Coarray-based Load Balancing on Heterogeneous and Many-Core Architectures

Valeria Cardellini

University of Rome Tor Vergata, Italy

Alessandro Fanfarillo*

National Center for Atmospheric Research, USA

Salvatore Filippone

Cranfield University, UK

Abstract

In order to reach challenging performance goals, computer architecture is expected to change significantly in the near future. Heterogeneous chips, equipped with different types of cores and memory, will force application developers to deal with irregular communication patterns, high levels of parallelism, and unexpected behavior.

Load balancing among the heterogeneous compute units will be a critical task in order to achieve an effective usage of the computational power provided by such new architectures. In this highly dynamic scenario, Partitioned Global Address Space (PGAS) languages, like Coarray Fortran, appear a promising alternative to standard MPI programming that uses two-sided communications, in particular because of PGAS one-sided semantic and ease of programmability. In this paper, we show how Coarray Fortran can be used for implementing dynamic load balancing algorithms on an exascale compute node and how these algorithms can produce performance benefits for an Asian option pricing problem, running in symmetric mode on Intel Xeon Phi Knights Corner and Knights Landing architectures.

*Corresponding author

Email addresses: `cardellini@ing.uniroma2.it` (Valeria Cardellini), `elfanfa@ucar.edu` (Alessandro Fanfarillo), `Salvatore.Filippone@cranfield.ac.uk` (Salvatore Filippone)

Keywords: Partitioned Global Address Space, Coarray Fortran, Many-core

1. Introduction

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 1974 Dennard et al. [1] 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, creating a threat of thermal runaway and therefore further increasing 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. When many 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 [2], 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. It is therefore almost certain that in the near future energy constraints will lead to highly heterogeneous processors, equipped with several specialized circuits. Furthermore, energy will not only impact on computation but also (and particularly) on communication, both within and among nodes. Indeed, the energy required for off-chip communication will be much higher than that required for mere computation.

In this scenario, load balancing strategies, at different levels, will be critical to obtain an effective usage of the heterogeneous hardware and to reduce the impact of communication on energy and performance. Implementing efficient dynamic load balancing algorithms, capable of managing heterogeneous hardware, can be a challenging task, especially when a parallel programming model for distributed memory architecture, like *message passing*, is employed. The message passing programming model has been shown to be effective in several problems in High Performance Computing, in particular with homogeneous and regular applications, where the time required by communication and computation phases can be accurately estimated and perturbations are rare and with limited impact. Heterogeneity and task-based parallelism introduce a much more dynamic and unpredictable environment; this requires an alternative approach to the message passing model.

The ease of programming and asynchronous semantics provided by Partitioned Global Address Space (PGAS) languages, like Coarray Fortran (CAF) [3], UPC [4] and Chapel [5], can be used very effectively on heterogeneous hardware and/or when complex parallel algorithms must be implemented efficiently.

In this work we present a case study focusing on the load balancing of a Monte Carlo simulation for computing Asian option pricing. Thanks to its simple implementation and parallelization, this problem enables a clear demonstration of 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 and optimized for Intel Xeon and Xeon Phi architectures, have been taken from [6] with the kind permission of the authors.

Specifically, 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 and unexpected behaviors. In order to simulate a hypothetical exascale node equipped with heterogeneous components, we run our tests on a single compute node equipped with 2 CPUs and 2 Intel Xeon Phis Knights Corner (KNC) in *symmetric mode*; this mode of operation supported by the Intel Xeon Phi allows us to run the same parallel application on both components (i.e., CPUs and Intel Xeon Phis) as a regular MPI application.

Our results also show that by running different versions of the same application, at the same time, on the appropriate compute units we can achieve a significant performance improvement. Moreover, our experiments demonstrate how a CAF-based load balancing algorithm can also effectively deal with unexpected situations, where one or more computational units go slower than others because of frequency throttling. Finally, we provide a performance comparison on the new Knights Landing (KNL) architecture between a traditional two-sided approach based on MPI and an asynchronous one-sided approach based on CAF.

To the best of our knowledge, our work presents the first attempt at running a parallel application, based on Coarray Fortran, on CPUs and Intel Xeon Phis in symmetric mode using dynamic load balancing strategies.

In [7], Lua et al. discuss the intra-node memory access problems and host-to-XeonPhi connection issues for running UPC [4] applications on a Intel Xeon Phi system under native and symmetric programming modes. They find out several significant problems that affect UPC when running on many-core systems like Intel Xeon Phi, such as the communication bottleneck between Accelerator and host, unbalanced physical memory, and computation power issues. They conclude that adopting a load balancing strategy among Intel Xeon Phis and CPUs can be a significant optimization for applications running in symmetric mode. With respect to our own work in [8], we show that CAF-based load balancing can effectively deal with both unexpected behaviors and new architectures, such as the KNL one.

Although we focused on the Asian Options pricing problem, the strategy and algorithms presented in this work are totally general and can be applied to any embarrassingly parallel problem.

