Heterogeneous CAF-based Load Balancing on Intel Xeon Phi

Valeria Cardellini[1], **Alessandro Fanfarillo**[12], Salvatore Filippone[3]

[1]University of Rome Tor Vergata, Rome, Italy

[2]National Center for Atmospheric Research, Boulder, CO, USA

[3]Cranfield University, Cranfield, UK

May 23rd, 2016

Introduction

Monte Carlo method for pricing Asian options (embarrassingly parallel algorithm).

Original code taken from *Parallel programming and optimization with Intel Xeon Phi coprocessors**.

Original code assumed correctly optimized for Intel Xeon Phi architecture.

Xeon Phis and CPUs used in *symmetric mode* (each device considered as a compute node).

Approach presented by Colfax based on Master-Slave paradigm using MPI two-sided functions.

We investigate the potential of PGAS languages for dynamic load balancing problems on heterogeneous nodes.

Proof of concept for dynamic load balancing on a single heterogeneous node using a PGAS language.

* Colfax International (http://www.colfax-intl.com/)

Asian Option Pricing

Options are contracts which allow one party to buy or sell, on some future date, an asset (e.g., a stock). "strike price" agreed upon the signing of contract.

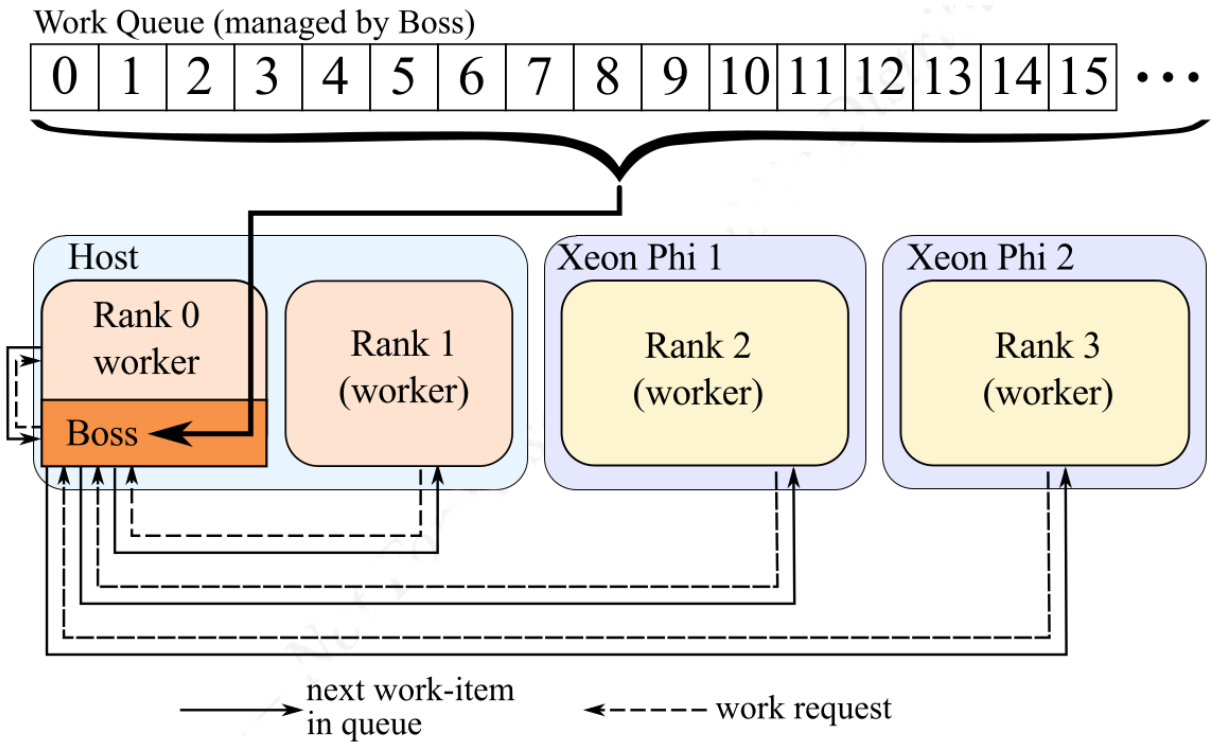
Asian options payoff is calculated based on the mean price of the asset, sampled at prearranged instances.

