Heterogeneous CAF-based Load Balancing on Intel Xeon Phi

Valeria Cardellini
University of Rome Tor Vergata
Rome, Italy
cardellini@ing.uniroma2.it

Alessandro Fanfarillo
University of Rome Tor Vergata
Rome, Italy
fanfarillo@ing.uniroma2.it

Salvatore Filippone
Cranfield University
Cranfield, UK
Salvatore.Filippone@cranfield.ac.uk

***Abstract*—In order to reach challenging performance goals, computer architectures will change significantly in the next future. Heterogeneous chips, equipped with different types of cores and memory will compel application developers to deal with irregular communication patterns, high parallelism, and unexpected behaviors. Load balancing among the heterogeneous compute units will be a critical task in order to exploit all the computational power provided by such new architectures. In this highly dynamic scenario, Partitioned Global Address Space (PGAS) languages, like Coarray Fortran (CAF), appear to be a promising alternative to standard MPI programming using two-sided communications, in particular because of their one-sided semantic. In this work, 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 (KNC).**

I. 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, 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. 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 [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. In conclusion, in the next 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, able to manage heterogeneous hardware, can be a challenging task, especially when a parallel programming model for distributed memory architecture, like *message passing*, is required. 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 unlikely and with minimal impact. Heterogeneity and task-based parallelism introduce a much more dynamic and unpredictable environment, which requires an alternative approach to the common and widely adopted message passing model.

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

In this work, we focus on load balancing a simple Monte Carlo simulation for computing Asian option price. Thanks to its simple 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 and optimized for Intel Xeon and Xeon Phi architectures, have been taken from [6] with the kind permission from 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. 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 supported by 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.

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 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-MIC connection issues for running UPC [4] applications on a MIC system under native and symmetric programming modes. They find out several significant problems that affect UPC when running on many-core systems like MIC, such as the communication bottleneck between MIC and host, unbalanced physical memory, and computation power issues. They conclude that adopting a load balancing strategy among MICs and CPUs can be a relevant optimization for applications running in symmetric mode.

The rest of this paper is organized as following. Section II describes the Asian option pricing problem and presents two possible parallel implementations. In Section III we provide some background on PGAS and coarrays. In Section IV we present the implementation of the dynamic load balancing algorithms based on MPI and coarrays. We also highlight the problems we encountered during the analysis and related to MPI message progress. In Section V we describe the platform used for the experimental results. In Section VI we present some considerations on the parallel task scheduling problem that the Asian option pricing embodies. In Section VII we discuss the experimental results, using CPUs and Xeon Phis in different ways in order to exploit as much heterogeneity as possible. Finally, we present our conclusions and outline future work in Section VIII.

II. 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 [8]. 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 [9], numerical partial differential equation (PDE) techniques such as in [10], and various approximations such as those proposed by Turnbull et al. [11].

Using Monte Carlo simulation, multiple stochastic histories of the asset price are simulated based on the available information of the asset volatility [8], [12], [13]. 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.

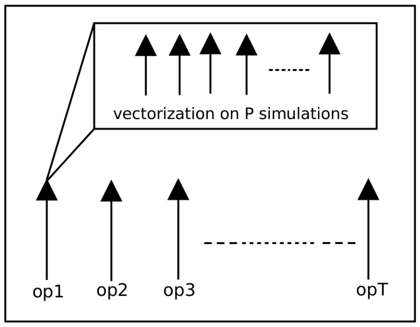

Fig. 1. T options running on T OpenMP threads (MO approach)

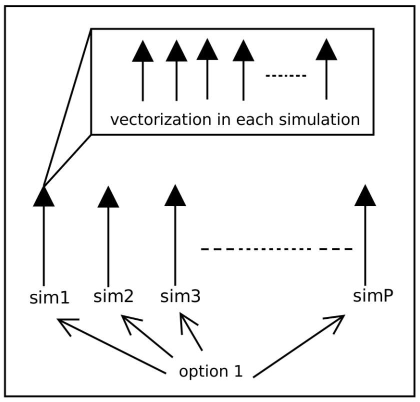

Fig. 2. P simulations of a single option running on P OpenMP threads (MT approach)

III. PGAS and OpenCoarrays

Partitioned Global Address Space (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 get the best from) the Single Program Multiple Data (SPMD) approach, used in the distributed memory systems, and the semantic of the shared memory systems. In the PGAS model, every process has its own memory address space but it can share a portion of its memory with other processes.

