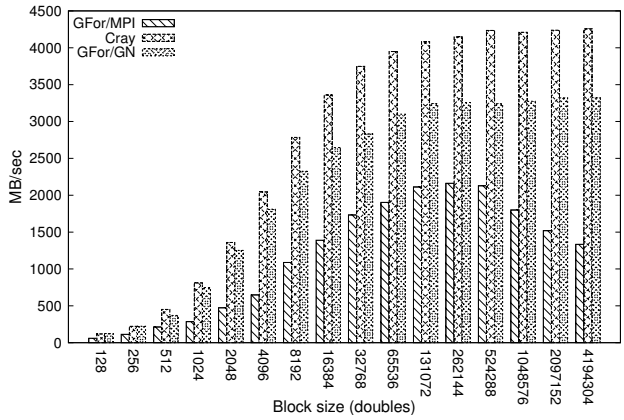


Figure 6.19: Bandwidth for big block sizes - Hopper 48 cores

6.4.2.6 Single point-to-point Strided Get on multiple nodes

The strided transfers on multiple node, for both Hopper and Edison, show an unexpected trend. Figures 6.24 and 6.25 show that LIBCAF_MPI yields better performance than Cray on both machines.

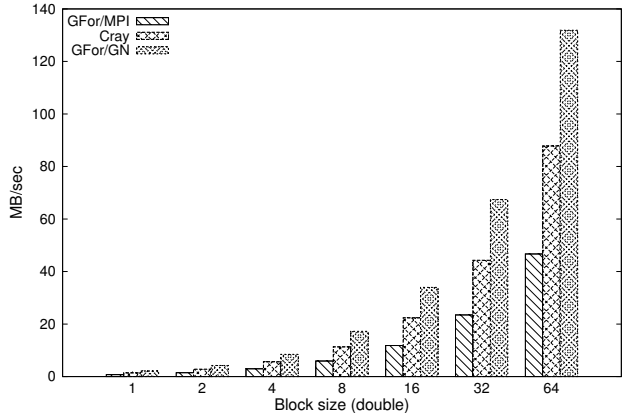


Figure 6.20: Bandwidth for small block sizes - Edison 48 cores

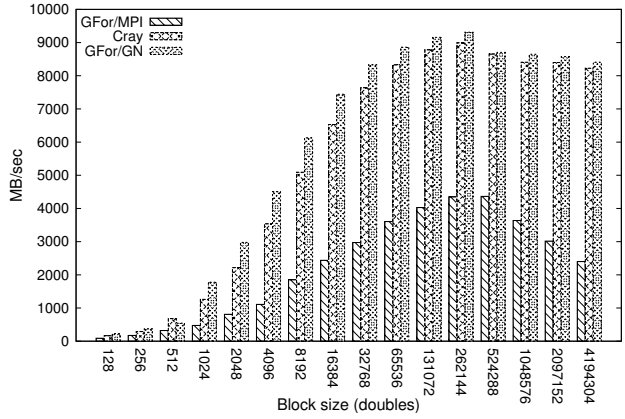


Figure 6.21: Bandwidth for big block sizes - Edison 48 cores

6.4.3 Burgers Solver - GFortran vs. Intel

The Burgers Solver is a real scientific application. It uses coarrays mainly for scalar transfers between neighbor images. In other words, this means that scalar transfers that usually take place within the same node. In Figure 6.26, we see that, on 16 cores (thus on one single compute node), Intel yields better performance than GFortran (as stated

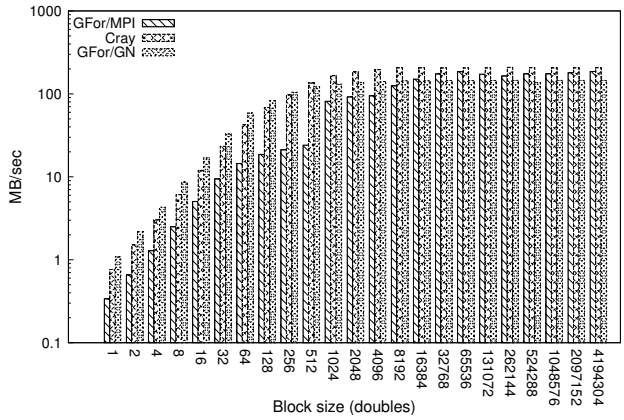


Figure 6.22: Bandwidth for multi pt2pt Get - Hopper 48 cores

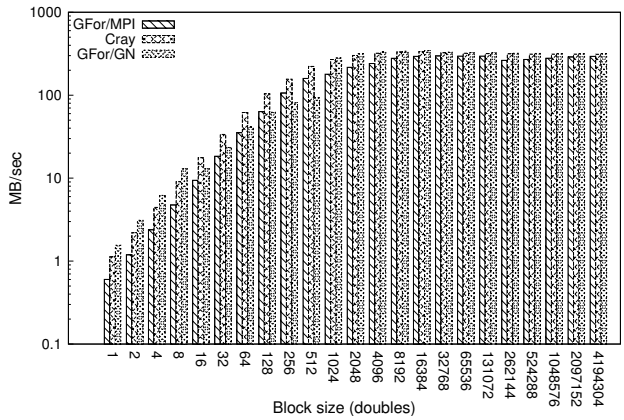


Figure 6.23: Bandwidth for multi pt2pt Get - Edison 48 cores

in Sections 6.4.1.1 and 6.4.1.2). On multiple compute nodes we see that GFortran is slightly better than Intel. The small difference is due to the fact that the communication takes place between neighbor images, which are usually placed on the same node.

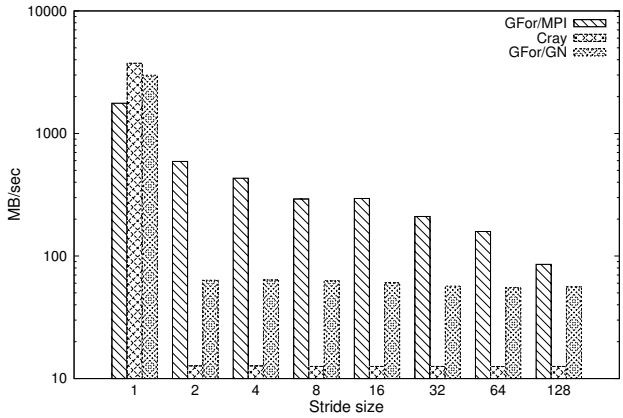


Figure 6.24: Strided Get on multiple nodes - Hopper 48 cores

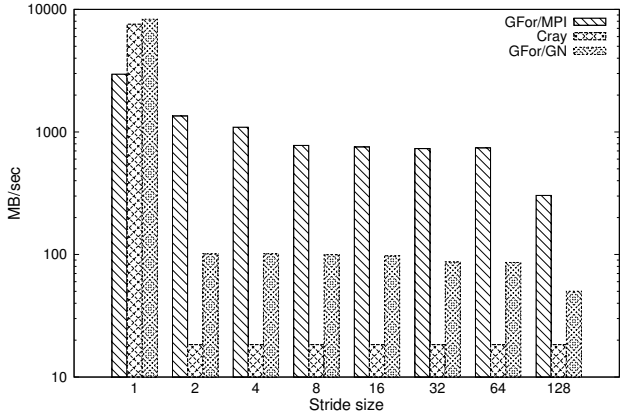


Figure 6.25: Strided Get on multiple nodes - Edison 48 cores

6.4.4 Burgers Solver - GFortran vs. Cray

In Figure 6.27, we report the results on Edison. In this case we have that GFortran evidences better performance than Cray. This test has been compiled with the `-O0` flag for both Cray and GFortran. Compiling the codes with `-O3` leads to the results shown in Figure 6.28, where Cray evidences better performance than GFortran mainly

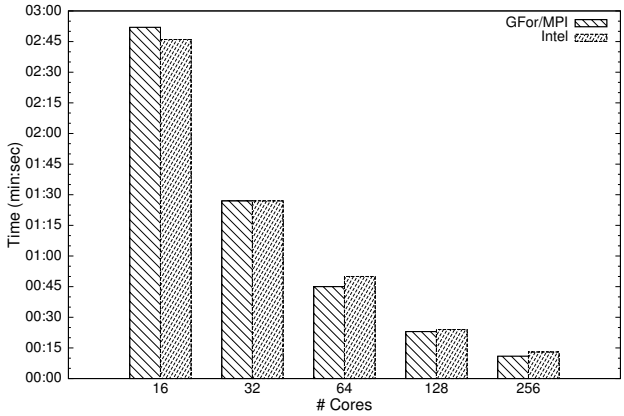


Figure 6.26: BurgersSolver GFortran vs. Intel

because of the better optimization capabilities (SIMD vectorization). This fact can be clearly observed in the last two groups of bars of Figure 6.28. For 128 and 256 cores, where the amount of computation per core is minimal and communication dominates the overall execution time, GFortran performs better than Cray.

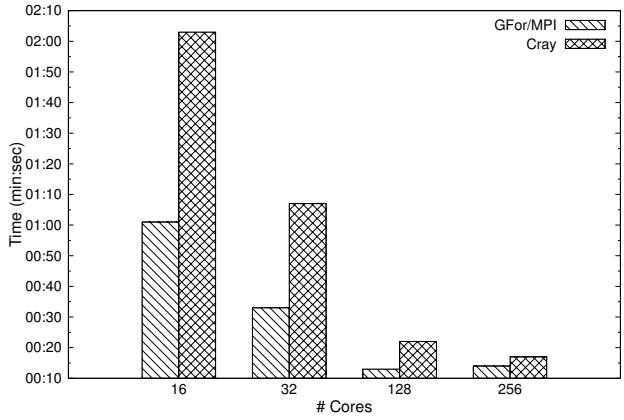


Figure 6.27: BurgersSolver GFortran vs. Cray - Optimization off

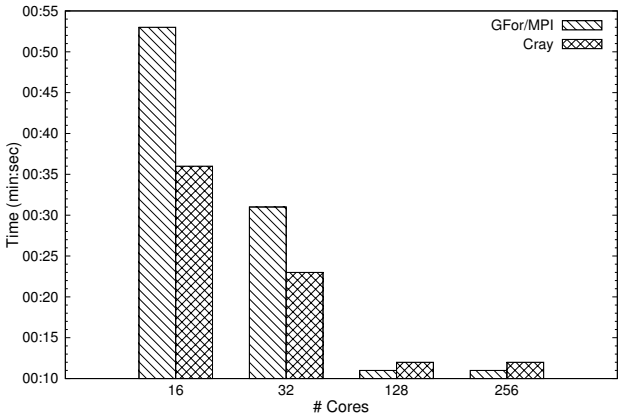


Figure 6.28: BurgersSolver GFortran vs. Cray - Optimization on

6.4.5 CAF Himeno - GFortran vs. Intel

CAF Himeno uses the Jacobi method for a 3D Poisson relaxation. The 3D nature of the problem implies strided transfers among several images. Using 64 cores (4 nodes of Yellowstone), Intel requires more than 30 minutes to complete. For this reason, we report in Figure 6.29 only the results for 16 and 32 cores. In this case, we report the MFLOPS on the y axis, thus higher value means better.

6.4.6 CAF Himeno - GFortran vs. Cray

The execution of CAF Himeno on Hopper required quite a bit of tuning. In order to run the 32 cores test, we were forced to place 8 images on each node due to memory constraints. Since each node has 24 cores, we wasted 16 cores on each node for memory reason.

The easiest and efficient coarray implementation to use for CAF Himeno was GFortran.

In Figure 6.30, we show the results for CAF Himeno on Cray.

Also in this case, Cray yields better performance than GFortran on single and multiple nodes.

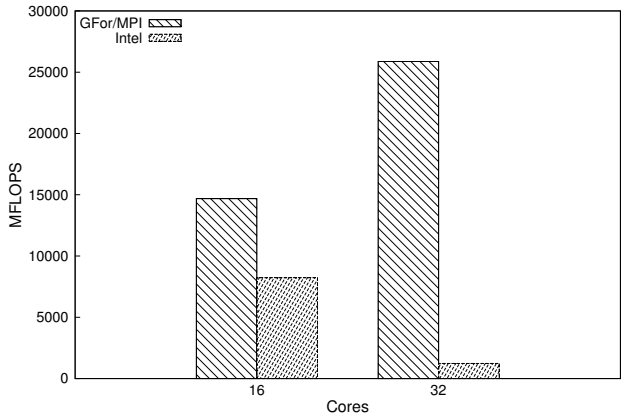


Figure 6.29: CAF Himeno - GFortran vs. Intel

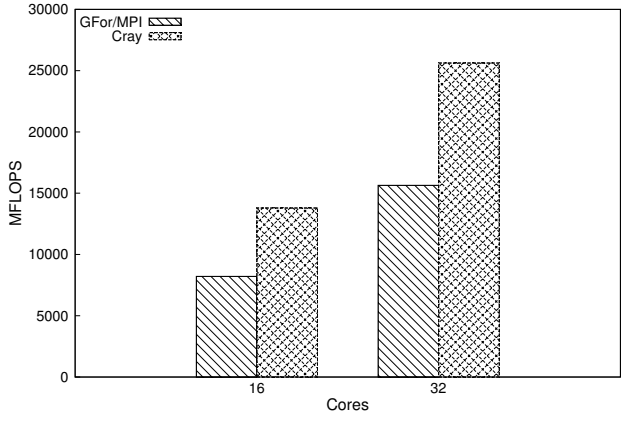


Figure 6.30: CAF Himeno - GFortran vs. Cray

6.4.7 3D Distributed Transpose - GFortran vs. Intel

This test case performs a distributed transpose of a 3D array.

In Figure 6.31 we see, again, that on a single node Intel is better than GFortran. On multiple nodes, the time required by Intel explodes because communication is spread across several processes among the nodes (and not only between images on the same

node).