The rest of this paper is organized as following. In Section 2 we provide some background on PGAS and CAF. In Section 3 we describe the approaches for dynamic load balancing on heterogeneous nodes based on MPI and CAF. In Section 4 we present some considerations on CAF-based dynamic scheduling using Asian option pricing described in Section 4.1 as a reference problem, with two possible parallel implementations. In Section 5 we discuss the experimental results, using CPUs and Xeon Phis in different ways in order to exploit as much heterogeneity as possible. Finally, we draw our conclusions and outline future work in Section 6.

2. PGAS and Coarray Fortran

The PGAS 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 to get the best from) the Single Program Multiple Data (SPMD) approach, used in distributed memory systems, with the semantics of shared memory systems. In the PGAS model, every process has its own memory address space but mechanisms are available to share a portion of that address space with other processes.

The most common PGAS languages include Coarray Fortran (CAF) [3, 9], Unified Parallel C (UPC) [4] and Chapel [5]. PGAS languages rely on one-sided communication semantics: a process can get/put data from/to a memory segment exposed by another remote process, without explicitly involving the application on the remote node. Several modern networks allow the implementation of one-sided communications with Remote Direct Memory Access (RDMA), where the network interface directly takes care of the data transfer, without involving the remote CPU. There are several cases when a PGAS approach can easily solve difficult message passing situations because of the one-sided semantic; in general, whenever the communication is irregular and/or there is space for overlapping communication with computation, PGAS languages can show significant performance advantages. In this paper, we show how a PGAS language like Coarray Fortran can be effectively used for implementing dynamic load balancing algorithms which are suitable for heterogeneous platforms.

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 [3] and is now part of the Fortran 2008 standard (ISO/IEC 1539-1:2010) [9]. The main goal of coarrays is to allow Fortran users to create parallel programs without the burden of explicitly invoking communication functions or directives, as is the case with MPI and OpenMP.

A program that uses coarrays is treated as if it were replicated at the start of execution, and each replication is called an *image*. Each image owns its data, 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 number between one and the number of images (inclusive). In order to identify a specific image at run time or the total number of images, the `this_image()` and `num_images()` functions are provided.

A coarray is a data item that is visible to all participating images; it can be a scalar or array, static or dynamic, and of intrinsic or derived type. 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 the expression of more complex and useful mechanisms for synchronization, images organization and failure management. The Technical Specification 18508 (TS 18508) [10], which will be included in the Fortran 2015 standard, proposes the following extensions to the coarray facilities defined in Fortran 2008:

1. teams;
2. failed images;
3. events;
4. new intrinsic procedures (collectives and atomics).

Support for *teams* allows to group images into non-overlapping sets in order to execute different parts of the same application independently. *Failed images* provide a mechanism to identify what images have failed during the execution of a program. *Events* provide a fine grain ordering of execution segments based on a limited implementation of the well known semaphore primitives. Finally, *new collectives and atomic intrinsics* provide intrinsic procedures for commonly used collective and atomic memory operations (e.g. *ATOMIC_FETCH_ADD*). These procedures can be highly optimized for the target computational system, providing significantly improved program performance.

Since the inclusion of coarrays in the Fortran standard, the number of compilers implementing them has increased, and now Coarrays are supported by the Cray Fortran compiler, the Intel ifort, GNU Fortran, the Rice compiler, the OpenUH compiler, and the g95 compiler. OpenCoarrays [11] is an open-source transport layer supporting Coarray Fortran compilers. This library is currently the transport layer used by the GNU Fortran compiler; it provides several implementations based on different underlying communication layers, with the most complete and stable version being the one based on MPI-3.0. OpenCoarrays already supports several coarray features listed in TS 18508 [10], including Events and the new atomic intrinsics like *ATOMIC_FETCH_ADD*.

A very natural question arises when dealing with all coarray implementations based on MPI: what is the point when a user might call the MPI one-sided functions directly from the application? There are two main issues with this approach:

1. The syntax and semantics of the MPI one-sided functions are much more complex and error-prone than the syntax of Coarray Fortran, and thus require a significant programming effort;
2. Using MPI explicitly, the code is tied to that specific parallel programming system, whereas using coarrays, it is possible to keep the same syntax (and the same code) while changing the underlying communication layer.

As an example, it is possible to replace the MPI-based implementation of OpenCoarrays with the GASNet-based implementation without changing a line in the source code; a similar change for an MPI base program would require a significant recoding effort.

3. 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 [12].

The effect of the load balancing algorithm can be highlighted by applying it to an *embarrassingly parallel* problem, that is, a problem that can be split in multiple subproblems independent from each other. In this way, the nonlinear effects due to the interaction among subtasks are reduced to the barest minimum of contention for resources, and the speedup of the parallel application depends only on the efficiency and implementation of the load balancing algorithm. One such application will be described in more detail in Section 4.1.

Achieving an efficient static load balancing, conducive to very high performance, requires collection of performance data with benchmark runs; potentially many such runs may be needed to determine the correct ratio between CPU and Xeon Phi.