The most common PGAS languages include Coarray Fortran (CAF) [3], [14], Unified Parallel C (UPC) [4] and Chapel [5]. PGAS languages rely on one-sided communication semantics: a process can get/put data from/on a memory segment exposed by another remote process, without explicitly involving the application on the remote node. Several modern networks allow to implement these semantics 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 improvement. 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) [14]. 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.

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 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 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 to express more complex and useful mechanisms for synchronization, images organization and failure management. Technical Specification 18508 (TS 18508) [15], 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).

Teams allow to group images into non-overlapping teams 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. *New collectives and atomic intrinsics* provide intrinsic procedures for commonly used collective and atomic memory operations (e.g. *ATOMIC_FETCH_ADD*). Such 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: besides the Cray Fortran compiler, the Intel ifort, GNU Fortran, Rice compiler, OpenUH compiler, and the g95 compiler support coarrays. OpenCoarrays [16] is an open-source transport layer supporting Coarray Fortran compilers. Such library is currently the communication library used by the GNU Fortran compiler; it provides several implementations based on different communication layers, with the most complete and stable version being that based on MPI-3.0. OpenCoarrays already supports several coarray features listed in TS 18508 [15], including Events and the new atomic intrinsics like *ATOMIC_FETCH_ADD*.

Even though a user might want to directly use the MPI one-sided functions, there are several issues related with this approach: i) the syntax of the one-sided functions is much more complex and error-prone than coarray's; ii) using MPI explicitly, the code is strictly tight to a specific parallel programming system; using Coarrays, the syntax stays the same whereas the transport layer can transparently change. For 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. OpenCoarrays Compiler Wrapper

Currently, OpenCoarrays[1] comprises three components: 1) run-time library; 2) executable file launcher; 3) compiler wrapper.

The run-time library supports compiler communication and synchronization requests by invoking a lower-level communication library (MPI by default). It exposes an Application Binary Interface (ABI) that translates high-level communication and synchronization requests into low-level calls. The ABI is usually invoked directly by OpenCoarrays-aware compilers.

The file launcher simply passes execution to the chosen communication library's parallel program launcher (mpirun

[1]www.opencoarrays.org

by default). The only aim of the launcher is to mask the communication layer used by the run-time library.

The compiler wrapper aims to support CAF even on compilers that provide limited or no support for CAF. To do so, 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 invocations to specific procedures implemented in a Fortran 2008 module (opencoarrays module). These procedures adapt the arguments coming from the source code and invoke the run-time library using the OpenCoarrays ABI.

Using the structures and procedures declared in the OpenCoarrays module, any Fortran 2008 compiler is able to use the OpenCoarrays run-time library. Currently, the OpenCoarrays module supports only a subset of CAF (collectives, basic synchronization statements and intrinsics); for the purpose of this work, we added the *ATOMIC_FETCH_ADD* procedure to the OpenCoarrays module.

IV. 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 [17].

The Asian option pricing problem we described in Section II is a very good candidate for showing the effects of good load balancing between CPUs and Xeon Phis. Since the options are independent with 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 benchmark 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 them 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.

A. Dynamic Workload Scheduling based on MPI

The most performaning version of dynamic workload scheduling presented in [6] is based on a 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 *option_index* variable (which represents an option) 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 will use several threads to parallelize the compute intensive part of the Monte Carlo simulation.