To make profit, the seller of the option must set a price that offsets the anticipated risks associated with the price fluctuations.

For risk analysis of Asian options, multiple stochastic histories of the asset price are simulated based on the available information.

Our task is to price N options, where for each option we have different sets of parameters such as starting price, volatility, time averaging interval, etc. For each option, we simulate P random paths and perform statistical analysis using these simulations.

Dynamic Load Balancing based on MPI two-sided



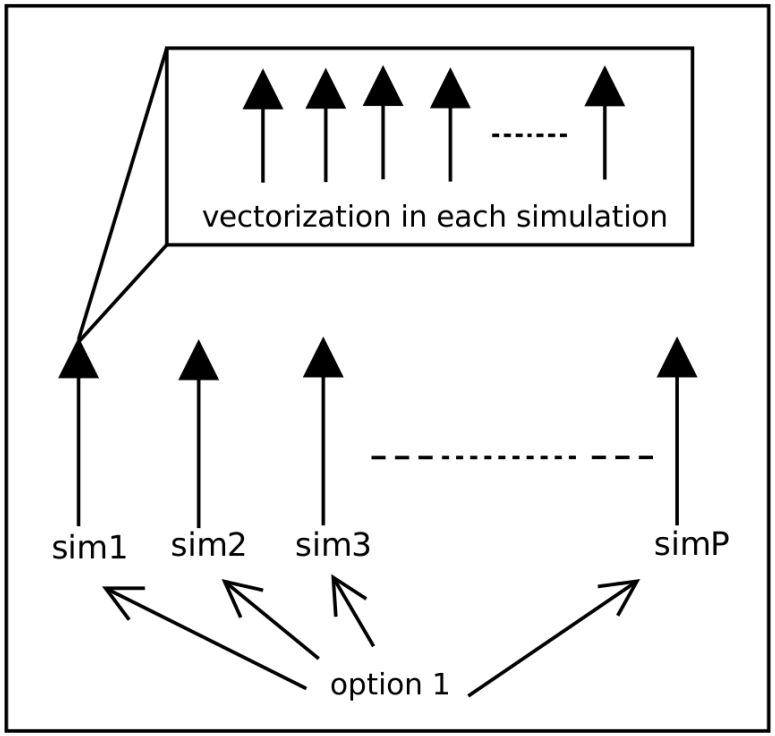
Copyright Colfax International

The boss is implemented by one thread of the CPU.

The boss thread blocks on a MPI_Recv function in an infinite loop.

Each worker asks for one option at time.

Simulations On Each Device (Multi-threaded)



Single option composed by P random simulations.

Each simulation can be executed by a single OpenMP thread.

Each thread applies vectorization techniques to speedup the computation.

PGAS-based Dynamic Load Balancing

Issues with MPI-based dynamic load balancing:

- the boss thread is able to satisfy only one request at time;
- low-level details involved in the algorithm;
- getting one option at time requires a lot of communication.

Partitioned Global Address Space (PGAS) languages may be a good alternative:

- allow to access remote memory directly (no need for boss thread);
- allow to write more complex and portable algorithms.

In this work we used Coarray Fortran (CAF) as PGAS language.

PGAS Languages

The PGAS model assumes a global memory address space that is logically partitioned and a portion of it is local to each process or thread.

It means that a process can directly access a memory portion owned by another process.

The model attempts to combine the advantages of a SPMD programming style for distributed memory systems (as employed by MPI) with the data referencing semantics of shared memory systems.

- **Fortran 2008/2015 (coarrays)**
- UPC (upc.gwu.edu)
- Titanium (titanium.cs.berkeley.edu)
- Chapel (Cray)
- X10 (IBM)

Compilers Supporting Coarray Fortran

- Cray Compiler (Gold standard - Commercial)
- Intel Compiler (Commercial)
- **GNU Fortran (Free - using OpenCoarrays)**
- Rice Compiler (Free - Rice University)
- OpenUH (Free - University of Houston)
- G95 (coarray support not totally free - Not up to date)

GFortran uses an external library to support coarrays (since GCC 5.1).