A dynamic load balancing approach relieves the users from performing preliminary tuning and allows them to manage unexpected performance perturbations in a transparent way. This flexibility comes at 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, after which we move onto the details of a CAF-based solution. The latter will be proven to be a valid alternative to MPI, thanks to its lightweight one-sided communication model and low overhead synchronization semantics.

3.1. Dynamic Workload Scheduling based on MPI

The most performing version of dynamic workload scheduling presented in [6] is based on a multi-threaded (MT) approach. One thread of processor 0 is dedicated to communication purposes, whereas the others are used for computation only. 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 by one the *workload_counter* variable, and sends the old index to the origin process (which is waiting for a reply). The master sends only one work unit for each request, then each process will use several threads to parallelize the compute intensive part of the Monte Carlo simulation.

3.2. Dynamic Workload Scheduling based on CAF

Coarray Fortran allows to directly access 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.

Besides usual synchronization mechanisms (full and partial barriers) and one-sided transfer subroutines, CAF provides a set of atomic operations. Among them, the *ATOMIC_FETCH_ADD* function allows 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 dynamic workload scheduling based on MPI that we have described in Section 3.1.

Unfortunately, the CAF implementation provided by the Intel compiler, which is required to run on Intel Xeon Phi (KNC), does not provide atomic operations like *ATOMIC_FETCH_ADD*. In order to face this issue, we implemented a wrapper module that makes the OpenCoarrays library [11] usable by any Fortran compiler. Since all communication functions of the OpenCoarrays library are based on MPI-3.0, we are able to compile it for both CPU and Intel Xeon Phi with the Intel Compiler and use it through the wrapper module.

Although MPI-3 offers great portability, several MPI implementations provide one-sided routines that do not take full advantage of the underlying network interconnect. For a PGAS communication library, the lack of fully asynchronous passive communication routines represent the most important limiting factor. In [13], we investigate the issue of asynchronous message progress in MPI applications, especially why it is hard to obtain even on networks equipped with adequate hardware. With currently available high-performance networks, there are essentially three strategies for making progress: manual progress, thread-based progress, and communication offload. Hoefler et al. [14] describe all three strategies and analyze the thread-based approach. They conclude that the thread-based progress, using polling (by-passing the operating system), 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).

4. Implementing CAF-based Dynamic Scheduling: Case Studies

Given that 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 work unit per request. This idea carries with it a very simple but important question: what is the right number of work units to assign to each device? Fortunately, this is a well known issue addressed since the beginning of the past century, where scheduling problems were related to manufacturing industry.

To study the effectiveness of the algorithms described in Section 3, we present now a use case from finance.

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 pre-specified period of time [15]. 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 Laplace transform of the Asian option price as in [16], numerical partial differential equation (PDE) techniques such as in [17], and various approximations such as those proposed by Turnbull et al. [18].

Using Monte Carlo simulation, multiple stochastic histories of the asset price are simulated based on the available information of the asset volatility [15, 19, 20]. 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 the usual master-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;
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-option) and to the latter as *MT* (multi-threaded). In Figures 1 and 2 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.

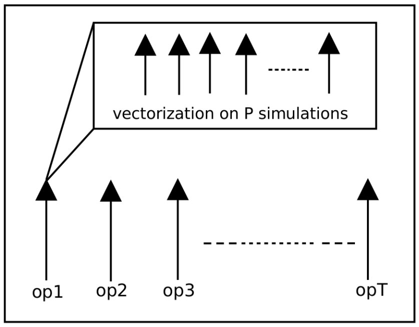


Figure 1: T options running on T OpenMP threads (MO approach)

We then need to consider what is the right number of options per device; to answer this question we need to analyze performance and consequences of the alternatives of multiple options per process (MO) and multi-threading on single option (MT). In both cases, we want to complete as soon as possible the pricing for all the options, given a fixed total number of options. This also means that we want to maximize the throughput expressed as the number of random simulations per second.

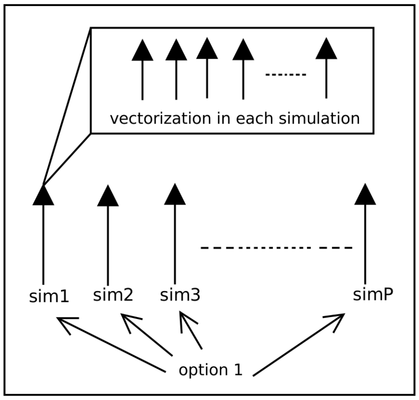


Figure 2: P simulations of a single option running on P OpenMP threads (MT approach)

We present here a few alternatives, with some preliminary performance data; a complete performance analysis will be presented in Section 5.