B. 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, Coarray Fortran 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 we have described in Section IV-A.

Unfortunately, the CAF implementation provided by the Intel compiler, which is required to run on Intel Xeon Phi, 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 [16] 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.

C. MPI Passive One-sided Progress

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 routine to match the one-sided operations invoked by the source process. In practice, the matching between MPI features and the underlying network capabilities is not perfect and, even if the NIC allows to overlap communication with computation, the MPI implementation may not be able to progress independently [18].

We will now give a more detailed definition and description of *Progress* and *Overlap*, and of their impact on the performance of MPI applications; this section is based on [19].

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

Progress is a characteristic related to MPI, which in the software stack resides above the network layer. The MPI standard defines a Progress Rule for asynchronous communication operations; unfortunately, there are two different interpretations of this rule leading to different behaviors, both compliant with the standard.

The stricter interpretation of the Progress Rule is that, once a non-blocking communication operation has been posted, the subsequent posting of a matching operation will allow the original one to make progress, regardless of whether the application makes any further library call. In short, this interpretation mandates non-local progress semantics for all non-blocking communication operations once they have been enabled.

The weaker interpretation allows a compliant implementation to require the application to make further library calls in order to achieve progress on other pending communication operations.

In general, it is possible to support overlap without supporting independent MPI progress. For example, an InfiniBand network subsystem can usually perform RDMA operations, thus fully overlapping communication and computation; the price to be paid is that the target memory address has to be known. If the transfer to/from the target address depends on the user application making an MPI library call, then progress is not independent from the computation. Conversely, it is also possible to have independent MPI progress without overlap.

Asynchronous message progress is a very intricate and controversial topic in high-performance computing [19], [20], [21]. With the current available high-performance networks, there are essentially three strategies for making progress: manual progress, thread-based progress, and communication offload.

Hoefler et al. [21] 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). However, passing through the operating system raises two concerns: 1) it is unclear how large the interrupt latency and overheads are on a modern system; 2) the scheduler has to schedule the progress thread right after the interrupts to achieve asynchronous progress. This latter issue can be tackled by using real-time functionalities in the Linux kernel.

In [22], Si et al. propose to use dedicated communication processes (called ghost processes) for ensuring message progress, and employ the MPI-3 shared memory capability for transferring data from the ghost process to the target process.

In this work, we analyze the performance of both approaches; as shown in Figure 11, the performances 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 (where the master sends only one option per request) version into the CAF-based version, even when manual progress is implemented, does not provide good results because of the higher communication cost. To overcome this problem, the idea is to send more than one option per request to the workers; this simple solution can be very effective, but introduces several other problems discussed below.

V. 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 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.

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*. Figure 3 shows such configuration; note that communication through the Intel QuickPath Interconnect (QPI) will provide lower latency than communication on PCI Express.

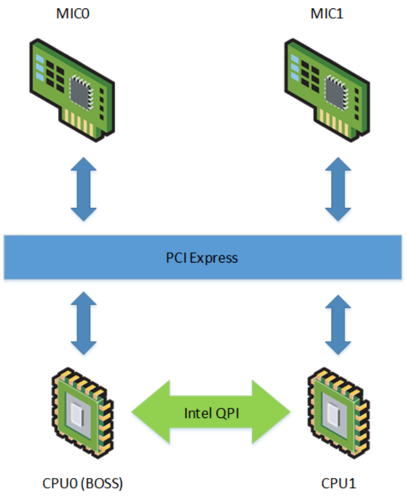

Fig. 3. Heterogeneous Galileo's single node

VI. 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 manufacturing industry.

Before answering the question on the right number of options per device, we need to analyze performance and consequences of the two possible solutions described in Section II, that is 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 (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 V.

A. 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 4 and 5 show the throughput while varying the number of options assigned to CPU and Intel Xeon Phi, respectively.