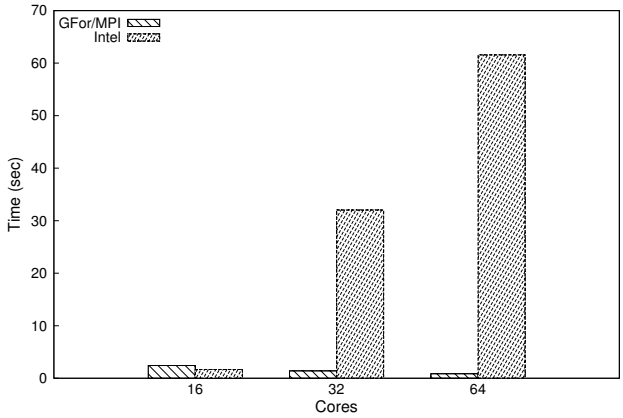


Figure 6.31: Distributed Transpose - GFortran vs. Intel

6.4.8 3D Distributed Transpose - GFortran vs. Cray

Only for this particular case we provide a comparison between the coarray version of GFortran and Cray with a pure MPI implementation. For this test case, we used only 16 of the 24 cores provided by Hopper. The reason is that the 3D matrix dimension has to be a multiple of the number of processes involved. We set this dimension to 1024x1024x512 elements.

Figure 6.32 shows that, within the same node (16 cores), GFortran with GASNet yields best performance, even better than Cray. On multiple nodes, Cray shows the best performance. Anyway, in any configuration, GFortran with LIBCAF_GASNet is better than a pure MPI implementation.

6.5 Conclusions

In this section, we present the conclusions of our investigation.

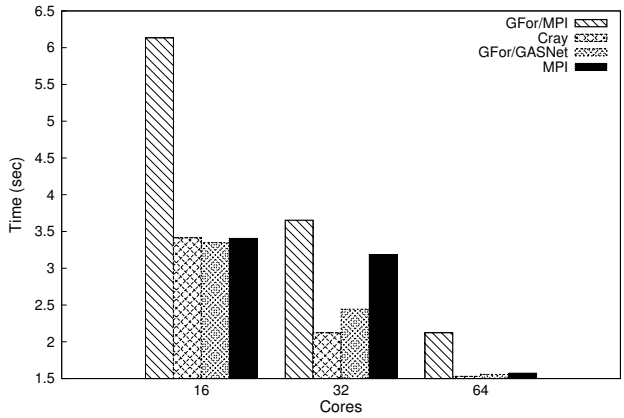


Figure 6.32: Distributed Transpose - Coarrays vs. MPI

6.5.1 Conclusions - GFortran vs. Intel

From our tests, we conclude that Intel shows better performance than GFortran only during small transfers within the same node. Since Intel sends one element at a time, the better performance for scalar transfers is probably related to the good performance provided by IntelMPI. On multiple nodes, the penalty of sending element-by-element becomes huge for Intel.

GFortran shows better performance than Intel on array transfers within the same node and in every configuration which involves the network.

6.5.2 Conclusions - GFortran vs. Cray

Hopper showed pretty good performance for big transfers on single and multiple nodes. The scientific codes we ran, show that Hopper provides the best performance for real applications.

On Edison, GFortran has always better performance, on single and multiple node, than Cray.

An unexpected result for us is the poor performance of Cray during strided transfers on multiple nodes.

Cray proposes a complete and efficient coarray implementation but it still requires some tuning in order to run the programs.

LIBCAF_GASNet shows good performance on Cray using the Gemini-conduit and Aries-conduit on Hopper and Edison, respectively.

6.5.3 Final Considerations

GFortran with LIBCAF_MPI provides a stable, easy to use and efficient implementation of coarrays. GFortran provides a valid and free alternative to privative compilers.

The most important fact is that the coarray support in GFortran can be used on any architecture able to compile GCC and which supports a standard MPI implementation.

This means that GFortran and LIBCAF_MPI can be used on Linux, Mac OSX or Windows without any limitation, without any cost but with remarkable performance.

Currently, on the compiler side, the multi-image coarray support is already available on GCC 5.1 and above.

At the time of this writing (October 2015), OpenCoarrays comes with a wrapper, implemented as a Fortran module, that exposes a subset of coarray functionalities (collectives and synchronization). By using this wrapper, any Fortran compiler is able to use the coarray functionalities provided by OpenCoarrays without supporting the coarray syntax.

6.5.4 Future Developments

For future developments, we plan to cover all the missing features expected by the standard and by TS-18508 and to use OpenCoarrays as a transport layer on languages other than Fortran. In particular, we are considering to implement the Coarray Python extension, proposed by Rasmussen et al. [95], using OpenCoarrays.

7

CAF on Heterogeneous Architectures

Contents

As already stated in Chapter 1, the exascale era will certainly represent a remarkable change in computer architectures. The high heterogeneity expected in future architectures will require to manage several tasks on different compute units using several memory types. In this scenario, a good parallel programming model should provide the programmer with a way to control at the software level resources that have a significant impact on performance, e.g. express data locality.

In this chapter, we merge the contributions exposed in Parts I and II by analyzing the behavior of coarray Fortran (and more generally of PGAS languages) when used with, as well as on, accelerators like GPUs and Intel MICs.

As a first contribution for merging GPUs and coarray Fortran, we present in Section 7.2 a new keyword for the Fortran language; this keyword aims at expressing the concept of "computational variable". This simple declaration suggests the compiler to store an "accelerated" variable in the memory level more suitable for computation. This simple mechanism allows to explicitly express data locality and to reduce data movement.

In Sections 7.3 and 7.4, we present our contributions on the other class of accelerators currently used in the HPC world: the Intel Xeon Phi. We describe a few simple load balancing algorithms with a detailed description of the difficulties encountered on the current hardware and the prospectives for the future.

7.1 Introduction to Intel Xeon Phi

In this section, we present a brief introduction to the Intel Xeon Phi architecture and the execution modes it offers. Most of the material presented in this section has been taken from [96] and [97].

Intel Xeon Phi coprocessors have been designed by Intel Corporation as a supplement to the Intel Xeon processor family. These coprocessors feature the Intel manycore architecture, which enables fast and energy-efficient execution of some High Performance Computing (HPC) applications. High energy efficiency is achieved through the use of low clock speed x86 cores with lightweight design suitable for parallel HPC applications. Therefore, only highly parallel applications supporting vectorized arithmetic with well-behaved (or negligible) memory traffic will thrive on the manycore architecture.

First generation Intel Xeon Phi coprocessors based on the Knights Corner (KNC) chip are end-point Peripheral Component Interconnect Express (PCIe) devices. They can be installed on the PCIe bus and operated in coprocessor-ready computing systems, including workstations and servers. An Intel Xeon Phi coprocessor cannot operate without a CPU-based host system, which is the reason for terming these products *coprocessors*. Because they reside on the PCIe bus and have their own on-board RAM, coprocessors do not share memory address space with the CPU. Consequently, the mere presence of a coprocessor in a system does not automatically improve the performance of applications running on the CPU. To utilize the MIC architecture, the application or the cluster execution manager must be aware of the presence of a coprocessor. The usage model of the second generation Intel MIC based on the Knights Landing (KNL) chip will be different. The second generation chip will be available as a standalone processor, as well as a PCIe-endpoint device. For the standalone processor version, applications need not be coprocessor-aware in order to be accelerated.

Because of the similarity of the manycore and multi-core architectures, an Intel Xeon Phi coprocessor can execute applications compiled from the same C/C++ or Fortran code as an Intel Xeon processor. Furthermore, Intel Xeon processors and Intel Xeon Phi coprocessors support the same parallel frameworks and require similar code optimization methods. This is a significant advantage of the Intel manycore architecture over computing accelerator technologies (GPGPUs and FPGAs). The process of application porting to GPGPUs typically involves re-writing from scratch the compute-intensive pieces of code.

Intel Xeon Phi coprocessors are Internet Protocol (IP)-addressable devices running

a Linux operating system (OS). This property enables straightforward porting of code written for the Intel Xeon architecture to the MIC architecture. This, combined with code portability, makes Intel Xeon Phi coprocessors a compelling platform for heterogeneous clustering. In heterogeneous cluster applications, host processors and MIC coprocessors can be used on an equal basis as individual compute nodes.

From the development maintenance point of view, having a single code for the main processor and for the coprocessor opens doors to heterogeneous computing and public code distribution. A heterogeneous application may utilize the CPU together with the MIC coprocessor, wasting no resources.

There are essentially three execution modes that can be used on the Intel Xeon Phi in association with the host processor:

- offload execution mode;
- native execution mode;
- symmetric execution mode.

With the offload mode, the host system is able to offload part or all of the computation to the Xeon Phi. This is the common execution model, adopted also on the GPUs.

As we said, an Intel Xeon Phi has a Linux OS and it can appear as another machine connected to the host, like another node in the cluster. It is possible to run parallel code (based on MPI and/or OpenMP) directly on the Intel Xeon Phi, without starting the application on the host. In order to do so, the application need to be cross-compiled for the Xeon Phi environment.

In symmetric mode, the application processes run on both the host and the accelerators at the same time. The accelerator is considered as a cluster node and communication is performed through MPI.

7.2 Hybrid Coarrays: PGAS GPU-to-GPU Communication

On hybrid clusters equipped with CPUs and GPUs, the most common way to exploit parallelism is through MPI for inter-node communication, and CUDA for GPU computation. This approach requires explicit data movement from/to GPU/CPU in order to send and receive data. In the latest evolution of both hardware and runtime libraries, this task has been either included in MPI GPU-aware implementations [98, 99] or performed by means of other technologies, like GPUDirect. In [100], we compared the performance of various manual data exchange strategies with a CUDA-aware MPI implementation using PSBLAS [49]; we concluded that the MPI CUDA-aware implementation is largely sensitive to data imbalance. A common strategy to exploit the computational power provided by the manycore devices is to use a hybrid approach, combining MPI and OpenMP (or a similar directive-based language) for inter- and intra-node communication, respectively. This approach is common in GPU clusters, where the inter-node communication is performed with MPI and the actual computation is performed with CUDA or OpenCL on the local GPU(s) [100].

An alternative to the MPI/OpenMP hybrid approach is to use a Partitioned Global Address Space (PGAS) model, as implemented for example by coarray Fortran (CAF) [69, 70] and Unified Parallel C (UPC) [71]. At this time, there are already publications [101, 102] on the usage of PGAS languages on Intel Xeon Phi (KNC architecture); even though the evidence is not conclusive, it is our feeling that PGAS languages will play an important role for the next generation of architectures. This is especially because, as already mentioned, on an exascale machine equipped with billions of computing elements, the bulk-synchronous execution model adopted in many current scientific codes will be inadequate.

7.2.1 Accelerated Keyword

In [32], we propose to merge the expressivity of coarray Fortran with the computational power of accelerators. As far as we know, this is the very first attempt to use

coarray Fortran with accelerators. The idea is to exploit the Unified Memory provided by CUDA 6.0 to make a coarray variable accessible from either the CPU or the GPU in a completely transparent way. The only changes required in OpenCoarrays are: (1) to separate the MPI window allocation and creation, and (2) to synchronize the CUDA device before using the memory. In MPI-2, the only way to create a window is to locally allocate the memory (via `malloc` or `MPI_Alloc_mem`) and then use the `MPI_Win_create` for the actual window creation, whereas with MPI-3 there is the option of a single call to `MPI_Win_allocate`. Our approach is to allocate the local memory using the `cudaMallocManaged` function in order to make that portion of memory "CUDA manageable", then call the `cudaSyncDevice` function, and finally create the window with `MPI_Win_create`. This approach is easy and general to implement, although it is not necessarily guaranteed to be the most efficient. A reasonable alternative would be either to delegate all communications to a CUDA-aware MPI implementation or to use a mapped memory approach, at the price of introducing a strong dependency on the quality of the MPI implementation. However, in our preliminary tests we found that the managed memory (Unified Memory) provided by CUDA 6.5 does not work too well with RDMA protocols (provided for example on Cray machines); we are fairly confident that this issue will be addressed in future CUDA implementations.

Using this approach, each coarray declared in the program requires interfacing with CUDA. What we suggest is a new variable attribute we call "accelerated". The meaning of this keyword is to mark a Fortran variable as "special", with faster access than a regular variable and suitable for accelerated computations.

In our current implementation, an "accelerated" variable is CUDA-accessible; note that it is not necessarily also a coarray variable. The keyword is not meant to replace OpenACC statements for CUDA allocations; it just suggests the compiler to treat the variable differently from usual variables.

As explained in Section 1.1.1.2, the Intel Knights-Landing will expose two types of memory: the first small and fast, the latter big and slow. Declaring a variable as "accelerated" would suggest the compiler that it could reside in the faster memory; in this case, the "accelerated" keyword assumes almost the same meaning as the "shared" key-

word of CUDA. To test these ideas, we modified GFortran by adding this new keyword as an extension, currently affecting only allocatable variables. For coarray variables, we modify the `_gfortran_caf_register` by adding one more argument representing the accelerated attribute. For non-coarray variables, we force the allocation through `cudaMallocManaged` using a new function called `_gfortran_caf_register_nc` implemented in LIBCAF_MPI.