Each coarray operation is translated in a function invocation (ABI).

The OpenCoarrays library implements this ABI using several transport layers (since August 2014).

Currently, the most complete and stable implementation is based on MPI-3.1 using passive one-sided communication functions.

Coarray example

```
real, dimension(10), codimension[*] :: a, x, y
integer :: num_img, me, old_counter, increment
integer(atomic_int_kind) :: counter[*]

num_img = num_images()
me = this_image()
counter = 0; increment = me

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

sync all

call atomic_fetch_add(counter[1],increment,old_counter)
```

CAF-Based Dynamic Load Balancing - The Idea

We span one image on each device, Image 1 (on CPU 0) keeps the counter of computed options.

Each image performs an `ATOMIC_FETCH_ADD` on the atomic counter on Image 1.

This one-sided approach (in theory) liberates CPU0 from using one thread for communication and allows the worker images to pick more than one option at time directly.

Issues:

1. The `ATOMIC_FETCH_ADD` intrinsic is supported by GFortran + OpenCoarrays but not yet by the Intel compiler.
2. Taking more than one option at time has an impact on performance.
3. Technical limitations due to MPI asynchronous progress.

OpenCoarrays

OpenCoarrays is composed by three parts:

- **Compiler wrapper**: aims to support CAF even on compilers that provide limited or no support for CAF.
- **Run-time library**: supports compiler communication and synchronization requests by invoking a lower-level communication library (MPI by default).
- **Executable file launcher**: passes execution to the chosen communication library's parallel program launcher (mpirun by default).

In this work we used the compiler wrapper in order to use the CAF features provided by OpenCoarrays through the Intel compiler (needed by Xeon Phis).

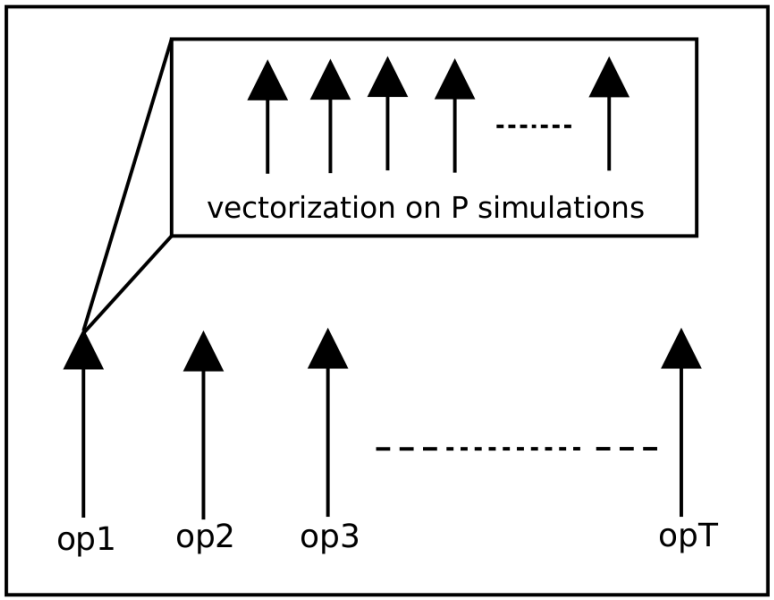
Multiple Options per Communication

Taking more than one option at time reduces the communication penalty (due to latency) and should improve the performance, but it poses new questions:

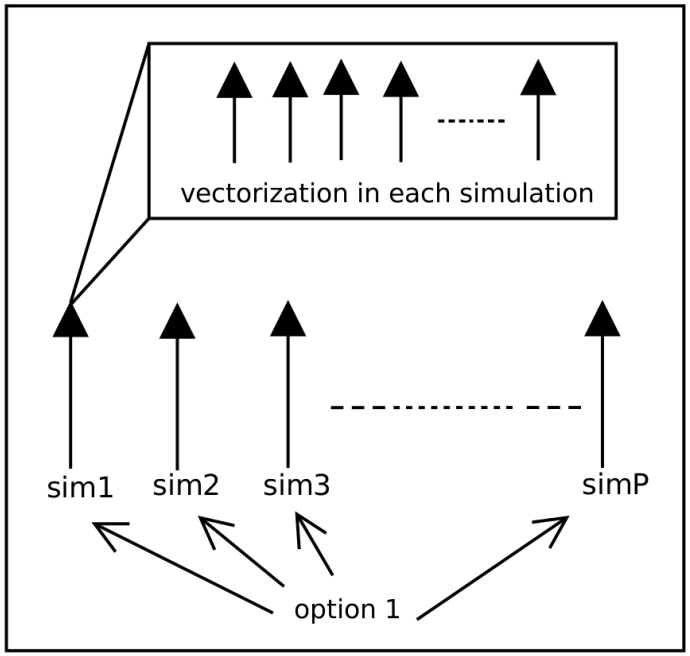
1. each device should keep the options queued or run them in parallel?
2. how many options should be taken by each device?

This questions can be answered by applying well known principles related to *scheduling* problems.

To Queue or Not to Queue? (1)



Multiple Options (MO)



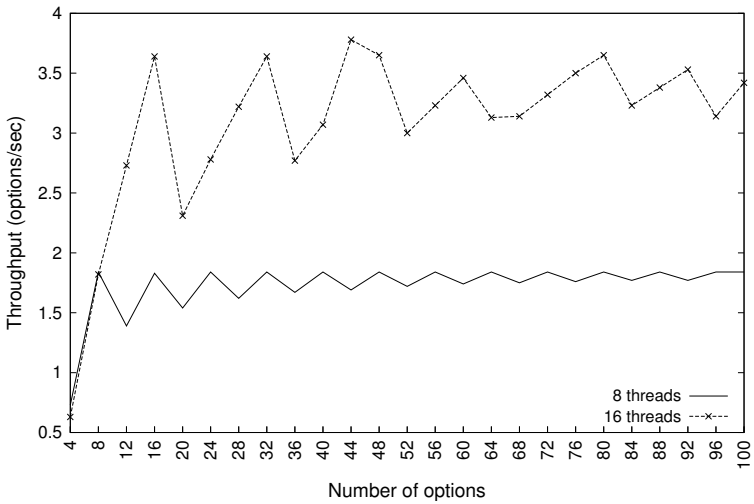
Multi-threading (MT)

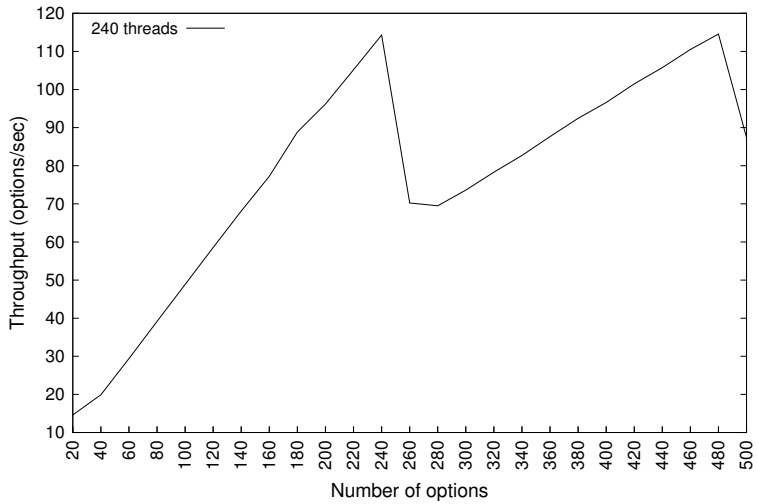
MO runs several options in parallel (OpenMP).