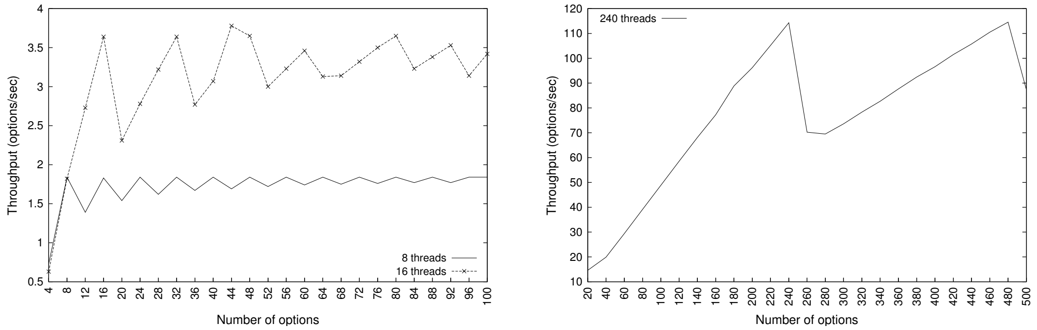
4.2. Multiple Options per Process (MO)

In the first case, T options are run in parallel, on each process, using OpenMP. Each thread will run P simulations, related to only one option, using as much as possible the AVX-2 and IMCI vector instructions, installed on Intel Haswell and Intel Xeon Phi *Knights Corner*, respectively. From a scheduling point of view, 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 3 (a) and (b) show the throughput while varying the number of options assigned to CPU and Intel Xeon Phi, respectively.

Each curve in Figures 3 (a) and (b) is labeled with the corresponding number of threads used in the computation. In particular, since each compute node has 2 CPUs with 8 cores each, we report the throughput using only one CPU (8 threads), as well as using both CPUs (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, or a multiple thereof.



(a) CPU

(b) Intel Xeon Phi

Figure 3: Performance comparison: throughput.

4.3. 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 on the throughput.

4.4. 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 an NP-hard problem. In our case, heterogeneity adds a further level of complexity and can be seen as a direct generalization of the homogenous 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 in the amount of work to be submitted 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 homogenous but does not tell us the exact amount of work to assign to each device.

We mentioned previously that minimizing the makespan also means maximizing the throughput; this means that the amount of work to be given to the heterogeneous devices, while still respecting the optimal 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 correct ratio of options. This problem can be now addressed by applying the usual heuristics suitable for $Pm|Cmax$ problems, i.e. find a schedule such that the makespan is minimized while running jobs on m homogeneous parallel machines.

One of the most popular heuristics is the Longest Processing Time first (LPT) [21]. 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 limit the idle time on all devices, as far as possible. As a last consideration, we need to reduce the communication costs as much as possible by sending more than one option at time to each device.

Summarizing:

1. All devices should take the same amount of time between two consecutive communications in order to emulate homogeneity;
2. All devices should work as close as possible to their maximum throughput;
3. The amount of work to be given to each device should to 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, such a large amount of data 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 any longer. Viceversa, 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 be given to each process; on the other hand, this increases the risk of having idle devices towards the end of the scheduling.

As already noted, for MO we cannot give to the devices less options than the number ensuring maximum throughput; in our case, 240 for the Xeon Phis and 8 for the CPUs. Such restriction will most likely produce a situation where one or more devices will not be able to get enough options and they will spend all the remaining time in the idle state.

With the MT approach, we are guaranteed to get the maximum throughput even with only one option per device. This allows us to simulate homogeneity by adjusting the number of options to assign to each device.

Figure 5 shows the maximum performance achievable for each single device, running the MT and MO implementations. Note how the latter (MO in the chart) has better performance than the former (MT).

4.5. Hybrid Approach

An appealing idea would be to merge the 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 (i.e., CPU1 which is the "furthest" from the master) and run the MT version on the other devices, balancing the load accordingly. An instance of scheduling related to this configuration is depicted in Figure 4. From now on, we will refer to this hybrid version as MTH.

4.6. Unexpected Slowdown

One of the most important advantage of a dynamic load balancing approach is the ability to manage unexpected slowdown events on one or more compute units. As noted in Section 1, unexpected behaviors due to frequency throttling are in general unavoidable, and therefore we have to cope with their consequences. In this scenario, taking one option at time represents a good solution in terms of flexibility; however, choosing more than one option at a time may still be a better performing solution if we can have a fully collaborative approach.

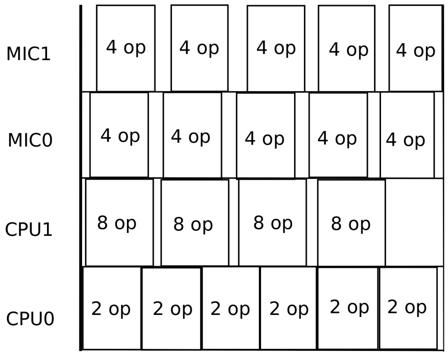


Figure 4: Instance of scheduling for MTH

The idea is then to provide information about the class of devices installed on the heterogeneous node (in our case, there are only two classes: CPUs and Xeon Phis) and perform an intra-group check in order to understand whether a device is experiencing a slowdown. In such a case, the remaining devices increase the amount of work units in order to compensate the slowdown.