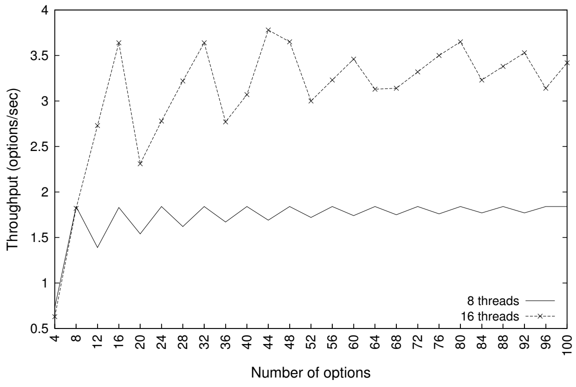

Fig. 4. CPU throughput

Each curve in Figures 4 and 5 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, in Figure 4 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.

B. 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.

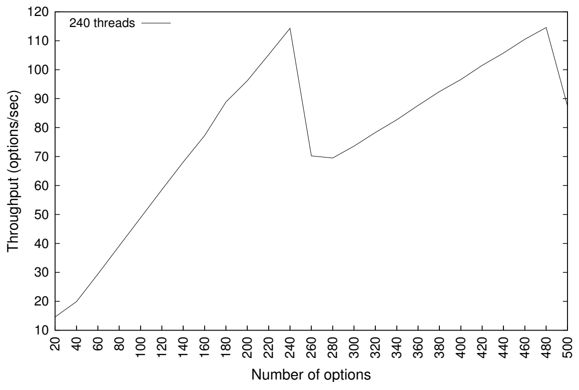

Fig. 5. Intel Xeon Phi throughput

C. 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 in the ordinary sense 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 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 homogenous 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 as 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 be now 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) [23]. 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 less idle 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 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 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, 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. 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 give to each process; on the other hand, this increases the risk of having idle devices towards the end of the scheduling.

Figures 6 and 7 show the final part of an instance of scheduling for the MO and MT approaches, respectively.

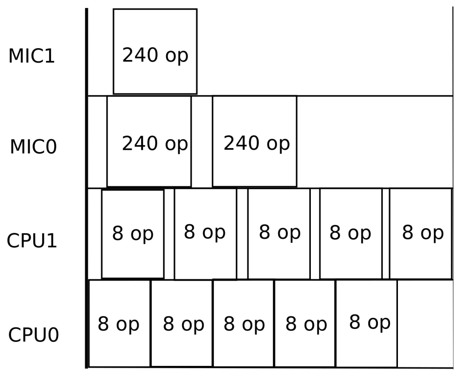

Fig. 6. Instance of scheduling for MO

As we said, for MO we cannot give to the devices less options than the number that ensures the maximum throughput; in our case, 240 for the Xeon Phis and 8 for the CPUs. Such restriction will most likely produce a situation like that depicted in Figure 6, where one or more devices will not be able to get enough options and they will spend all the remaining time in idle state.

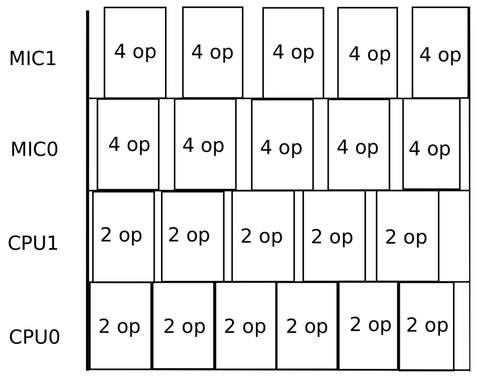

Fig. 7. Instance of scheduling for MT

On the other hand, for 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. In Figure 7 we have chosen to give 2 options to the CPUs and 4 options to the Xeon Phi. As we will show in Section VII-B, this configuration leads to good performance on our platform.

Figure 9 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).

D. 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 (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 8. From now on, we will refer to this hybrid version as MTH.