7.2.2 Experimental Evaluation

To show the benefits of hybrid coarrays, we analyze in this section the performance of a matrix-matrix multiplication kernel based on the SUMMA algorithm [103]. We run the tests on Eurora, a heterogeneous cluster provided by CINECA, equipped with Tesla K20 and Intel Xeon Eight-Core E5-2658. We used the pre-release GCC-6.0, with OpenCoarrays 0.9.0 and IntelMPI-5. This unusual combination is because IntelMPI is the best MPI implementation provided on Eurora; however, OpenCoarrays can be linked with any MPI-3 compliant implementation. On a cluster of GPUs, the most commonly used approach consists in employing MPI for communication among GPUs, assuming that each process uses only one GPU, and then calling the CUDA kernel on each process. This simple approach allows one to use several GPUs on the cluster but it may suffer from the synchronization imposed by the two-sided functions (MPI_Send, MPI_Recv) provided by MPI. In order to invoke the CUDA kernels from Fortran using GNU Fortran, we make extensive use of the C-interoperability capabilities introduced in Fortran 2003. A typical example of C interoperability for the dot product $a \cdot b$ performed with CUDA is the following:

```
interface
 subroutine memory_mapping(a,b,a_d,b_d,n,img) &
                    &bind(C, name="memory_mapping")
   use iso_c_binding
   real(c_float) :: a(*),b(*)
   type(c_ptr) :: a_d, b_d
   integer(c_int),value :: n
   integer(c_int),value :: img
 end subroutine memory_mapping
 subroutine manual_mapped_cudaDot(a,b,partial_dot,n) &
```

```
                    & bind(C, name="manual_mapped_cudaDot")
    use iso_c_binding, only : c_float,c_int,c_ptr
    type(c_ptr),value :: a,b
    real(c_float) :: partial_dot
    integer(c_int),value :: n
  end subroutine
 end interface
```

The two subroutines are interfaces for the C functions called `memory_mapping` and `manual_mapped_cudadot`. The first is used to map the memory previously allocated on the CPU for *a* and *b* onto the GPU; the function returns two C pointers called `a_d` and `b_d`, which represent pointers usable on the GPU. The latter is the wrapper for the actual computational kernel. It takes as input arguments the GPU pointers returned by the `memory_mapping` function. NVIDIA claims that Unified Memory, besides reducing code complexity, could also improve performance by transferring data on demand between CPU and GPU. There are already some studies [104] on Unified Memory performance, that show these advantages to be strongly problem-dependent.

7.2.2.1 SUMMA Algorithm

SUMMA stands for Scalable Universal Matrix Multiplication Algorithm. The SUMMA algorithm, currently used in ScaLAPACK, is particularly suitable for PGAS languages because of the one-sided nature of the involved transfers.

Listing 7.1: Usual matrix product

```
do i=1,n1
  do j=1,n2
    do k=1,n3
      C(i,j) = C(i,j) &
               + A(i,k)*B(k,j)
    end do
  end do
end do
```

Listing 7.2: SUMMA approach

```
do k=1,n3
  do i=1,n1
    do j=1,n2
      C(i,j) = C(i,j) &
               + A(i,k)*B(k,j)
    end do
  end do
end do
```

Listings 1 and 2 allow to compare the pseudo-code for the usual matrix product with that of the SUMMA algorithm when we want to multiply matrices A and B, resulting in matrix C. SUMMA performs n partial outer products (column vector by row vector). This formulation allows to parallelize the two innermost loops on i and j.

Using MPI two-sided, each process has to post a send/receive in order to exchange the data needed for the computation; with coarrays, because of the one-sided semantics, each image can take the data without interfering with the remote image flow.

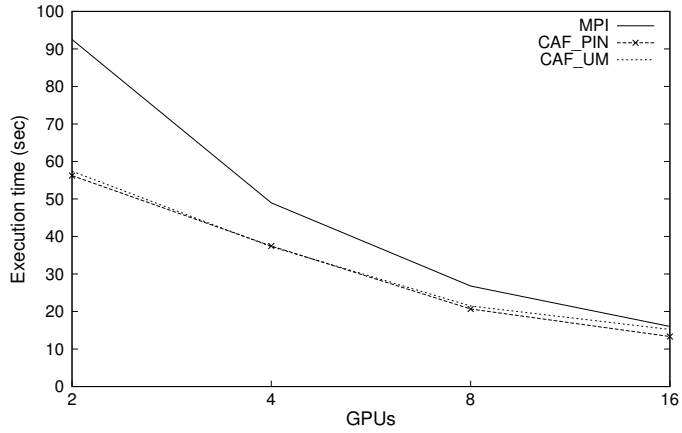


Figure 7.1: Performance of SUMMA: MPI-based vs. coarray Fortran implementations on Eurora cluster

Figure 7.1 compares the performance achieved by the coarray Fortran and MPI based implementations of the SUMMA algorithm. The chart shows the mean execution time on 10 runs using a matrix of size 4096x4096. We also report the performance of LIBCAF_MPI with CUDA support based on CUDA mapped memory as well as on Unified Memory, labeled as CAF_PIN and CAF_UM respectively. We observe that the performance achieved with Unified Memory is equal to or worse than that achieved with the usual pinned memory, as already noted in [104].

7.2.3 Conclusions on Hybrid Coarrays

In this section, we show how PGAS languages, and in particular coarray Fortran, can provide significant speedup in a hybrid CPU+Accelerator context. We show that using coarray Fortran, besides simplifying the code, improves performance because of the one-sided semantic which characterizes PGAS languages. We also propose a new

variable attribute, called "accelerated", for the Fortran language. Such attribute instructs the compiler to treat the variable as suitable for acceleration. Based on what we currently know about future architectures, we think that such a keyword can play a significant role in the post-petascale era, where heterogeneous code will be a must for exploiting all the computational power provided by complex and energy efficient architectures.

7.3 CAF-based Load Balancing Stategies on Intel Xeon Phi

Solving scientific problems, using multi- and many-core devices at the same time, possibly doing different types of computation, will be highly rewarded in the exascale era, where each compute node will be equipped with specialized and heterogeneous hardware. In this scenario, load balancing strategies, at different levels, will be critical for an effective usage of the heterogeneous hardware.

In this section, we present the performance results of a few dynamic load balancing strategies that have been implemented by exploiting coarray Fortran running on Intel Xeon Phi in the so-called *native mode*. *Native mode*, as opposed to *offload mode* means that a program is run directly on the Intel Xeon Phi architecture without host-to-coprocessor interaction.

So far, there have already been previous efforts to run PGAS languages on Intel Xeon Phi in native mode [101, 102]. They demonstrate that Phi-based systems can benefit from the exploitation of new programming models that allow one to overcome the communication wall problem. Anyway, as far as we know, this is the very first attempt of running a parallel application, based on coarray Fortran, on Intel Xeon Phi in native mode using dynamic load balancing strategies.

7.3.1 OpenCoarrays Wrapper

OpenCoarrays already provides complete support for coarray collectives, atomics, locks, critical section as well as events. In this work, we invoked the coarray *atomic* functions,

already implemented in OpenCoarrays, from the Intel Compiler using the wrapper module described in Section 6.1. Specifically, we use the new *ATOMIC_FETCH_ADD* function defined in Technical Specification 18508 not yet implemented in the Intel Compiler. This function is based on the MPI-3.0 *MPI_Fetch_and_op* function, which allows one to atomically fetch and update a remote variable.

Even though a user might want to use directly the MPI one-sided functions, the syntax of these functions is much more complex and error prone than coarray's. This allows the programmer to design and implement more easily complex parallel algorithms.

7.3.2 Test Case Description

In order to test the performance of OpenCoarrays on Intel Xeon Phi, we have run the following test cases:

- latency/bandwidth test;
- non work stealing test (NWS);
- process work stealing test (PWS);
- thread work stealing test (TWS).

7.3.2.1 Latency/Bandwidth Test

The latency/bandwidth test probes the network performance of the Intel Xeon Phi in native mode. This test represents a good indicator of the overhead paid bacause of the dynamic load balancing algorithm. In fact, a higher-latency/lower-bandwidth means that the time spent in doing *non computation* is higher than in a case with lower-latency/higher-bandwidth. The performance comparison is executed between the classic *MPI_Send/Recv* two-sided functions and the one-sided *MPI_Get* on which LIBCAF_MPI relies upon for simple data transfers. The test returns the average of 100 transfers of a fixed amount of data when there is contention on the communication layer. This test is significant because the MPI-based version of OpenCoarrays uses

MPI one-sided operations. Higher performance of the one-sided operations leads to higher performance of OpenCoarrays than the usual MPI two-sided approach.

7.3.2.2 Non Work Stealing (NWS)

To test the performance of such a simple homogeneous load balancing, we implemented a master-slave pattern, where one process dispatches and works on the data it owns and the other processes require a new task as soon as they have completed the previous one. In this case, each task requires the same amount of computation. In Sections 7.3.2.3 and 7.3.2.4, we analyze the case where some tasks require more time to complete. The test is designed as a hybrid MPI/CAF + OpenMP parallel approach. Four processes, where one represents the boss and the others the workers, have only one computational loop parallelized with OpenMP. The number of available cores (up to 240 on KNC) are equally partitioned among the four processes. The work is represented as a task index that only the boss owns, whereas the data to be used by each worker are already local and do not need to be sent. The computational part is represented by several arithmetic operations repeated for a fixed amount of iterations parallelized with OpenMP. Once a single task has completed, the worker keeps the results in its own memory and asks the boss for one more task to perform. The program ends when the boss runs out of tasks. The scope of this test case is to compare the performance of an MPI two-sided based implementation with a CAF based version.

MPI Based An easy and efficient MPI two-sided implementation of this test case consists in sacrificing one thread on the boss process and keeps it spinning in a loop with a blocking *MPI_Recv*; this blocking function waits for packets from any source. Once a packet arrives to the boss process it: 1) increases the task index; 2) sends the updated index to the worker; 3) calls *MPI_Recv* in order to serve the next request. When the task index hits the upper bound, the boss sends the value -1 to the next workers which will stop their execution.

CAF Based With coarrays, each process can make a portion of its own memory accessible to the others. Each process can get or send data from/to a remote memory

area without the involvement of the target process. This capability can be used by each worker to update the task index owned only by the boss process. In order to maintain consistency, the fetch of the current task index and the remote update have to be atomic operations. As we mentioned in 7.3.1, OpenCoarrays implements the coarray atomic function *atomic_fetch_add* which performs exactly this operation.

7.3.2.3 Process Work Stealing (PWS)

During the execution of a program like that described in Section 7.3.2.2, one (or several) processes may require more time than others. This happens, for example, in image processing, where some areas can express characteristics that require more computation than others. In this scenario, we assume that each process has assigned a static portion of the image to analyze; this means that the data to process are local to only one process. If a process receives an image portion which is computationally heavier than others, it will be the slowest, whereas the fastest processes will wait in an idle state, wasting energy and resources. A solution to mitigate this effect is to store all the data and the execution time in coarray variables on each process. In this way, each process is potentially a boss process able to provide work to others. When a process ends its tasks before the others, it checks the total execution time of each other process and restarts the computation by stealing work from the process with the longest time. This popular and simple load balancing technique based on the work stealing paradigm [105] can be implemented very efficiently with coarrays or any other PGAS language.

7.3.2.4 Thread Work Stealing (TWS)

In some cases "stealing" the computation from a process as described in Section 7.3.2.3 may be more efficient when performed at a finer granularity level by a higher number of threads on the same process. This technique can be useful when data transfers are very expensive in terms of time and/or energy. In this case, a faster process that ends its computation before the others provides all its threads to the slowest process. The operation and the algorithm are simpler than in the process-based counterpart; each process

has a coarray variable called *nThreads* that represents the number of threads actually used for computation by the process. When a faster process terminates its computation, it sums, using an atomic operation, its value of *nThreads* to the coarray variable on the slowest process. Each process calls the *omp_set_num_threads(nThreads)* subroutine at the beginning of the computation of each new task. Although this work stealing algorithm is simple in its formulation, it can be hard to implement efficiently using MPI two-sided routines. The one-sided semantic brought by coarrays allows one to implement this strategy with minimal impact.

7.3.3 Experimental Results

We first describe the experimental platform used for performance comparison and then analyze the results of the tests described in Section 7.3.2.

Each test reported has been run on Galileo, a Tier-1 system owned by CINECA, the Italian supercomputing consortium. Each compute node is equipped with two 8-cores Intel Haswell processors at 2.40 GHz. On more than a half of the available compute nodes, there are 2 Intel Xeon Phi 7120p. Each Xeon Phi has 61 cores at 1.1 GHz able to handle up to 4 threads and 8GB of RAM. We compiled each code using the Intel Fortran Compiler 15.0.2 and IntelMPI-5.0.2 and linked with OpenCoarrays-1.0.0 compiled for IntelMIC.

Latency and Bandwidth Test In the first test, we compare the latency and bandwidth achieved by the classic MPI two-sided functions and the one-sided OpenCoarrays implementation for increasing block sizes.

The lower latency and higher bandwidth of the one-sided versus the two-sided operations, shown by Figures 7.2 and 7.3, allow us to implement more efficient load balancing algorithms.

NWS Load Balancing In the next test, we analyze the performance of an MPI-based algorithm versus a CAF-based algorithm, organized according to the master-slave paradigm, where each task requires the same amount of computation. Figure 7.4

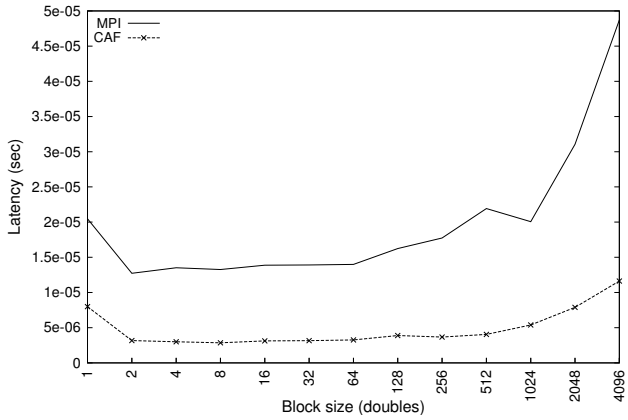


Figure 7.2: Latency on Xeon Phi using 60 cores

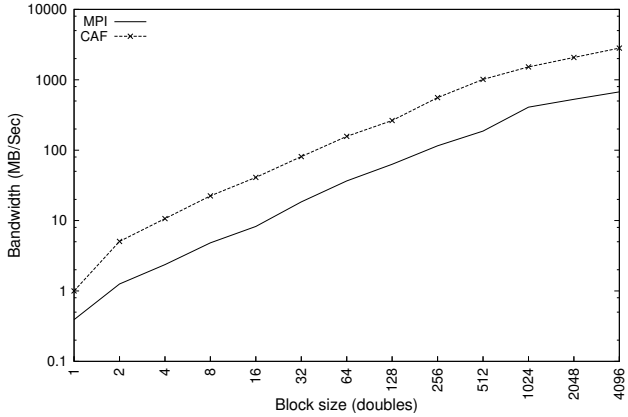


Figure 7.3: Bandwidth on Xeon Phi using 60 cores

shows that the CAF implementation is always slightly better than the MPI-based one using execution time as performance metric.