5. Experimental Results

In this section we present a set of experiments on multiple platforms. First of all, we analyze the performance of MT and MO on individual devices (without inter-process communication) using all the cores available on each device. Then, we focus on the trade-off between the amount of work and the scheduling granularity for the MT version. Afterwards, in Sections 5.4 and 5.5, we discuss the results of 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 fabrics. In Section 5.6 we then describe a similar comparison in the presence of unexpected delay on a compute unit. Finally, in Section 5.7 we present a performance comparison between MPI and CAF-based implementations on the new architecture Intel Xeon Phi *Knights Landing*.

5.1. Experimental Platforms

A first set of tests 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 E5-2630 v3 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. For the purposes of this work, we consider only one compute node and use the two CPUs and Xeon Phis together in *symmetric mode.*

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 Intel Xeon Phi and regular CPU, using a wrapper module for invoking the OpenCoarrays functions.

In order to show the advantages of a CAF-based solution for the Asian options pricing problem on the new Intel Xeon Phi Knights Landing (KNL) processor, a second set of tests have been run on Stampede, a 10 PFLOPS Dell Linux cluster based on 6400+ Dell PowerEdge operated by the TACC (Texas Advanced Computing Center). The KNL was configured in Quadrant cluster mode and the high-bandwidth memory (MCDRAM) was configured in Cache mode.

5.2. Performance on Single Device

Our first test examines the application performance using only a single device, without inter-process communication (i.e., without the master-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 5 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 each Xeon Phi runs 240 threads.

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

Furthermore, the MO implementation (when working with as many options as cores available on the device) gives higher performance than MT. On a single CPU, the MT version achieves 1.56 billions random values per second whereas the MO achieves 2.0 billions random values per second while working on eight options in parallel.

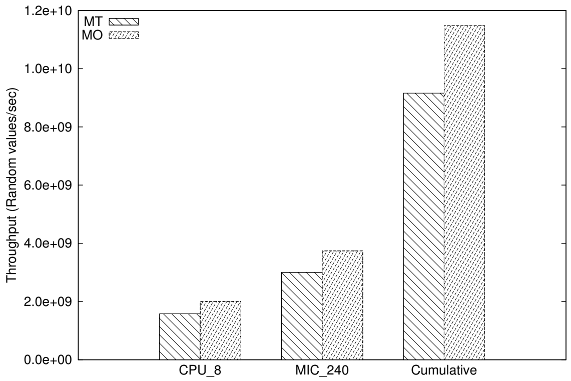


Figure 5: Homogenous performance on CPU, Xeon Phi and theoretical heterogeneous throughput

5.3. Communication/Job Size Trade-off

We now analyze how the number of options assigned to each device affects the performance of the CAF-based version of MT. As explained in Section 4.4, when a dynamic load balancing approach is used, the application performance is influenced by two factors: 1) communication costs needed for transferring data; 2) idle time spent by devices without enough work to do. These two factors are related in inverse proportion, and both are directly influenced by the job size. Indeed, if we increase the number of options sent to a device, the communication costs will be lower, because a single large transfer costs less than several small transfers, which have less opportunity to achieve maximal bandwidth and pay multiple times the latency cost. On the other hand, assigning multiple options in one shot to a single device (job) negatively influences the scheduling granularity; towards the end of the execution, some devices will be unable to get enough options because they have been already taken in a previous job by other devices.

In Figure 6 we compare the effect of idle time with the communication time (after normalizing both quantities in the range between 0 and 1). From the graph we can deduce that we should assign no more than 3 options per CPU (and consequently no more than 6 options per Xeon Phi).

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

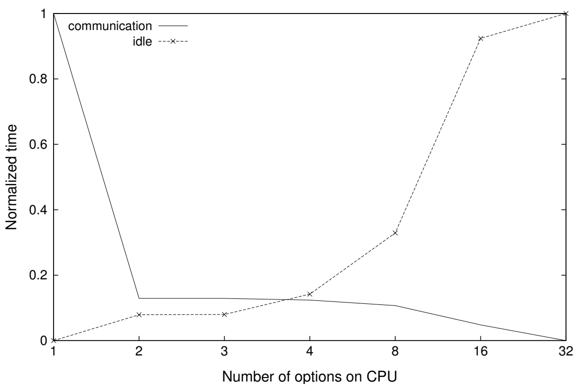


Figure 6: Trade-off between communication and idle time

5.4. MT CAF-based Performance

In the MT CAF-based code, every process starts the computation by getting just one option from the master process and saving the processing time in a coarray variable, accessibile by any process. By doing so, each device can deduce how much time is needed for computing a single option; after a fixed number of computations, each accelerator checks the value of the processing time on the correspondent host device (e.g., XeonPhi0 will check CPU0), and sets accordingly the number of options needed to simulate homogeneity (on Galileo, Xeon Phis are twice as fast as CPUs). This is called the “learning phase” of the load balancing algorithm; in the current version it only happens once, but more complex versions might repeat this phase, sampling the compute and/or remote communication time, several times using the same strategy. However, the learning phase of the load balancing has a cost, so we should not execute it too often otherwise its benefits will be outweighed.