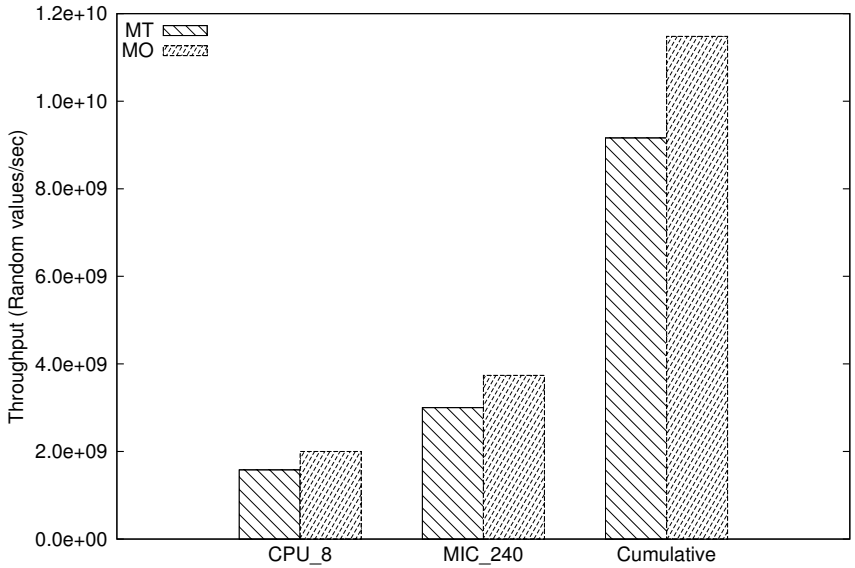
MT runs several simulations in parallel belonging to one option.

To Queue or Not to Queue? (2)

Throughput on CPU and Xeon Phi using the MO approach

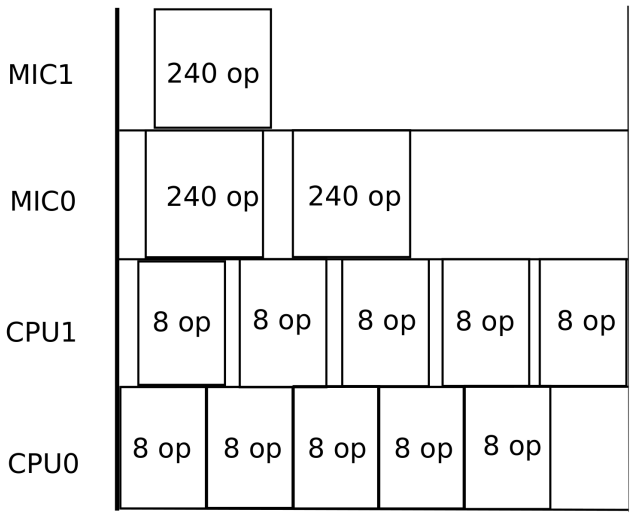


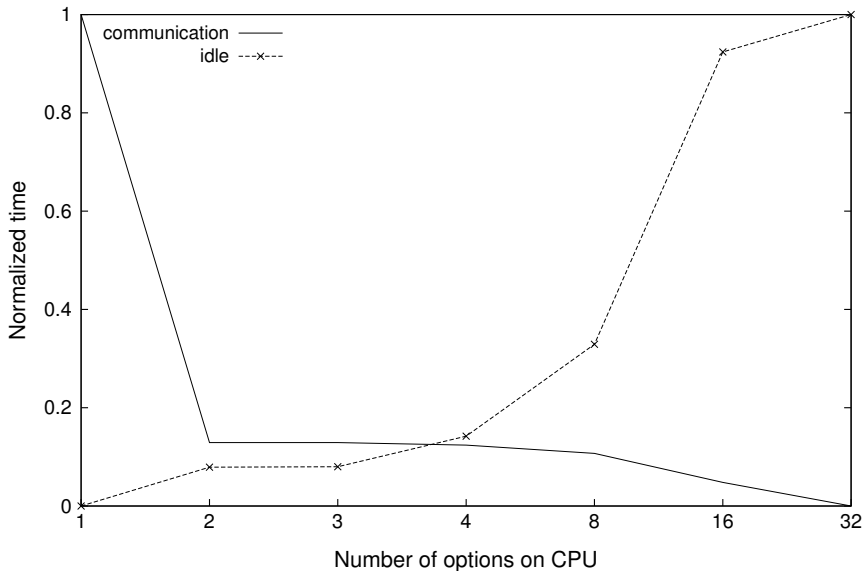




- MO reaches max throughput when all threads are used.
- MT reaches max throughput with a single option.
- MO has higher throughput than MT.