Figure 7.5 shows the unbalance factor λ, defined as:

$$\lambda = \left(\frac{L_{max}}{L} - 1\right) \times 100 \ ,$$

where L_{max} represents the maximum number of tasks computed by a process and L represents the ideal number of tasks to be computed.

For the test case analyzed in this section, better performance means to process all the tasks provided by the boss process as fast as possible. Because all the tasks require the same amount of time, a perfect load balancing strategy should dispatch an equal amount of tasks to each process. This means a perfect parallelization without any "bubble" in the execution flow of the processes.

The lower performance of the two-sided primitives affects the dynamic load balancing algorithm, producing an unbalanced usage of resources and incomplete parallelization.

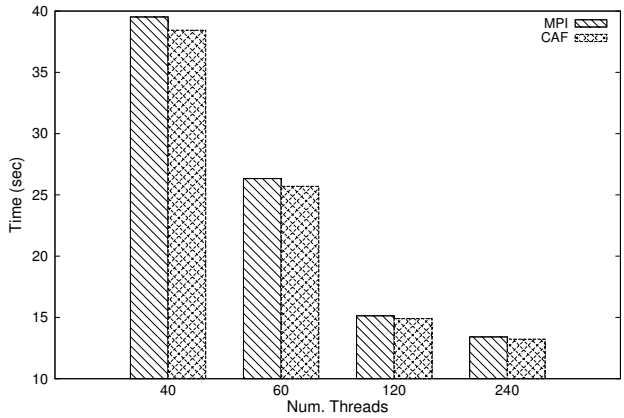


Figure 7.4: Execution time of NWS: MPI vs. CAF

PWS and TWS Load Balancing In the last test, we analyze the effects of process-based and thread-based load balancing while varying the number of threads involved in the computation. Figure 7.6 shows clearly that the process-based approach (labeled as *proc_bal* in the figure) is the most effective in any case, even when the number of requested cores is higher than the number of available cores. We also observe that the efficiency of the thread-based approach (labeled as *thread_bal* in the figure) decreases as the number of cores provided by the other processes increases. This fact does not

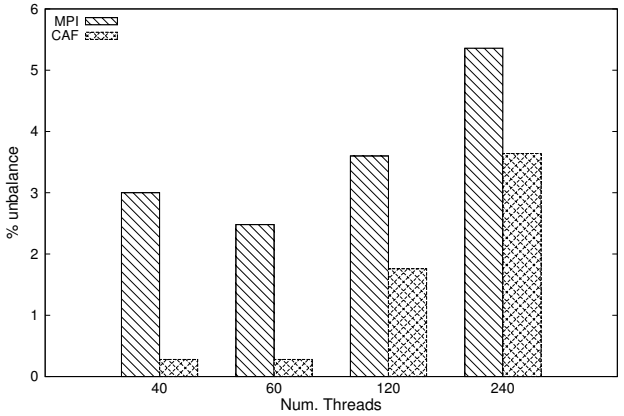


Figure 7.5: Unbalance factor: MPI vs. CAF

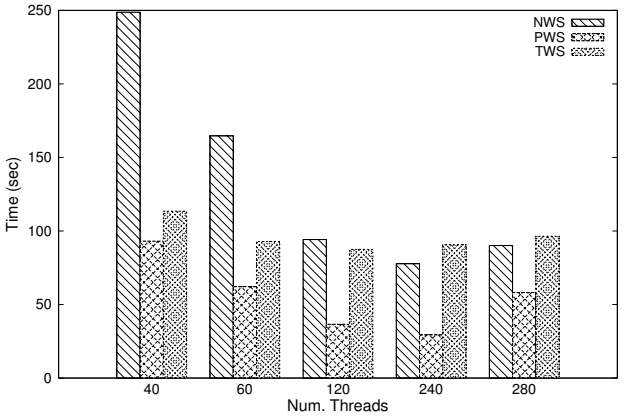


Figure 7.6: Comparison of execution time of load balancing strategies

surprise us; the effect of the overhead brought by the startup of new threads and their movement penalizes performance. Furthermore, the amount of data stored on each process in this particular test case (transferred across processes in the process-based version) is small. This reduces the advantage of moving threads across cores (generating affinity issues) compared to moving data across processes.

7.3.4 Conclusions

In this section, we have explored the behavior of coarray Fortran on Intel Xeon Phi running in native mode using OpenCoarrays. The asynchronous communication provided by coarrays and, more generally, by PGAS languages, allows one to implement dynamic load balancing algorithms that may be hard to implement with the usual MPI two-sided approach. We have shown a direct comparison between an MPI- and CAF-based implementation of a master-slave dispatcher for homogeneous tasks. Even in this simple case, the CAF-based implementation resulted in better performance and resource utilization. Furthermore, we have examined two dynamic load balancing strategies, based on coarrays, that allow one to address unexpected computational requirements due to the irregularity of the application data. Since, in native mode, network transfers are not as expensive as on an interconnection network, migrating threads on slower processes rather than stealing work from them turns out not to be a good practice. Anyway, this strategy can lead to good results when communication cost among processes is more expensive, in terms of energy and/or time.

As future work, we plan to explore the behavior of load balancing strategies on applications running in symmetric mode, using CPU and Xeon Phi at the same time. The presence of such heterogeneous compute units will certainly get benefits from the efficient and dynamic load balancing policies that coarrays (and PGAS languages) allow one to implement.

7.4 Heterogeneous Asian Options Pricing

In this section, we focus on load balancing a simple Monte Carlo simulation for the computation of the price of Asian options. Because of its ease of implementation and parallelization, this problem allows us to show pretty well the effects of load balancing when applied to heterogeneous compute units. The basic ideas and part of the underlying code, related to the Asian options pricing problem, have been taken from [96] with the kind permission from the authors. Most of the content presented in this section is part of [106].

In particular, we show how a PGAS-based approach can be truly effective for implementing a dynamic load balancing algorithm, with the ability to manage heterogeneous compute units. In order to simulate an exascale node equipped with heterogeneous components, we run all our tests on a single compute node equipped with 2 CPUs and 2 Intel Xeon Phi KNC in *symmetric mode*. This mode of operation provided by the Intel Xeon Phi allows one to run the same parallel application on both the CPUs and Intel Xeon Phis, as a regular MPI application.

We also show that running different versions of the same application, at the same time, on the right compute units, can make a difference in terms of performance. As far as we know, this is the very first attempt of running a parallel application, based on coarray Fortran, on CPU and Intel Xeon Phi in symmetric mode using dynamic load balancing strategies. In [101], the authors discuss the intra-node memory access problems and host-to-MIC connection issues for running UPC [71] applications on a MIC system under native and symmetric programming modes. They found out several significant problems for UPC running on many-core system like MIC, such as the communication bottleneck between MIC and host, the unbalanced physical memory, and computation power issues. They finally conclude that adopting a workload balancing strategy among MICs and CPUs can be a relevant optimization for running applications in symmetric mode.

7.4.1 Asian Option Pricing

An option is a contract between a buyer and a seller which allows one party to buy or to sell, on a future date, an asset from/to another party at a "strike price" agreed upon signature of the contract. The Asian options are a particular class of options in which the option payoff is calculated based on the mean price of the asset, sampled over a prespecified period of time [107]. This strategy reduces the risk associated with market volatility and short-term market manipulation. To make a profit, the seller of the option must set a price that offsets the anticipated risks associated with the asset price fluctuations. Asian options are commonly traded on currencies and commodity products which have low trading volumes.

Since there are no known closed form analytical solutions for pricing the Asian options, a variety of techniques have been developed to study this problem, resulting in a vast amount of related works. Popular techniques include Monte Carlo simulation, numerical inversion of the Laplacian transform of the Asian option price [108], numerical partial differential equation (PDE) techniques such as in [109], and various approximations, e.g., [110].

Using Monte Carlo simulation, multiple stochastic histories of the asset price are simulated based on the available information of the asset volatility [107,111,112]. Each Monte Carlo simulation is independent from the others and does not require intensive data transfers; therefore, this method can be categorized as *embarrassingly parallel*.

Suppose the task is to price N options, where for each option we have different sets of parameters. For each option, we will simulate P random paths and perform statistical analysis using these simulations. Adopting a parallel hybrid approach based on MPI+OpenMP, organized according to a boss-worker paradigm, there are two different ways to proceed:

1. compute T options in parallel on each process using OpenMP, each of which will run P simulations serially;

2. compute one option at a time on each process, running P simulations in parallel with OpenMP.

From now on, we will refer to the first approach as *MO* (multi-options) and to the second as *MT* (multi-threaded). In Figures 7.7 and 7.8 we provide a graphical description of the MO and MT approaches, respectively. Note how, for the MT approach, increasing the number of options on the device does not change the utilization of the internal compute units.

7.4.2 Load Balancing on Heterogeneous Nodes

Load balancing on heterogeneous nodes is a critical task in order to get high performance. The hardware heterogeneity makes it difficult to ensure reasonably uniform resource utilization, thus leading to performance losses due to load imbalance [113].

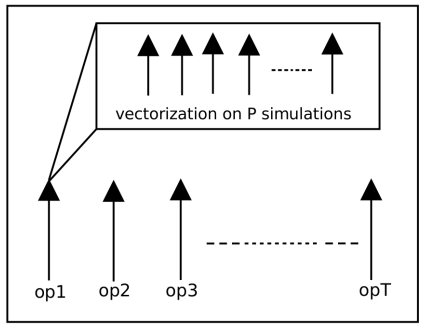


Figure 7.7: MO representation

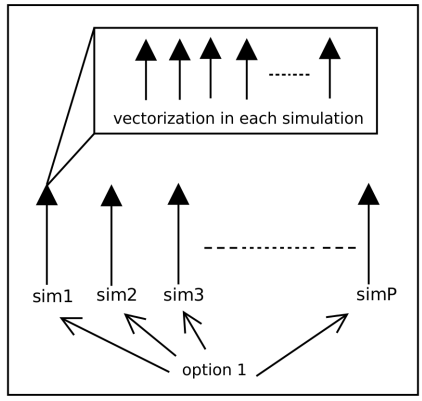


Figure 7.8: MT representation

The Asian option pricing problem we described in Section 7.4.1 is a very good candidate for showing the effects of good load balancing between CPUs and Xeon Phis. Since the options are independent from each other (*embarrassingly parallel*), the performance of the whole parallel application depends only on the efficiency and implementation of the load balancing algorithm.

Static load balancing can lead to very high performance only after several bench-

mark runs; these are needed to determine the correct ratio between CPU and Xeon Phi.

A dynamic load balancing approach relieves the users from performing such a preliminary tuning and allows to manage unexpected performance perturbations in a transparent way. This flexibility comes with the price of an increased implementation and communication cost and raises some questions about dynamic workload scheduling. In this section, we review how to realize a dynamic load balancing strategy based on the traditional MPI two-sided routines and then we focus on the exploitation of a CAF-based solution. The latter seems to be a valid alternative to MPI thanks to its light weight one-sided communication model and low overhead synchronization semantics.

7.4.2.1 Dynamic Workload Scheduling based on MPI

The most efficient version of dynamic workload scheduling presented in [96] was based on a MT approach. One thread of processor 0 is dedicated to communication purposes, whereas the others are used only for computation. The communication thread keeps invoking a blocking MPI_Recv, in order to get messages from unspecified sources. As soon as a message arrives, the thread increments the *option_index* variable (which represents an option) by one and sends the old index to the origin process (which is waiting for a reply). The master sends only one option for each request, then each process uses several threads to parallelize the compute intensive part of the Monte Carlo simulation.

7.4.2.2 Dynamic Workload Scheduling based on CAF

As we already said, coarray Fortran allows one to access directly the memory exposed by other processes through get or put operations without explicitly involving the target process. This asymmetric paradigm can be used very effectively in those cases where processes cannot predict if and when a message will arrive. Coarray Fortran provides, in addition to usual synchronization mechanisms (full and partial barriers) and one-sided transfers subroutines, a set of atomic operations. In Technical Specification 18508, which will be included in the Fortran 2015 standard, there are new

intrinsics for atomic operations like *ATOMIC_FETCH_ADD*. This function allows one to perform an atomic fetch of the data from a remote memory segment and to update the remote value, by summing a new constant number. This intrinsic completely replaces the spinning communication thread adopted by the MPI-based version described in Section 7.4.2.1.

Unfortunately, the CAF implementation provided by the Intel compiler, which is required to run on Intel Xeon Phi, does not provide the atomic operations like *ATOMIC_FETCH_ADD*. In order to face this issue, we used the wrapper module provided with OpenCoarrays described in Sections 6.1 and 7.3.1.

7.4.2.3 MPI Passive One-sided Progress

As already stated in Section 5.3, the direct access to remote memory (RMA), implemented through the one-sided functions exposed by MPI-3.0, is supposed to provide better performance than the usual two-sided approach by overlapping communication and computation. Theoretically, the program running on the remote process does not need to call any routines to match the one-sided operations invoked by the source process. However, even though the network fabric is able to perform the data transfer without involving the host CPU, the MPI implementation requires to make library calls in order to make progress on outstanding communcation operations.

With the currently available high-performance networks, there are essentially three strategies for making progress: manual progress, thread-based progress, and communication offload.