In Figure 7 we compare the performance of the CAF-based versions against the MT MPI-based version; here we also take into account different Intel MPI transport fabrics, and specifically the TMI (Tag Matching Interface) and the TCP fabrics. TMI is an API used by Intel MPI to get performance benefits from transport layers that provide their own message matching logic.

Each label such as “shm:TCP” indicates that the fabric on the left of the colon is used for intra-node communication (in this case, shared memory), while the fabric on the right of the colon is used for inter-node communication

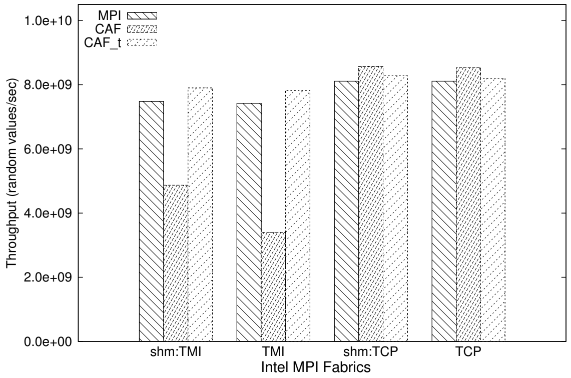


Figure 7: Comparison of MPI vs. CAF using different MPI fabrics

(in this case, TCP). 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 also shows the effect of changing the message progress strategy. The bars with a "_t" name suffix represent the performance using the thread-based progress strategy provided by Intel MPI. We explicitly note that the throughput of the CAF versions is better than that of the MPI version 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 throughput is lower than with manual progress. In both cases, using the TCP fabrics provides better performance than TMI.

5.5. Hybrid CAF-based Performance

According to foreseeable trends, 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 the next 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.

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 Xeon Phis.

We have chosen to declare CPU1 as "special" because it suffers from higher communication costs than CPU0 (the master process always runs on CPU0). This is related to the costs introduced by the NUMA architecture: the two Intel Haswell processors installed on a Galileo's node are organized as two non-uniform memory access (NUMA) CPUs. Each portion of local memory on the CPU is called a memory domain; one CPU can access the memory domain of the other CPU but at a higher cost than accessing the local domain.

The master 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 benefits the 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.

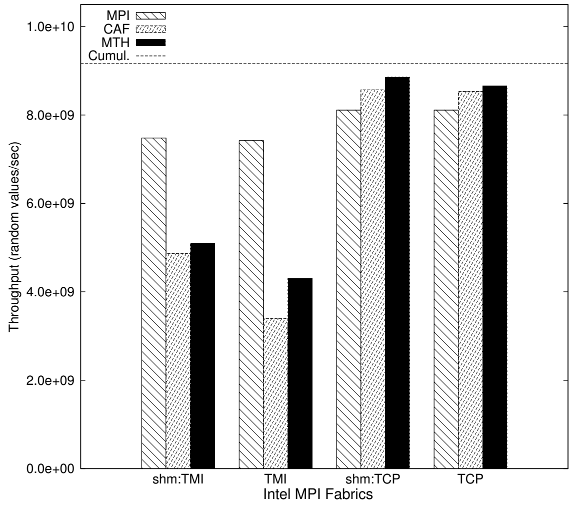


Figure 8: Hybrid CAF-based performance using different MPI fabrics

Figure 8 shows the remarkable results obtained by the MTH strategy when the TCP fabric is used. In fact, assuming the cumulative bar of MT in Figure 5 as the maximum performance reachable with the available hardware (represented by the dashed line), we can see that the MTH solution provides performance closest to the maximum possible.

5.6. Unexpected Slowdown

A dynamic load balancing algorithm allows the users to automatically manage unexpected performance perturbations and performance differences.

As explained in Section 5.3, there is a clear trade-off between communication cost and scheduling cost. A dynamic load balancing algorithm becomes more "flexible" to changes when the amount of work taken from the master is small (high communication cost).

When a compute unit is affected by an unexpected slowdown, the approaches described in Sections 3.1 and 3.2 already constitute viable choices. However, the one-sided semantics provided by CAF allows to design smarter and more complex algorithms, capable of reducing the communication cost and improving the performance.

The algorithm we propose in this section (from now on called UNX), derives from the dynamic load balancing approach described in Section 5.4. Every device is supposed to know a-priori how many classes of heterogeneous devices compose the node and the class to which it belongs to (in our particular case, there are two classes of devices: CPUs and Xeon Phis). In order to simulate a delay on a Xeon Phi, we introduce a call to the `usleep` function in the portion of code executed during the evaluation of each option. After completing the "learning phase" described in Section 5.4, each device reads (using a CAF get operation) the processing time on the other device belonging to the same class. If the different between the processing time taken from the remote device and the local processing time is greater than a predefined threshold (set to 0.01 seconds), the fastest device increases the amount of options taken each time. The increment Δ depends on the processing time difference between the two devices, and is calculated as:

$$\Delta = max(\lceil T_r/T_l \rceil, 1) \tag{1}$$

where T_r and T_l represent the remote and local processing time, respectively.