VII. Experimental Results

We first 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. Finally, in Sections VII-C and VII-D 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 fabrics.

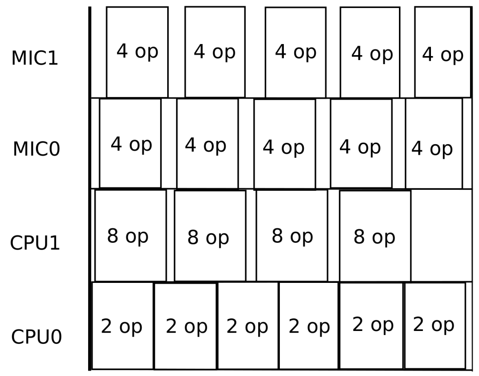

Fig. 8. Instance of scheduling for MTH

A. 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 9 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.

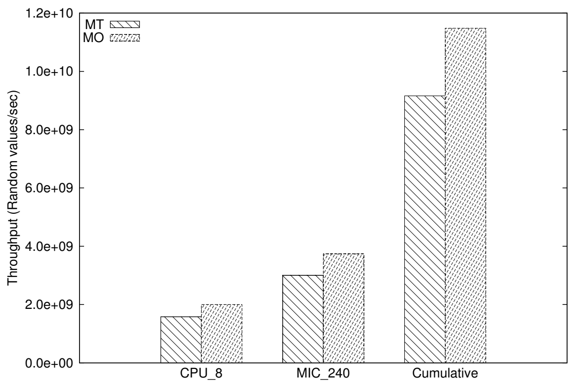

Fig. 9. Homogenous performance on CPU, Xeon Phi and theoretical heterogeneous throughput

The test shows that a single Intel MIC is about twice as fast than 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 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 the MO achieves 2.0 billions of random values per second while working on eight options in parallel.

B. 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 VI-C, 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 inversely related and both directly influenced by the job size. In fact, if we increase the number of options to send to a device, the communication costs will be lower (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 (job) negatively influences the 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 in a previous job by other devices.

In Figure 10 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 it is clear that we should assign no more than 3 options per CPU (and consequently no more than 6 options per MIC).

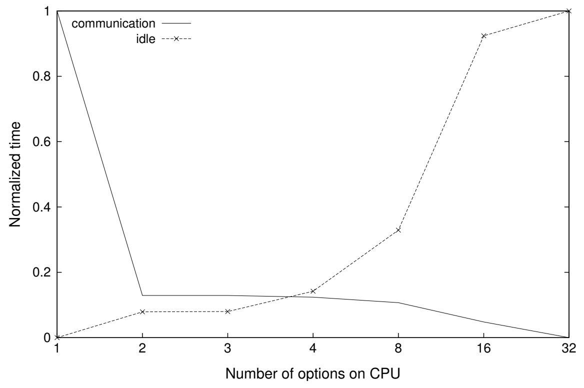

Fig. 10. Trade-off between communication and idle time

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 MICs), because a smaller granularity makes the application more flexible against possible performance changes on heterogeneous devices.

C. MT CAF-based Performance

In the MT CAF-based code, every process starts the computation by getting only one option from the master process and saving the processing time in a coarray variable, accessibile by any process. By doing so, each device understands how much time is 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 can repeat this phase (sampling the compute and/or remote communication time) several times 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 11 we compare the performance of the CAF-based versions with the MT MPI-based 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.

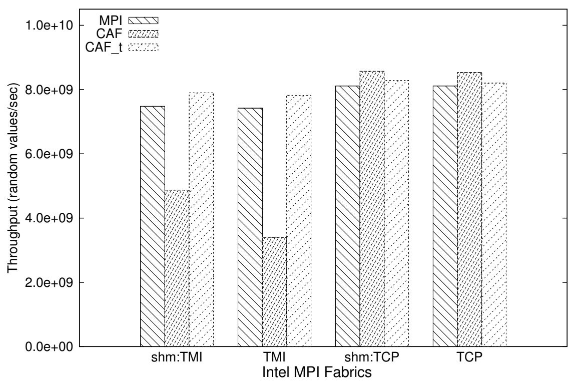

Fig. 11. Comparison of MPI vs. CAF using different MPI fabrics

Figure 11 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 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.