In this work, we analyze performance for both approaches; as we show in Figure 7.13, the performance provided by the different progress strategies are closely related to the network fabric.

The overhead imposed by manual and thread-based progress turns into a performance penalty on the one-sided functions. Furthermore, the atomic operations have a variable cost, based on the contention on the variable to be updated. A direct translation of the MPI-based version (where the master sends only one option per request) into the CAF-based version, even when manual progress is implemented, does not provide good results because of the higher communication cost. The idea to overcome

this problem is to send more than one option per request to the workers. This simple solution can be very effective, but introduces several other problems that we discuss in the next section.

7.4.3 Experimental Platform

Each reported test has been run on Galileo, a Tier-1 system operated by CINECA, the Italian supercomputing consortium. Each compute node is equipped with two 8-core Intel Haswell processors at 2.40 GHz. About half of the available compute nodes also host dual Intel Xeon Phi 7120p. Each Xeon Phi has 61 cores at 1.1 GHz able to handle up to 4 threads and 8GB of RAM.

The application code for the Asian options pricing based on the Monte Carlo method has been compiled using the Intel Fortran Compiler 15.0.2 and IntelMPI-5.0.2 and linked with OpenCoarrays-1.0.0, compiled for IntelMIC and regular CPU, using a wrapper module for invoking the OpenCoarrays functions.

For the purposes of this work, we consider only one compute node and use the two CPUs and Xeon Phis together in *symmetric mode*.

7.4.4 Implementing CAF-based Dynamic Scheduling

Because the *ATOMIC_FETCH_ADD* intrinsic provided by Coarray Fortran allows to sum any constant number to the remote variable, a good idea for reducing the amount of transfers is to get more than one option per request. This idea carries with it a very simple but important question: what is the right number of options to use on each device? Fortunately, this is a well known issue addressed since the beginning of the past century, where scheduling problems were related to the manufacturing industry.

Before answering the question on the right number of options per device, we need to analyze the two possible solutions described in Section 7.4.1, that is, multiple options per process (MO) and multi-threading on single option (MT). In both cases, we want to finish as soon as possible the pricing for all the options (the number of total options is fixed). This also means that we want to maximize the throughput, expressed as the number of random simulations per second.

All the tests shown in the next subsections have been run on a single node of the platform described in Section 7.4.3.

7.4.5 Multiple Options per Process (MO)

In this case, T options are run in parallel, on each process, using OpenMP. Each thread will run P simulations (related to only one option) serially; both CPUs and Xeon Phis can be seen as parallel machines, parametrized by the number of cores.

In this scenario, the throughput of each device will increase as the number of assigned options increases until the device reaches its capacity. Figures 7.9 and 7.10 show the throughput while varying the number of options assigned to CPU and Intel Xeon Phi, respectively.

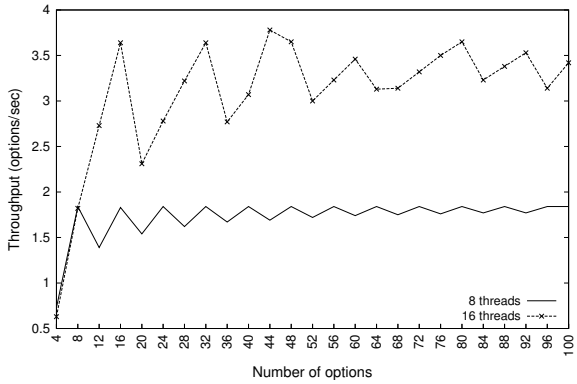


Figure 7.9: CPU throughput

Each curve in Figures 7.9 and 7.10 is labeled with the corresponding number of threads used in the computation. In particular, because each compute node has 2 CPUs with 8 cores each, in Fig. 7.9 we report the throughput using only one core (8 threads) and using both cores (16 threads).

It is clear that the maximum throughput is reached when the number of options is equal to the number of cores (threads) available on the device.

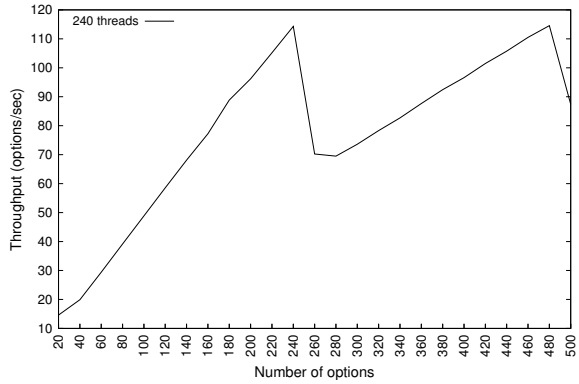


Figure 7.10: Intel Xeon Phi throughput

7.4.6 Multi-threading on Single Option (MT)

In this case, only one option is assigned to each process and P Monte Carlo simulations are run in parallel using OpenMP; within each simulation, vectorization is used as much as possible. From a scheduling point of view, CPUs and Xeon Phis can be seen as single machines with different service times. In other words, all the cores inside each device are already running at their maximum. Assigning more options to each device represents a queue of tasks that does not impact the throughput.

7.4.7 Analysis of the Two Approaches

Minimizing the time needed to compute for all the options, using a heterogeneous node equipped with four devices (2 multi-core CPUs and 2 many-core Intel Xeon Phis), can be seen as a *makespan minimization* problem without preemption.

The makespan represents the length of the schedule or, more precisely, the time when the last job leaves the system. Minimal makespan usually represents very good load balance.

From scheduling theory, finding a deterministic schedule that minimizes the makespan on 2 identical parallel machines, with jobs having different processing time, is NP-hard

problem in the ordinary sense. In our case, heterogeneity adds a further level of complexity and can be seen as a direct generalization of the homogeneous problem.

As a first step towards the creation of an effective heuristic, we notice that the heterogeneous problem can be easily transformed into a homogeneous problem. The idea is to find the ratio of work to submit to CPU and Xeon Phi such that both devices end the computation at the same time.

This rule transforms the parallel problem from heterogeneous to homogeneous but does not tell us anything about the exact quantity of work to give to each device.

At the beginning of this section, we mentioned that minimizing the makespan also means that the throughput has to be maximized. This means that the amount of work to be given to the heterogeneous devices, respecting the ratio, has to ensure maximum throughput on all devices.

Let us now consider the new homogeneous problem where all the devices work at maximum throughput and with the right ratio of options. This problem can now be addressed by applying the usual heuristics suitable for $Pm|Cmax$ problems (find a schedule such that the makespan is minimized while running jobs on *m* homogeneous parallel machines).

One of the most famous heuristics is the Longest Processing Time first (LPT) [114]. The idea is to place the shorter jobs towards the end of the schedule, where they can be used for balancing the load. In our case, this heuristic suggests to keep the amount of work on each device as small as possible, in order to keep all the devices as busy as possible.

As a last consideration, we need to reduce the communication costs as much as possible by sending more than one option at a time to each device.

Summarizing:

1. All the devices should take the same amount of time between two consecutive communications in order to simulate homogeneity.

2. All the devices should work as close as possible to their maximum throughput.

3. The amount of work to be given to each device should be as small as possible towards the end of the entire computation.

4. Communications with the master process should be reduced as much as possible.

For MO, respecting condition #2 means to send 240 options to each Xeon Phi and 8 options to each CPU, but this conflicts with rule #1. Furthermore, keeping the amount of data so high on each device also conflicts with rule #3; in fact, reducing the amount of work close to the end of computation means that each device does not work at maximum throughput anymore. Conversely, keeping the amount of computation at the maximum will likely leave some devices without enough work to do.

On the other hand, for the MT approach, even with a single option, condition #2 is always respected and condition #1 can be easily addressed. The only two conditions that interfere with each other are #3 and #4.

Reducing the number of communications means increasing the amount of work to give to each process; on the other hand, this increases the risk of having idle devices towards the end of the computation.

Figure 7.11 shows the maximum performance achievable for each single device, running the MT and MO implementations. Note how the latter (MO) evidences better performance than the former (MT).

7.4.8 Hybrid Approach

A good idea is to mix these two versions together and exploit as much as possible the characteristics of the available heterogeneous hardware. Because a CPU running the MO version provides higher performance than its MT counterpart, we decided to run the MO code, using eight options, only on one CPU (CPU1, the “farthest” from the boss) and run the MT version on the other devices, balancing the load accordingly.

7.4.9 Experimental Results

In this section, we analyze the performance of MT and MO, first on individual devices (without inter-process communication) using all the cores available on each device. Afterwards, we show the trade-off between amount of work and scheduling granularity for the MT version. Finally, in Sections 7.4.9.3 and 7.4.9.4, we present a performance

comparison between the MPI and CAF-based MT implementations with the hybrid implementation. We also show the behavior of manual and thread-based progress using different network fabric.

7.4.9.1 Performance on Single Device

Our first test examines the application performance using only a single device, without inter-process communication (i.e., without the boss-worker approach). In other words, only one process, running only on CPU or Xeon Phi, executes the whole amount of options. This provides an estimate of the maximum performance on each device, for a given implementation. Figure 7.11 shows the maximum performance achievable on CPU, Intel Xeon Phi, and the theoretical cumulative throughput running on a node with 2 CPUs and 2 Xeon Phi. Each CPU runs 8 threads, whereas the Xeon Phis run 240 threads.

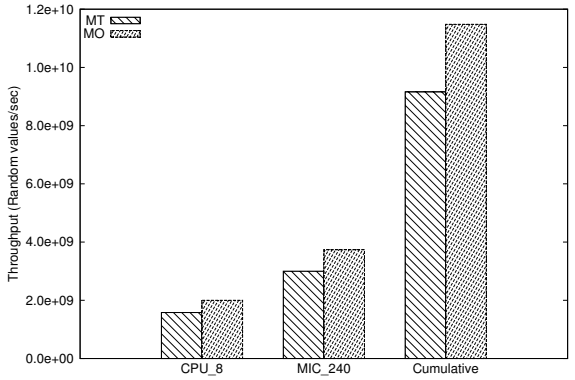


Figure 7.11: Homogeneous performance on CPU, Xeon Phi and theoretical heterogeneous throughput

The test shows that a single Intel MIC is about twice as fast as a CPU; therefore, to simulate homogeneity the options provided to the Intel MIC should be twice as many as the options provided to the CPU.

Furthermore, the MO implementation (when working with as many options as there

are the number of cores available on the device) gives higher performance than MT. On a single CPU, the MT version achieves 1.56 billions of random values per second, whereas MO achieves 2.0 billions of random values per second while working on eight options in parallel.

7.4.9.2 Communication/Task Size Trade-off

We now analyze how the number of options assigned to each device affects performance of the CAF-based version of MT. As explained in Section 7.4.4, when a dynamic load balancing approach is used, the application performance is influenced by two factors: 1) communication costs needed for transfering data; 2) idle time spent by devices without enough work to do. These two factors are inversely related and both directly influenced by the job size. In fact, increasing the number of options to send to a device reduces communication costs (a single big transfers costs less than several small transfers, in terms of latency and bandwidth). On the other hand, multiple options assigned in one shot to a single device impact scheduling granularity. In fact, close to the end of the execution, some devices will be unable to get enough options because they have been already taken by other devices. In Figure 7.12 we compare the idle time with the communication time (after normalizing both quantities in the range between 0 and 1). From the graph it is clear that we should assign no more than 3 options per CPU (and consequently no more than 6 options per MIC).

Since the costs in terms of idle time and communication is roughly the same for 2 and 3 options on the CPU, it is better to assign 2 options (4 to MICs), since a smaller granularity makes the application more flexible against possible performance changes on heterogeneous devices.

7.4.9.3 MT CAF-based Performance

In the MT CAF-based code, every process starts the computation by getting only one option from the boss process and saving the processing time in a coarray variable, accessible by all processes. By doing so, each device understands how much time is

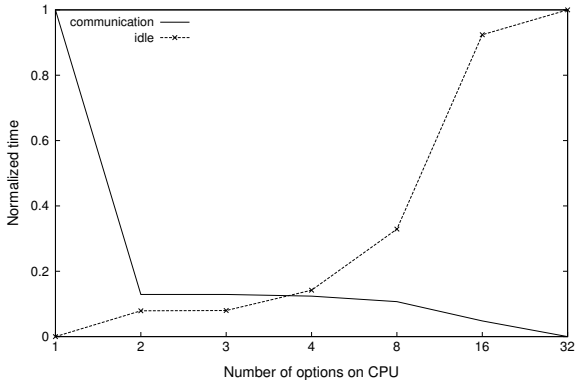


Figure 7.12: Trade-off between communication and idle time

needed for computing one single option. After a fixed number of computations, each accelerator checks the value of the processing time on the correspondent host device (e.g., MIC0 will check CPU0), and sets accordingly the number of options needed to simulate homogeneity (on Galileo, MICs are twice as fast as CPUs). Such phase is called the "learning phase" of the load balancing algorithm; in the current version this only happens once, but more complex and adaptive versions, repeating the sampling of the compute and/or remote communication time, can be implemented easily using the same strategy. However, it should be noted that the learning phase of the load balancing has a cost, so we should not search too many times.

In Figure 7.13, we compare the performance of the CAF-based versions with the MPI-based MT version, also taking into account different Intel MPI transport fabrics, specifically, the TMI (Tag Matching Interface) and the TCP fabrics. For instance, the label "shm:TCP", means that the fabric on the left hand side of the colon is used for intra-node communication (in this case, shared memory), and the fabric on the right hand side is used for inter-node communication. We observe how the network fabric has a huge impact on performance, in particular for the CAF-based version; in our case, since we are running on a single node, inter-node communication means communication between CPU and Xeon Phi using MPI.