Despite its simplicity, this formula produces a noticeable performance improvement compared to the original algorithm described in Section 3.1, as shown in Figure 9.

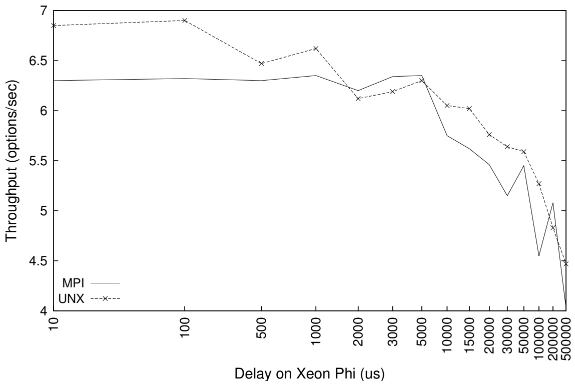


Figure 9: Performance comparison in case of unexpected delay

During the peer checking, the slowest device becomes aware of being slower than the other device and it will perform again the “learning phase” explained in Section 5.4, resulting in a reduction of the amount of options taken from the master image.

Because the number of options on the faster node is represented by an integer, a change in the number of options may represent a performance improvement or penalty. In Figure 9, the fastest Xeon Phi does not increment the number of options until the delay reaches 10000 us, whereas the slowest Xeon Phi reduces the number of options (from 4 to 2) since the very beginning. This difference in terms of “sensitivity” between the two heuristics leads to the performance improvement with respect to the original algorithm.

5.7. Early Experience on Knights Landing

The new Xeon Phi architecture, known as Knights Landing (KNL), brings several substantial changes compared to the old Knights Corner (KNC). The most relevant change consists in the fact that KNL is no longer a coprocessor and thus it does not require a host CPU. Since the presence of the PCIe bus between host and accelerator has always been one of the most restrictive performance constraints, eliminating the host processor has potentially huge advantages.

Furthermore, KNL adopts a 2D network topology in order to connect all the cores (actually the tiles, each of which contains two cores), instead of the bidirectional high-bandwidth ring used on KNC. This new configuration improves latency and bandwidth when multiple MPI processes are executed on the device.

As discussed previously in section 2, the use of one-sided communication semantics potentially increases performance, and PGAS languages such as CAF are a convenient way to embed this semantics into the application code.

The fact that every MPI process is allocated on a shared memory device and that communication is high performing, should make PGAS languages very convenient for the exploitation of the KNL architecture.

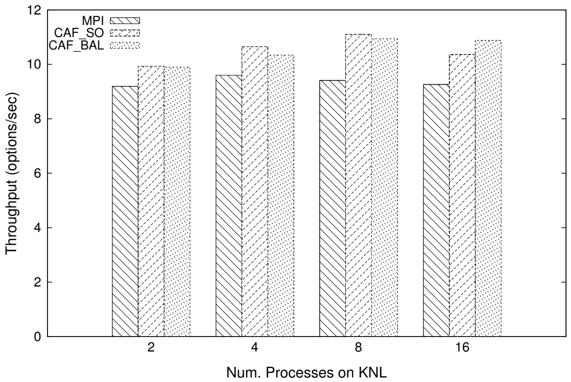


Figure 10: Performance comparison between MPI and CAF on KNL

In order to evaluate the above statement, we compare the original algorithm, based on MPI two-sided semantics and described in Section 3.1, with the equivalent CAF-based version with only a single option at time (represented by CAF_SO); the results are presented in Figure 10 for a varying number of processes on KNL. We also show the performance achieved by the CAF-based load balancing algorithm described in Section 5.4 (represented by CAF_BAL). Observing the results, we can confirm that the MPI two-sided approach is the slowest one, while the one-sided semantics of CAF provides a performance gain. Furthermore, load balancing achieves the best performance when the number of processes on the device is equal to 16, because, with a growing number of processes, the effect of contention on the 2D mesh becomes higher, and therefore the benefit of reducing the communication cost becomes more noticeable.

6. Conclusions

In this paper we presented dynamic load balancing algorithms based on Coarray Fortran and evaluated their performance with respect to an MPI two-sided approach. The one-sided semantic of coarrays allowed us to design more advanced load balancing algorithms able to adapt to the heterogeneous hardware. 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 than the 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 the performance needs. Even though the CAF implementation used in the tests is based on MPI-3.0, it provides better performance than the explicit MPI two-sided implementation, because the communication pattern required by the application is more suitable for the one-sided semantic. The pure MPI two-sided implementation works well when a single thread on the master process is used as communication thread, dispatching only one option for each request. A direct translation of this algorithm from MPI two-sided to CAF leads to poor performance, mainly because of the poor one-sided implementation provided by the MPI layer. On the other hand, a more complex algorithm which sends more than one option at time, is more suitable for a one-sided semantic than a two-sided one and allows to implement the hybrid solution proposed in Section 5.5, which leads to the best performance.