D. 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 from higher communication costs than CPU0 (the master process always runs on CPU0). This fact is related with 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 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 12 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 9 as the maximum performance reachable with the available hardware (represented by the dashed line), we can see that the MTH solution provides the closest performance to the maximum.

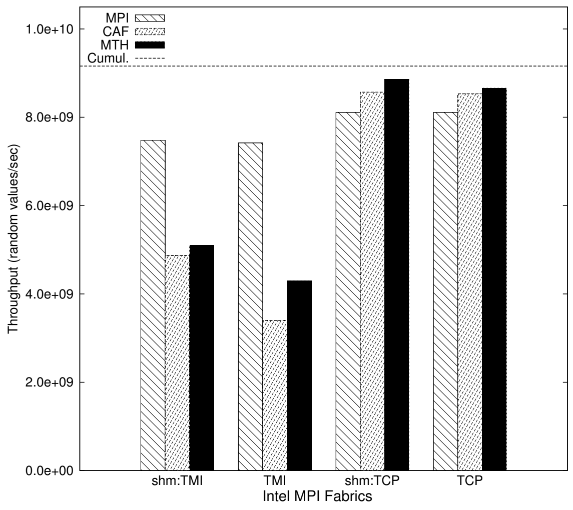

Fig. 12. Hybrid CAF-based performance using different MPI fabrics

VIII. Conclusions

In this work 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 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 for the tests is based on MPI-3.0, it is able to provide better performance than an explicit MPI two-sided implementation. The reason for that is due to the communication pattern required by the application, which is more suitable for a 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 VII-D, which leads to the best performance.

Although it is possible to implement efficient algorithms using explicitly MPI one-sided routines, 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.

Acknowledgment

We gratefully acknowledge the support received from CINECA for the access on Galileo for the project OpenCAF under the ISCRA grant program for 2015.

References

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

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

[3] R. Numrich and J. Reid, "Co-array Fortran for parallel programming," *SIGPLAN Fortran Forum*, vol. 17, no. 2, pp. 1–31, Aug. 1998.

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

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

[6] A. Vladimirov, R. Asai, and 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, 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. ACM, Oct. 2013.

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

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

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

[11] 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.

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

[13] 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.

[14] R. W. Numrich and J. Reid, "Co-arrays in the next Fortran standard," *SIGPLAN Fortran Forum*, vol. 24, no. 2, pp. 4–17, Aug. 2005.

[15] ISO/IEC/JTC1/SC22/WG5, "TS 18508 additional parallel features in Fortran," Aug. 2015.

[16] 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, Oct. 2014.

[17] 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, 2015.

[18] V. Cardellini, A. Fanfarillo, and S. Filippone, "Overlapping communication with computation in MPI applications," Università di Roma Tor Vergata, Tech. Rep. DICII RR-16.09, Feb. 2016, http://hdl.handle.net/2108/140530.

[19] R. Brightwell and K. D. Underwood, "An analysis of the impact of MPI overlap and independent progress," in *Proc. of 18th Ann. Int'l Conf. on Supercomputing*, ser. ICS '04. ACM, 2004, pp. 298–305.

[20] T. Hoefler, G. Bronevetsky, B. Barrett, B. R. D. Supinski, and A. Lumsdaine, "Efficient MPI support for advanced hybrid programming models," in *Recent Advances in the Message Passing Interface*, ser. LNCS, vol. 6305. Springer, 2010, pp. 50–61.

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

[22] 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 *Proc. of 29th IEEE In'l Parallel & Distributed Processing Symp.*, ser. IPDPS '15, May 2015, pp. 665–676.

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