Figure 7.13 also shows the effect of changing the message progress strategy. The

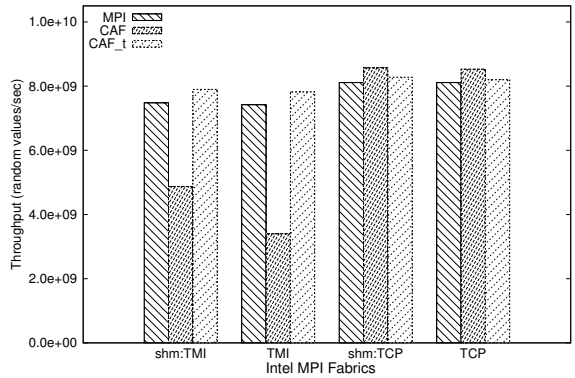


Figure 7.13: Comparison of MPI vs. CAF using different MPI fabrics

bars with a "_t" name suffix represent the performance using the thread-based progress strategy provided by Intel MPI. We explicitly note that performance of the CAF versions are better than MPI when the thread-based progress is used on the TMI fabric. Switching the fabric from TMI to TCP changes performance as well; in this case, the thread-based progress is worse than the manual progress. In both cases, using the TCP fabrics provides better performance than TMI. In Table 7.1, we report the time (in seconds) spent for communication with the boss process (on CPU0) for both MPI and CAF-based versions.

Table 7.1: Communication time (sec.)

Fabrics	MPI CPU1	MPI MIC1	CAF CPU1	CAF MIC1
shm:tmi	5.01×10^{-6}	2.00×10^{-5}	9.54×10^{-7}	3.08×10^{-2}
tmi	9.53×10^{-7}	2.91×10^{-5}	2.53×10^{-2}	6.42×10^{-2}
shm:tcp	9.54×10^{-7}	1.91×10^{-4}	9.54×10^{-7}	4.54×10^{-4}
tcp	5.96×10^{-6}	4.84×10^{-4}	6.10×10^{-5}	1.28×10^{-3}

7.4.9.4 Hybrid CAF-based Performance

As mentioned in the introduction, using different devices for different types of computation will become commonplace in the exascale era, where the compute nodes will be equipped with heterogeneous hardware.

In this last experiment, we run two different implementations (MO and MT) of the same application on different hardware, in order to exploit as much as possible the available heterogeneity.

As already mentioned, only CPU1 runs the MO version, taking eight options per communication. Each process, except the one running on CPU1, checks the compute time of CPU1 and adjusts the number of options to use accordingly (to achieve homogeneity). A typical run on Galileo has eight options (fixed) on CPU1, two on CPU0 and four on the two Intel MICs.

We have chosen to declare CPU1 as "special" because it suffers higher communication costs than CPU0 (the boss process runs always on CPU0). This fact is related with the costs induced by the NUMA architecture: the two Intel Haswell processors installed on a Galileo node are organized as two non-uniform memory access (NUMA) CPUs. Each portion of local memory on the CPU is called a memory domain. For a CPU, accessing the memory domain of the other CPU is possible but has a higher cost than accessing the local domain.

The boss process on CPU0, when the latter behaves as worker, can get the data from the same memory domain, which is very cheap; on the other hand, the process on CPU1 pays a higher cost than CPU0, because it has to get the data from a different memory domain. Having a bigger amount of data on CPU1 is beneficial to communication costs, but penalizes the scheduling granularity; on the other hand, because CPU0 has the lowest communication cost, it mitigates the bad effects of the scheduling granularity due to the eight options given to CPU1.

Figure 7.14 shows the remarkable results of this strategy (MTH) when the TCP fabric is used. Assuming the cumulative bar of MT in Fig. 7.11 as the maximum performance reachable with the hardware available, the MTH solution provides the

closest performance to the maximum.

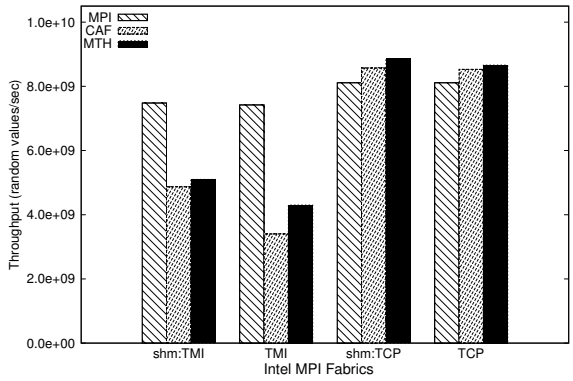


Figure 7.14: Hybrid CAF-based performance using different MPI fabrics

7.4.10 Conclusions

In this section, we analyzed the performance of dynamic load balancing algorithms implemented with MPI two-sided and Coarray Fortran. The one-sided semantic of coarrays allowed us to implement more advanced load balancing algorithms able to adapt to the heterogeneous hardware provided. Using the TCP fabric provided by Intel MPI, all coarray based versions show better results than the original MPI two-sided version. With the TMI fabric, choosing the right progression strategy is critical; in fact, using the thread-based progress provided by Intel MPI, leads to higher performance compared to the one shown by manual progress.

The CAF-based algorithm also allows us to manage highly heterogeneous situations, where two different versions of the same code run, at the same time, on the hardware more suitable for performance needs.

As future work, we plan to explore heterogeneous solutions based on dynamic load balancing strategies for different and more complex scientific problems. We also plan to enrich the analysis presented in this thesis introducing faults and performance variations.

8

Conclusions and Future Research

Contents

8.1 Summary

This thesis addressed some of the multiple aspects related with communication on exascale platforms. In particular, it showed how PGAS languages, mainly coarray Fortran, can be used effectively on exascale machines in order to reduce the impact of data movement. Current software is based on the idea that computing is the most expensive component but, in the exascale era, computing will be cheap and massively parallel, while data movement will dominate performance and energy consumption.

In this thesis, we focused on sparse and dense linear algebra kernels, as the most representative compute intensive kernels in scientific computing. We analyzed the difficulties of running such kernels on heterogeneous CPU+Accelerators platforms and how to improve performance using a different parallel programming model, more suitable for highly dynamic communication patterns.

Our main contribution was to provide an open source and standard implementation of coarray Fortran based on MPI-3.0, able to run on any platform equipped with a standard MPI implementation, including accelerators like Intel Xeon Phi. This contribution made also possible to analyze the potential of coarray Fortran on heterogeneous platforms like never before. The most remarkable example of this is the possibility to run coarray code on Intel Xeon Phis and CPUs at the same time, in symmetric mode, thanks to the MPI-based implementation of OpenCoarrays.

From a high-level perspective, in this thesis, we devised the following contributions:

- We proposed load balancing algorithms for hybrid CPUs+GPUs sparse matrix-vector computations.
- We designed, implemented and tested OpenCoarrays, a free and standard coarray transport layer, used by the GNU Fortran compiler.
- We proposed a new keyword for the Fortran language able to merge language features, like coarrays, with heterogeneous compute units, like accelerators. This keyword allows one to express data locality, potentially reducing data movement due to coherency protocols.
- We showed how to use effectively coarray Fortran on heterogeneous compute nodes, using Intel Xeon Phi, by implementing dynamic load balancing algorithms.

Working on a internationally standardized programming language like Fortran, required to interact with the international Fortran standards committee ISO/IEC/JTC1/SC22/WG5 (from now on WG5) and the US Fortran standards committee J3. OpenCoarrays was presented and well received at the J3/WG5 meeting 204 in Las Vegas

on June 2014. The idea behind the accelerated keyword was presented and discussed during the J3/WG5 meeting 207 in London on August 2015.

8.2 Future Research

As we mentioned in the introduction, exascale will introduce so much challenges that applications will need to be deeply reexamined; focusing on expressing parallelism as much as possible and considering data movement as the main bottleneck. Redesigning an application by focusing on data movement requires deep changes that involve several levels like: new algorithms, parallel programming models and dynamic load balancing strategies.

In the following sections, we propose several areas where the research presented in this thesis can be fruitfully applied and extented.

8.2.1 Asynchronous Algorithms

Algorithms currently considered computationally inefficient might be used effectively on the new architectures, where computation will not be the main issue anymore. As a concrete example, let us consider parallel asynchronous iterative solvers.

In [115–118], the authors investigate the pros and cons of asynchronous solvers, concluding that they can provide high performance benefits when communication has a substantial impact on overall performance (the papers consider asynchronous iterative solvers in a global computing context, where the machines are scattered all around the world). The problem with this class of algorithms is convergence detection and consequent halting procedure.

In order to express this concept more clearly, let us consider the iterative solvers used for solving linear systems. The classic approach performed by a synchronous iterative solver expects each process to run the same number of iterations until convergence and, at the end of each iteration, to synchronize with some “neighbours” for exchanging the data needed for the next iteration. It is clear that the implicit synchronization and the lack of overlapping, penalize this class of solvers on exascale platforms.

With an asynchronous iterative solver, each process does not wait for data and keeps on computing, trying to solve the given problem with whatever data happen to be available at that time. It is likely that each process will perform a number of iterations different from the others. Furthermore, because communication does not happen regularly, some processes might need more iterations in order to converge locally. This last concept represents what we mentioned at the beginning of this section: performing more computation (cheap) in order to avoid communication (expensive).

As reported in [119], asynchronous iterative solvers have been studied since 1969 but they have not been considered mainstream by researchers, mainly because of the difficulties related with convergence detection and halting condition. In general, for iterative solvers, global convergence is achieved when each node is in a stable state, that is, each node has locally converged.

Detecting convergence and stopping the execution can be implemented easily and efficiently with a PGAS language like coarray Fortran. At the same way, the one-sided communication provided by PGAS languages perfectly fits the asynchronous semantics exploited by this class of solvers.

8.2.2 Communication-Avoiding Algorithms, Heterogeneous Computing and Load Balancing

Parallel asynchronous algorithms are quite easy to implement using PGAS languages, but they do not consider data movement between memory levels. A much more effective, but sophisticated way to reduce communication is to transform the original algorithm in order to avoid/postpone communication as long as possible by performing redundant computation. Remarkable results has been pursued by James Demmel, Kathy Yelick and their collaborators at UC Berkeley on communication minimization in numerical linear algebra algorithms [120]. In [121, 122], they also focus their research on minimizing and avoiding communication on sparse matrix computation; the algorithms presented are able to minimize communication both within a local memory hierarchy and between processors.

Finding new communication-avoiding algorithms for scientific kernels other than

linear algebra is still an interesting and open problem. A good way to proceed is to consider the "13 Dwarfs" [123]: 13 typical scientific motifs that try to enclose all the most relevant scientific kernels. For the future, we plan to implement a test case for each motif, trying to exploit heterogeneity as much as possible, besides using avoiding- and overlapping- communication techniques based on coarray Fortran.

Using heterogeneous units for solving different tasks belonging to the same application, will require some sort of dynamic load balancing algorithms. In [124], Dinan et al. show how the work-stealing approach, implemented on top of PGAS languages, can work and scale well on thousands of cores. Finding new effective, dynamic, load balancing algorithms, suitable for the PGAS model can be another interesting research topic to pursue.

8.2.3 Optimal Strategies for MPI Progression

In Section 5.3, we described pros and cons of using MPI as transport layer for a PGAS language implementation like OpenCoarrays. Because one of the most important features of PGAS languages is the ability to perform asynchronous transfers, the underlying MPI implementation must guarantee asynchronous message progress. Basically, there are three strategies for making progress: manual progress, thread-based progress, and communication offload.

The manual progress approach is considered the easiest and most general to implement, because it gives complete control and responsibility to the programmer for the implementation of message progress.

To ensure asynchronous progress, traditional MPI implementations usually adopt a thread-based approach. Although this approach has been considered the "silver bullet" to address the MPI progress problem, it has several drawbacks. Polling the MPI progress engine with a communication thread, associated with each MPI process, can seriously impact performance because it halves the available hardware threads or cores. Relying on an interrupt-based approach, where the communication thread is woken up at the right time, can be effective when all the cores are busy doing computation, but it requires the OS involvement. This approach might not be the best possible on many-

core architectures, where the number of cores is very large.

In [89], the authors address this problem by dedicating a "ghost process" to communication purposes on each compute node. This ghost process is responsible for the communication of a set of processes allocated on the same compute node. When a remote process needs to communicate, it invokes an MPI one-sided function directed to the ghost process. The memory window on the ghost process is also a shared memory window accessible by any process on the same compute node. Message progression happens because of a blocking *MPI_Recv* called on the ghost process. In other words, message progression is managed only by a dedicated process in a manual fashion and the data delivery to the right process in performed through shared memory.

Even though the performance of an MPI implementation may not be as good as that of a communication library designed and implemented for a specific network fabric, we believe that using MPI as transport layer for PGAS languages, can make a difference in the widespread adoption of PGAS languages.

To the best of our knowledge, we see the MPI message progression as the most critical and difficult capability to ensure in order to provide a high quality PGAS transport layer. For this reason, we plan to investigate new ways, like the one proposed in [89], to implement efficiently asynchronous MPI communication.

8.2.4 Parallel Programming Models

Many research groups, like the DEGAS group[1], work at developing a new set of programming concepts based on a hierarchical model of parallelism and data locality, hierarchical fault containment/recovery for resilience and introspective dynamic resource management, using extensions to existing languages.

On the same wavelength, we plan to enrich OpenCoarrays with the missing features listed in TS-18508 and to adapt new features, suitable for exascale platforms, to the international standard committee. Furthermore, we are considering to propose the coarray semantic to modern programming languages other than Fortran. The most attractive option is to add coarray support to the Python language [95].