In a scenario where unexpected performance slowdown are possible, the one-sided semantic of CAF allows us to implement a collaborative approach which performs better than the algorithm based on the MPI two-sided approach.

Furthermore, on the new Intel Xeon Phi *Knights Landing* architecture, because of technological reasons the one-sided communication operations perform better that the usual MPI two-sided. Nonetheless, dynamic load balancing algorithms based on PGAS languages can still be effective on many-core devices when the number of processes allocated, and the relative traffic generated, is high.

Finally, although efficient algorithms using explicitly MPI one-sided routines can be realized, we observe that CAF provides a cleaner and more understandable syntax, allowing to easily describe complex parallel algorithms.

As future work, we plan to explore heterogeneous solutions based on dynamic load balancing strategies for different and more complex scientific problems.

Acknowledgments

We gratefully acknowledge the support received from CINECA for the OpenCoarrays2.0 project under the ISCRA 2016 grant program. A. Fanfarillo also acknowledges the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1053575.

References

[1] R. Dennard, V. Rideout, E. Bassous, A. LeBlanc, Design of ion-implanted MOSFET's with very small physical dimensions, IEEE J. of Solid-State Circuits 9 (5) (1974) 256–268.

[2] M. Taylor, A landscape of the new dark silicon design regime, IEEE Micro 33 (5) (2013) 8–19.

[3] R. Numrich, J. Reid, Co-array Fortran for parallel programming, SIGPLAN Fortran Forum 17 (2) (1998) 1–31.

[4] UPC Consortium, UPC Language Specifications, v1.2, Tech Report LBNL-59208, Lawrence Berkeley National Lab (2005).

[5] Chamberlain, B.L. and Cray Inc., Chapel, `http://chapel.cray.com` (2013).

[6] A. Vladimirov, R. Asai, V. Karpusenko, Parallel Programming and Optimization with Intel Xeon Phi Coprocessors, 2nd Ed., Colfax Int'l, 2015.

[7] M. Luo, M. Li, M. Venkatesh, X. Lu, 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, PGAS '13, 2013, pp. 198–210.

[8] V. Cardellini, A. Fanfarillo, S. Filippone, Heterogeneous CAF-based load balancing on Intel Xeon Phi, in: Proc of 30th IEEE Int'l Parallel and Distributed Processing Symp. Workshops, IPDPSW '16, 2016, pp. 702–711.

[9] R. W. Numrich, J. Reid, Co-arrays in the next Fortran standard, SIGPLAN Fortran Forum 24 (2) (2005) 4–17.

[10] ISO/IEC/JTC1/SC22/WG5, TS 18508 additional parallel features in Fortran (Aug. 2015).

[11] A. Fanfarillo, T. Burnus, V. Cardellini, S. Filippone, D. Nagle, 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, PGAS '14, ACM, 2014.

[12] R. Glenn Brook, A. Heinecke, A. Costa, P. Peltz, V. Betro, T. Baer, M. Bader, P. Dubey, Beacon: Exploring the deployment and application of Intel Xeon Phi coprocessors for scientific computing, Computing in Science & Engineering 17 (2).

[13] V. Cardellini, A. Fanfarillo, S. Filippone, Overlapping communication with computation in MPI applications, Tech. Rep. DICII RR-16.09, Univ. Rome Tor Vergata, http://hdl.handle.net/2108/140530 (2016).

[14] T. Hoefler, A. Lumsdaine, Message progression in parallel computing - to thread or not to thread?, in: Proc. of 2008 IEEE Int'l Conf. on Cluster Computing, 2008, pp. 213–222.

[15] P. Boyle, D. Emanuel, Options on the general mean, Working paper, University of British Columbia, Canada (1980).

[16] H. Geman, M. Yor, Bessel processes, Asian options, and perpetuities, Mathematical Finance 3 (1993) 349–375.

[17] J. Vecer, A new PDE approach for pricing arithmetic average Asian options, J. of Computational Finance 4 (4) (2001) 105–113.

[18] S. Turnbull, L. Wakeman, A quick algorithm for pricing European average options, J. of Financial and Quantitative Analysis 26 (3) (1991) 377–389.

[19] A. Kemna, A. Vorst, A pricing method for options based on average values, J. of Banking Finance 14 (1990) 113–129.

[20] P. Boyle, M. Broadie, P. Glasserman, Monte Carlo methods for security pricing, J. of Economic Dynamics and Control 21 (1997) 1267–1321.

[21] R. Graham, Bounds on multiprocessing timing anomalies, SIAM J. Appl. Math. 17 (1969) 416–429.