[1] http://crd.lbl.gov/departments/computer-science/CLaSS/research/DEGAS/

Bibliography

[1] J. A. Ang *et al.*, "Abstract machine models and proxy architectures for exascale computing," in *Proc. of 1st Int'l Workshop on Hardware-Software Co-Design for High Performance Computing*, ser. Co-HPC '14. IEEE, 2014, pp. 25–32. [Online]. Available: http://dx.doi.org/10.1109/Co-HPC.2014.4

[2] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *SIGARCH Comput. Archit. News*, vol. 23, no. 1, pp. 20–24, Mar. 1995. [Online]. Available: http://doi.acm.org/10.1145/216585.216588

[3] J. Shalf, S. Dosanjh, and J. Morrison, "Exascale computing technology challenges," in *Proceedings of the 9th International Conference on High Performance Computing for Computational Science*, ser. VECPAR'10. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 1–25. [Online]. Available: http://dl.acm.org/citation.cfm?id=1964238.1964240

[4] B. M. Rogers, A. Krishna, G. B. Bell, K. Vu, X. Jiang, and Y. Solihin, "Scaling the bandwidth wall: Challenges in and avenues for cmp scaling," *SIGARCH Comput. Archit. News*, vol. 37, no. 3, pp. 371–382, Jun. 2009. [Online]. Available: http://doi.acm.org/10.1145/1555815.1555801

[5] J. Jeddeloh and B. Keeth, "Hybrid memory cube new dram architecture increases density and performance," in *VLSI Technology (VLSIT), 2012 Symposium on*, June 2012, pp. 87–88.

[6] HMC Consortium, "Hybrid memory cube," 2015, http://www.hybridmemorycube.org/.

[7] D. Miller and H. Ozaktas, "Limit to the bit-rate capacity of electrical interconnects from the aspect ratio of the system architecture," *J. Parallel Distrib. Comput.*, vol. 41, no. 1, pp. 42–52, Feb. 1997. [Online]. Available: http://dx.doi.org/10.1006/jpdc.1996.1285

[8] D. Miller, "Rationale and challenges for optical interconnects to electronic chips," *Proceedings of the IEEE*, vol. 88, no. 6, pp. 728–749, June 2000.

[9] C. Li, Y. Yang, H. Dai, S. Yan, F. Mueller, and H. Zhou, "Understanding the tradeoffs between software-managed vs. hardware-managed caches in GPUs," in *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, March 2014, pp. 231–242.

[10] R. Dennard, V. Rideout, E. Bassous, and A. LeBlanc, "Design of ion-implanted MOSFET's with very small physical dimensions," *Solid-State Circuits, IEEE Journal of*, vol. 9, no. 5, pp. 256–268, Oct 1974.

[11] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA'11. New York, NY, USA: ACM, 2011, pp. 365–376. [Online]. Available: http://doi.acm.org/10.1145/2000064.2000108

[12] M. Taylor, "A landscape of the new dark silicon design regime," *Micro, IEEE*, vol. 33, no. 5, pp. 8–19, Sept 2013.

[13] M. Gordon, P. Goldhagen, K. Rodbell, T. Zabel, H. Tang, J. Clem, and P. Bailey, "Measurement of the flux and energy spectrum of cosmic-ray induced neutrons on the ground," *Nuclear Science, IEEE Transactions on*, vol. 51, no. 6, pp. 3427–3434, Dec 2004.

[14] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson, A. A. Chien, P. Coteus, N. A. Debardeleben, P. C. Diniz, C. Engelmann, M. Erez, S. Fazzari, A. Geist, R. Gupta, F. Johnson, S. Krishnamoorthy, S. Leyffer, D. Liberty, S. Mitra, T. Munson, R. Schreiber, J. Stearley, and E. V. Hensbergen, "Addressing failures in exascale computing," *Int. J. High Perform. Comput. Appl.*, vol. 28, no. 2, pp. 129–173, May 2014. [Online]. Available: http://dx.doi.org/10.1177/1094342014522573

[15] J. M. D. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. H. Bisseling, "Bsplib: The BSP programming library," 1998.

[16] D. B. Skillicorn, J. M. D. Hill, and W. F. Mccoll, "Questions and answers about BSP," 1996.

[17] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990. [Online]. Available: http://doi.acm.org/10.1145/79173.79181

[18] H. Kaul, M. Anders, S. Hsu, A. Agarwal, R. Krishnamurthy, and S. Borkar, "Near-threshold voltage (NTV) design; opportunities and challenges," in *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, June 2012, pp. 1149–1154.

[19] W. Gropp and M. Snir, "Programming for exascale computers," *Computing in Science Engineering*, vol. 15, no. 6, pp. 27–35, Nov 2013.

[20] F. Petrini, D. J. Kerbyson, and S. Pakin, "The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q," in *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, ser. SC '03. New York, NY, USA: ACM, 2003, pp. 55–. [Online]. Available: http://doi.acm.org/10.1145/1048935.1050204

[21] D. Tsafrir, Y. Etsion, D. G. Feitelson, and S. Kirkpatrick, "System noise, OS clock ticks, and fine-grained parallel applications," in *Proceedings of the 19th Annual International Conference on Supercomputing*, ser. ICS '05. New York, NY, USA: ACM, 2005, pp. 303–312. [Online]. Available: http://doi.acm.org/10.1145/1088149.1088190

[22] T. Hoefler, T. Schneider, and A. Lumsdaine, "Characterizing the influence of system noise on large-scale applications by simulation," in *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, Nov 2010, pp. 1–11.

[23] R. J. LeVeque, *Finite Difference Methods for Ordinary and Partial Differential Equations*. Philadelphia, PA: SIAM, 2007.

[24] A. Quarteroni and A. Valli, *Numerical Approximation of Partial Differential Equations*. Berlin: Springer-Verlag, 1994.

[25] S. V. Patankar, *Numerical Heat Transfer and Fluid Flow*, 1st ed., ser. Series in Computational Methods in Mechanics and Thermal Sciences. New York, NY, USA: Hemisphere Publishing Corp., 1980.

[26] T. Davis, "Wilkinson's sparse matrix definition," *NA Digest*, vol. 07, no. 12, pp. 379–401, Mar. 2007.

[27] D. Luebke, M. Harris, N. Govindaraju, A. Lefohn, M. Houston, J. Owens, M. Segal, M. Papakipos, and I. Buck, "GPGPU: general-purpose computation on graphics hardware," in *Proc. of 2006 ACM/IEEE Conf. on Supercomputing*, ser. SC '06, 2006.

[28] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *ACM Queue*, vol. 6, pp. 40–53, Mar. 2008.

[29] J. Sanders and E. Kandrot, *CUDA by example: An introduction to general-purpose GPU programming*, 1st ed. Boston, MA, USA: Addison-Wesley, 2010.

[30] N. Wilt, *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. Pearson Education, 2013. [Online]. Available: https://books.google.it/books?id=ynydqKP225EC

[31] R. Landaverde, Z. Tiansheng, A. Coskun, and M. Herbordt, "An investigation of unified memory access performance in cuda," in *Proc. of IEEE High Performance Extreme Computing Conf.*, ser. HPEC '14, Sep. 2014, pp. 1–6.

[32] V. Cardellini, A. Fanfarillo, and S. Filippone, "Hybrid coarrays: a PGAS feature for many-core architectures," in *Proceedings of International Conference on Parallel Computing (ParCo2015)*, Edinburgh, UK, Sep 2015, accepted for publication.

[33] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC'09. New York, NY, USA: ACM, 2009, pp. 18:1–18:11. [Online]. Available: http://doi.acm.org/10.1145/1654059.1654078

[34] J. W. Choi, A. Singh, and R. W. Vuduc, "Model-driven autotuning of sparse matrix-vector multiply on GPUs," *SIGPLAN Not.*, vol. 45, pp. 115–126, Jan. 2010.

[35] A. Monakov, A. Lokhmotov, and A. Avetisyan, "Automatically tuning sparse matrix-vector multiplication for GPU architectures," in *High Performance Embedded Architectures and Compilers*, ser. LNCS. Springer-Verlag, 2010, vol. 5952, pp. 111–125.

[36] H.-V. Dang and B. Schmidt, "CUDA-enabled sparse matrix-vector multiplication on GPUs using atomic operations," vol. 39, no. 11, pp. 737–750, 2013.

[37] J. C. Pichel, F. F. Rivera, M. Fernández, and A. Rodríguez, "Optimization of sparse matrix-vector multiplication using reordering techniques on GPUs," *Microprocess. Microsyst.*, vol. 36, no. 2, pp. 65–77, 2012.

[38] S. Mittal and J. S. Vetter, "A survey of CPU-GPU heterogeneous computing techniques," *ACM Comput. Surv.*, vol. 47, no. 4, pp. 69:1–69:35, Jul. 2015. [Online]. Available: http://doi.acm.org/10.1145/2788396

[39] J. Gummaraju, L. Morichetti, M. Houston, B. Sander, B. R. Gaster, and B. Zheng, "Twin peaks: A software platform for heterogeneous computing on general-purpose and graphics processors," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '10. New York, NY, USA: ACM, 2010, pp. 205–216. [Online]. Available: http://doi.acm.org/10.1145/1854273.1854302

[40] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal,

and P. Dubey, "Debunking the 100x GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU," *SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 451–460, Jun. 2010. [Online]. Available: http://doi.acm.org/10.1145/1816038.1816021

[41] V. Cardellini, A. Fanfarillo, and S. Filippone, "Heterogeneous sparse matrix computations on hybrid GPU/CPU platforms," in *Parallel Computing: Accelerating Computational Science and Engineering (CSE)*, ser. Advances in Parallel Computing. IOS Press, 2014, vol. 25, pp. 203–212.

[42] S. B. Indarapu, M. Maramreddy, and K. Kothapalli, "Architecture- and workload- aware heterogeneous algorithms for sparse matrix vector multiplication," in *Proc. of 19th IEEE Int'l Conf. on Parallel and Distributed Systems*, ser. ICPADS '13, Dec. 2013.

[43] W. Yang, K. Li, Z. Mo, and K. Li, "Performance optimization using partitioned SpMV on GPUs and multicore CPUs," p. To appear, 2014.

[44] K. Matam, S. Indarapu, and K. Kothapalli, "Sparse matrix-matrix multiplication on modern architectures," in *19th International Conference on High Performance Computing (HiPC), 2012*, Dec 2012, pp. 1–10.

[45] J. A. Stuart, P. Balaji, and J. D. Owens, "Extending MPI to accelerators," in *Proceedings of the 1st Workshop on Architectures and Systems for Big Data*, ser. ASBD '11. New York, NY, USA: ACM, 2011, pp. 19–23. [Online]. Available: http://doi.acm.org/10.1145/2377978.2377981

[46] D. B. Kirk and W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, 2nd ed. Morgan Kaufmann, 2012.

[47] S. Filippone and M. Colajanni, "PSBLAS: a library for parallel linear algebra computations on sparse matrices," *ACM Trans. on Math Software*, vol. 26, pp. 527–550, 2000.

[48] N. Whitehead and A. Fit-Florea, "Precision & performance: Floating point and IEEE 754 compliance for NVIDIA GPUs," NVIDIA Corporation, Tech. Rep., 2011.

[49] S. Filippone and A. Buttari, "Object-oriented techniques for sparse matrix computations in Fortran 2003," vol. 38, no. 4, pp. 23:1–23:20, 2012.

[50] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[51] V. Cardellini, S. Filippone, and D. Rouson, "Design patterns for sparse-matrix computations on hybrid CPU/GPU platforms," vol. 22, no. 1, pp. 1–19, 2014.

[52] Z. Zhong, V. Rychkov, and A. Lastovetsky, "Data partitioning on heterogeneous multicore and multi-GPU systems using functional performance models of data-parallel applications," in *Proc. of IEEE Cluster '12*, Sep. 2012, pp. 191–199.

[53] C.-K. Luk, S. Hong, and H. Kim, "Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *Proc. of 42nd IEEE/ACM Int'l Symp. on Microarchitecture*, 2009, pp. 45–55. [Online]. Available: http://doi.acm.org/10.1145/1669112.1669121

[54] A. Lastovetsky and R. Reddy, "Data partitioning with a functional performance model of heterogeneous processors," *Int'l J. of High Performance Computing Applications*, vol. 21, no. 1, pp. 76–90, 2007. [Online]. Available: http://hpc.sagepub.com/content/21/1/76.abstract

[55] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz, "Runtime measurements in the cloud: observing, analyzing, and reducing variance," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 460–471, Sep. 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1920841.1920902

[56] M. Bernaschi, M. Bisson, T. Endo, S. Matsuoka, M. Fatica, and S. Melchionna, "Petaflop biofluidics simulations on a two million-core system," in *Proc. of 2011 Int'l Conf. for High Performance Computing, Networking, Storage and Analysis*, 2011.

[57] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, A. Basermann, and A. R. Bishop, "Sparse matrix-vector multiplication on GPGPU clusters: A new storage format and a scalable implementation," in *Proc. of 26th IEEE Int'l Parallel and Distributed Processing Symposium Workshops & PhD Forum*, ser. IPDPSW '12, 2012, pp. 1696–1702.

[58] D. Jacobsen, J. Thibault, and I. Senocak, "An MPI-CUDA implementation for massively parallel incompressible flow computations on multi-GPU clusters," in *48th AIAA Aerospace Sciences Meeting*, 2010.

[59] D. R. Kincaid, T. C. Oppe, and D. M. Young, "ITPACKV 2D User's Guide," May 1989, http://rene.ma.utexas.edu/CNA/ITPACK/manuals/userv2d/.

[60] F. Vazquez, E. Garzon, and J. Fernandez, "The sparse matrix vector product on GPUs," 2009.

[61] T. Schroeder, "Peer-to-Peer & Unified Virtual Addressing," 2011, http://on-demand.gputechconf.com/gtc-express/2011/presentations/cuda_webinars_GPUDirect_uva.pdf.

[62] "Open MPI: Open Source High Performance Computing," http://www.open-mpi.org/.

[63] "What kind of CUDA support exists in Open MPI?" http://www.open-mpi.org/faq/?category=running#mpi-cuda-support.

[64] H. Wang, S. Potluri, D. Bureddy, C. Rosales, and D. Panda, "GPU-aware MPI on RDMA-enabled clusters: Design, implementation and evaluation," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 25, no. 10, pp. 2595–2605, Oct 2014.

[65] R. Preissl, N. Wichmann, B. Long, J. Shalf, S. Ethier, and A. Koniges, "Multithreaded global address space communication techniques for gyrokinetic fusion applications on ultra-scale platforms," in *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, Nov 2011, pp. 1–11.

[66] H. Shan, B. Austin, N. J. Wright, E. Strohmaier, J. Shalf, and K. Yelick, "Accelerating applications at scale using one-sided communication," in *Proceedings of the Conference on Partitioned Global Address Space Programming Models (PGAS12)*, 2012.

[67] H. Shan, A. Kamil, S. Williams, Y. Zheng, and K. Yelick, "Evaluation of PGAS communication paradigms with geometric multigrid," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. ACM, 2014, p. 8.

[68] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick, "Optimizing bandwidth limited problems using one-sided communication and overlap," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, April 2006, pp. 10 pp.–.

[69] R. W. Numrich and J. Reid, "Co-array Fortran for parallel programming," *SIGPLAN Fortran Forum*, vol. 17, no. 2, pp. 1–31, Aug. 1998. [Online]. Available: http://doi.acm.org/10.1145/289918.289920

[70] ——, "Co-arrays in the next Fortran standard," *SIGPLAN Fortran Forum*, vol. 24, no. 2, pp. 4–17, Aug. 2005. [Online]. Available: http://doi.acm.org/10.1145/1080399.1080400

[71] UPC Consortium, "UPC language specifications, v1.2," Lawrence Berkeley National Lab, Tech Report LBNL-59208, 2005. [Online]. Available: http://www.gwu.edu/~upc/publications/LBNL-59208.pdf

[72] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith, "Introducing openshmem: Shmem for the pgas community," in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, ser. PGAS '10. New York, NY, USA: ACM, 2010, pp. 2:1–2:3. [Online]. Available: http://doi.acm.org/10.1145/2020373.2020375

[73] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the chapel language," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 3,

pp. 291–312, Aug. 2007. [Online]. Available: http://dx.doi.org/10.1177/1094342007078442

[74] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele, and S. Tobin-Hochstadt, "The Fortress Language Specification," Sun Microsystems, Inc., Tech. Rep., March 2008, version 1.0.

[75] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An object-oriented approach to non-uniform cluster computing," *SIGPLAN Not.*, vol. 40, no. 10, pp. 519–538, Oct. 2005. [Online]. Available: http://doi.acm.org/10.1145/1103845.1094852

[76] J. Nieplocha, R. J. Harrison, and R. J. Littlefield, "Global arrays: A nonuniform memory access programming model for high-performance computers," *The Journal of Supercomputing*, vol. 10, no. 2, pp. 169–189, 1996.

[77] ISO/IEC/JTC1/SC22/WG5, "TS 18508 additional parallel features in Fortran," Aug. 2015, http://isotc.iso.org/livelink/livelink?func=ll&objId=17288706&objAction=Open.

[78] G. Jin, J. Mellor-Crummey, L. Adhianto, W. Scherer, and C. Yang, "Implementation and performance evaluation of the hpc challenge benchmarks in coarray fortran 2.0," in *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, May 2011, pp. 1089–1100.

[79] D. Eachempati, H. J. Jun, and B. Chapman, "An open-source compiler and runtime implementation for coarray Fortran," in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, ser. PGAS '10. New York, NY, USA: ACM, 2010, pp. 13:1–13:8. [Online]. Available: http://doi.acm.org/10.1145/2020373.2020386

[80] J. Daily, A. Vishnu, B. Palmer, and H. van Dam, "PGAS models using an MPI runtime: Design alternatives and performance evaluation," in *The International Conference for High Performance Computing, Network, Storage and Analysis. IEEE Computer Society*, 2013.

[81] J. Daily, A. Vishnu, B. Palmer, H. van Dam, and D. Kerbyson, "On the suitability of MPI as a PGAS runtime," in *21st International Conference on High Performance Computing (HiPC), 2014*, Dec 2014, pp. 1–10.

[82] A. Vishnu, J. Daily, and B. Palmer, "Designing scalable PGAS communication subsystems on cray gemini interconnect," in *19th International Conference on High Performance Computing (HiPC), 2012*, Dec 2012, pp. 1–10.

[83] A. Vishnu, D. Kerbyson, K. Barker, and H. van Dam, "Building scalable PGAS communication subsystem on BlueGene/Q," in *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, May 2013, pp. 825–833.

[84] A. Vishnu and M. Krishnan, "Efficient on-demand connection management mechanisms with PGAS models over InfiniBand," in *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, May 2010, pp. 175–184.

[85] R. Brightwell and K. D. Underwood, "An analysis of the impact of MPI overlap and independent progress," in *Proceedings of the 18th Annual International Conference on Supercomputing*, ser. ICS '04. New York, NY, USA: ACM, 2004, pp. 298–305. [Online]. Available: http://doi.acm.org/10.1145/1006209.1006251

[86] R. Brightwell, W. Lawry, A. Maccabe, and C. Wilson, "Improving processor availability in the MPI implementation for the ASCI/Red supercomputer," in *Local Computer Networks, 2002. Proceedings. LCN 2002. 27th Annual IEEE Conference on*, Nov 2002, pp. 639–647.

[87] T. Hoefler, G. Bronevetsky, B. Barrett, B. R. Supinski, and A. Lumsdaine, *Recent Advances in the Message Passing Interface: 17th European MPI Users' Group Meeting, EuroMPI 2010, Stuttgart, Germany, September 12-15, 2010. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, ch.

Efficient MPI Support for Advanced Hybrid Programming Models, pp. 50–61. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-15646-5_6

[88] T. Hoefler and A. Lumsdaine, "Message progression in parallel computing - to thread or not to thread?" in *Cluster Computing, 2008 IEEE International Conference on*, Sept 2008, pp. 213–222.

[89] M. Si, A. Pena, J. Hammond, P. Balaji, M. Takagi, and Y. Ishikawa, "Casper: An asynchronous progress model for MPI RMA on many-core architectures," in *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, May 2015, pp. 665–676.

[90] V. Cardellini, A. Fanfarillo, and S. Filippone, "Overlapping communication with computation in mpi applications," Tech. Rep. DICII RR-16.09, Università di Roma Tor Vergata, Tech. Rep., 2016.

[91] D. Bonachea, "GASNet Specification, v1.1." U.C. Berkeley, Technical Report UCB/CSD-02-1207, 2002.

[92] A. Fanfarillo, T. Burnus, V. Cardellini, S. Filippone, D. Nagle, and D. Rouson, "OpenCoarrays: Open-source transport layers supporting coarray Fortran compilers," in *Proc. of 8th Int'l Conf. on Partitioned Global Address Space Programming Models*, ser. PGAS '14. ACM, 2014, pp. 4:1–4:11. [Online]. Available: http://doi.acm.org/10.1145/2676870.2676876

[93] D. Henty, "A parallel benchmark suite for Fortran coarrays," in *Applications, Tools and Techniques on the Road to Exascale Computing*. IOS Press, 2012, pp. 281–288.

[94] D. Rouson, J. Xia, and X. Xu, *Scientific Software Design: The Object-Oriented Way*, 1st ed. New York, NY, USA: Cambridge University Press, 2011.

[95] C. E. Rasmussen, M. J. Sottile, J. Nieplocha, R. W. Numrich, E. Jones, and E. Inc, "Co-array Python: A parallel extension to the Python language."

[96] A. Vladimirov, R. Asai, and V. Karpusenko, *Parallel Programming and Optimization with Intel Xeon Phi Coprocessors, 2nd Edition*. Colfax International, 2015. [Online]. Available: http://www.colfax-intl.com/

[97] R. Rahman, *Intel Xeon Phi Coprocessor Architecture and Tools: The Guide for Application Developers*, 1st ed. Berkely, CA, USA: Apress, 2013.

[98] H. Wang, S. Potluri, D. Bureddy, C. Rosales, and D. Panda, "GPU-aware MPI on RDMA-enabled clusters: Design, implementation and evaluation," vol. 25, no. 10, 2014.

[99] A. Aji, J. Dinan, D. Buntinas, P. Balaji, W.-C. Feng, K. Bisset, and R. Thakur, "MPI-ACC: An integrated and extensible approach to data movement in accelerator-based systems," in *Proc. of IEEE 14th Int'l Conf. on High Performance Computing and Communication*, ser. HPCC '12, 2012.

[100] V. Cardellini, A. Fanfarillo, and S. Filippone, "Sparse matrix computations on clusters with GPGPUs," in *Proc. of 2014 Int'l Conf. on High Performance Computing Simulation*, ser. HPCS '14, 2014, pp. 23–30.

[101] M. Luo, M. Li, M. Venkatesh, X. Lu, and D. K. Panda, "UPC on MIC: Early experiences with native and symmetric modes," in *Proc. of Int'l Conf. on Partitioned Global Address Space Programming Models*, ser. PGAS '13, Oct. 2013.

[102] N. Namashivayam, S. Ghosh, D. Khaldi, D. Eachempati, and B. Chapman, "Native mode-based optimizations of remote memory accesses in OpenSHMEM for Intel Xeon Phi," in *Proc. of 8th Int'l Conf. on Partitioned Global Address Space Programming Models*, ser. PGAS '14. ACM, 2014, pp. 12:1–12:11. [Online]. Available: http://doi.acm.org/10.1145/2676870.2676881

[103] R. A. van de Geijn and J. Watts, "SUMMA: Scalable universal matrix multiplication algorithm," vol. 9, pp. 255–274, 1997.

[104] R. Landaverde, Z. Tiansheng, A. Coskun, and M. Herbordt, "An investigation of unified memory access performance in CUDA," in *Proc. of IEEE High Performance Extreme Computing Conf.*, ser. HPEC '14, Sep. 2014, pp. 1–6.

[105] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, no. 5, pp. 720–748, Sep. 1999. [Online]. Available: http://doi.acm.org/10.1145/324133.324234

[106] V. Cardellini, A. Fanfarillo, and S. Filippone, "Heterogeneous CAF-based load balancing on intel xeon phi," in *Proceedings of the 30th IEEE International Parallel and Distributed Processing Symposium (IPDPS) - AsHES16 Workshop*, Chicago, IL, May 2016, accepted for publication.

[107] P. Boyle and D. Emanuel, "Options on the general mean," University of British Columbia, Canada, Working paper, 1980.

[108] H. Geman and M. Yor, "Bessel processes, Asian options, and perpetuities," *Mathematical Finance*, vol. 3, pp. 349–375, Oct. 1993.

[109] J. Vecer, "A new PDE approach for pricing arithmetic average Asian options," *J. of Computational Finance*, vol. 4, no. 4, pp. 105–113, 2001.

[110] S. Turnbull and L. Wakeman, "A quick algorithm for pricing European average options," *J. of Financial and Quantitative Analysis*, vol. 26, no. 3, pp. 377–389, 1991.

[111] A. Kemna and A. Vorst, "A pricing method for options based on average values," *J. of Banking Finance*, vol. 14, pp. 113–129, 1990.

[112] P. Boyle, M. Broadie, and P. Glasserman, "Monte Carlo methods for security pricing," *J. of Economic Dynamics and Control*, vol. 21, pp. 1267–1321, 1997.

[113] R. Glenn Brook, A. Heinecke, A. Costa, P. Peltz, V. Betro, T. Baer, M. Bader, and P. Dubey, "Beacon: Exploring the deployment and application of Intel Xeon Phi coprocessors for scientific computing," *Computing in Science & Engineering*, vol. 17, no. 2, Mar. 2015.

[114] R. Graham, "Bounds on multiprocessing timing anomalies," *SIAM J. Appl. Math.*, vol. 17, pp. 416–429, 1969.

[115] J. M. Bahi, "Asynchronous iterative algorithms for nonexpansive linear system," *J. Parallel Distrib. Comput.*, vol. 60, no. 1, pp. 92–112, Jan. 2000. [Online]. Available: http://dx.doi.org/10.1006/jpdc.1999.1587

[116] J. M. Bahi, S. Contassot-Vivier, and R. Couturier, "Asynchronism for iterative algorithms in a global computing environment," *High Performance Computing Systems and Applications, Annual International Symposium on*, vol. 0, p. 90, 2002.

[117] J. Bahi, S. Contassot-Vivier, and R. Couturier, "Coupling dynamic load balancing with asynchronism in iterative algorithms on the computational grid," in *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, April 2003, pp. 9 pp.–.

[118] J. M. Bahi, S. Contassot-Vivier, and R. Couturier, "Evaluation of the asynchronous iterative algorithms in the context of distant heterogeneous clusters," *Parallel Comput.*, vol. 31, no. 5, pp. 439–461, May 2005. [Online]. Available: http://dx.doi.org/10.1016/j.parco.2005.02.009

[119] A. Frommer and D. B. Szyld, "On asynchronous iterations," *Journal of Computational and Applied Mathematics*, vol. 123, no. 12, pp. 201 – 216, 2000, numerical Analysis 2000. Vol. III: Linear Algebra. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S037704270000409X

[120] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz, "Minimizing communication in numerical linear algebra," *SIAM Journal on Matrix Analysis and Applications*, vol. 32, no. 3, pp. 866–901, 2011.

[121] J. Demmel, M. Hoemmen, M. Mohiyuddin, and K. Yelick, "Avoiding communication in sparse matrix computations," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, April 2008, pp. 1–12.

[122] M. Mohiyuddin, M. Hoemmen, J. Demmel, and K. Yelick, "Minimizing communication in sparse matrix solvers," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC

'09. New York, NY, USA: ACM, 2009, pp. 36:1–36:12. [Online]. Available: http://doi.acm.org/10.1145/1654059.1654096

[123] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from berkeley," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, Dec 2006. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html

[124] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha, "Scalable work stealing," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09. New York, NY, USA: ACM, 2009, pp. 53:1–53:11. [Online]. Available: http://doi.acm.org/10.1145/1654059.1654113