To Queue or Not to Queue? (3)





With an MO approach, each Xeon Phi needs 240 options in order to reach its max throughput.

This has a huge impact on the scheduling granularity.

Because the MT approach provides max throughput event with a single option we decided **to queue**.

How Many Options per Device?

This is a well known scheduling problem: makespan minimization using heterogeneous devices without preemption (NP-Hard).

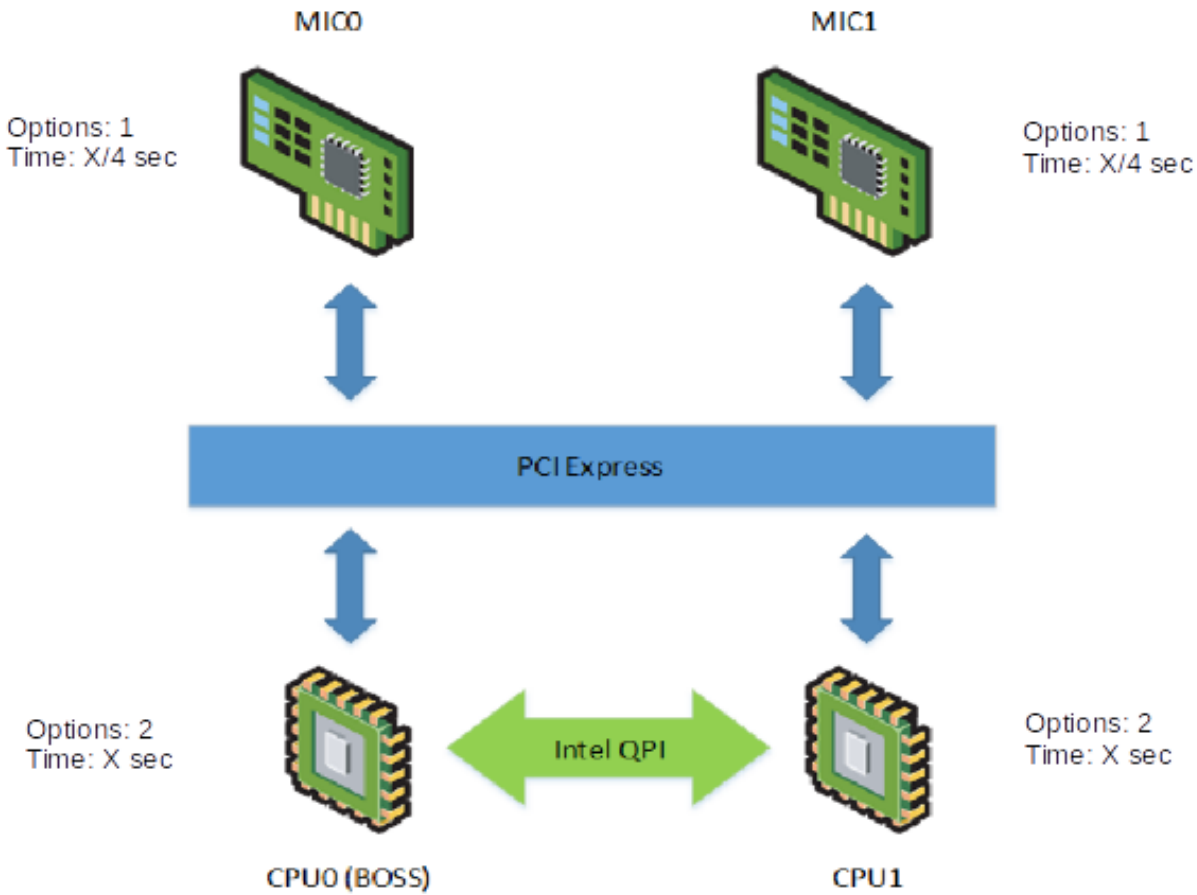
In order to simplify the problem we try to simulate homogeneity among the heterogeneous devices: **all devices take the same time to complete a different amount of work**.

Assuming a MT approach on all devices, it is preferrable to keep the number of options on CPUs between 2 and 3.

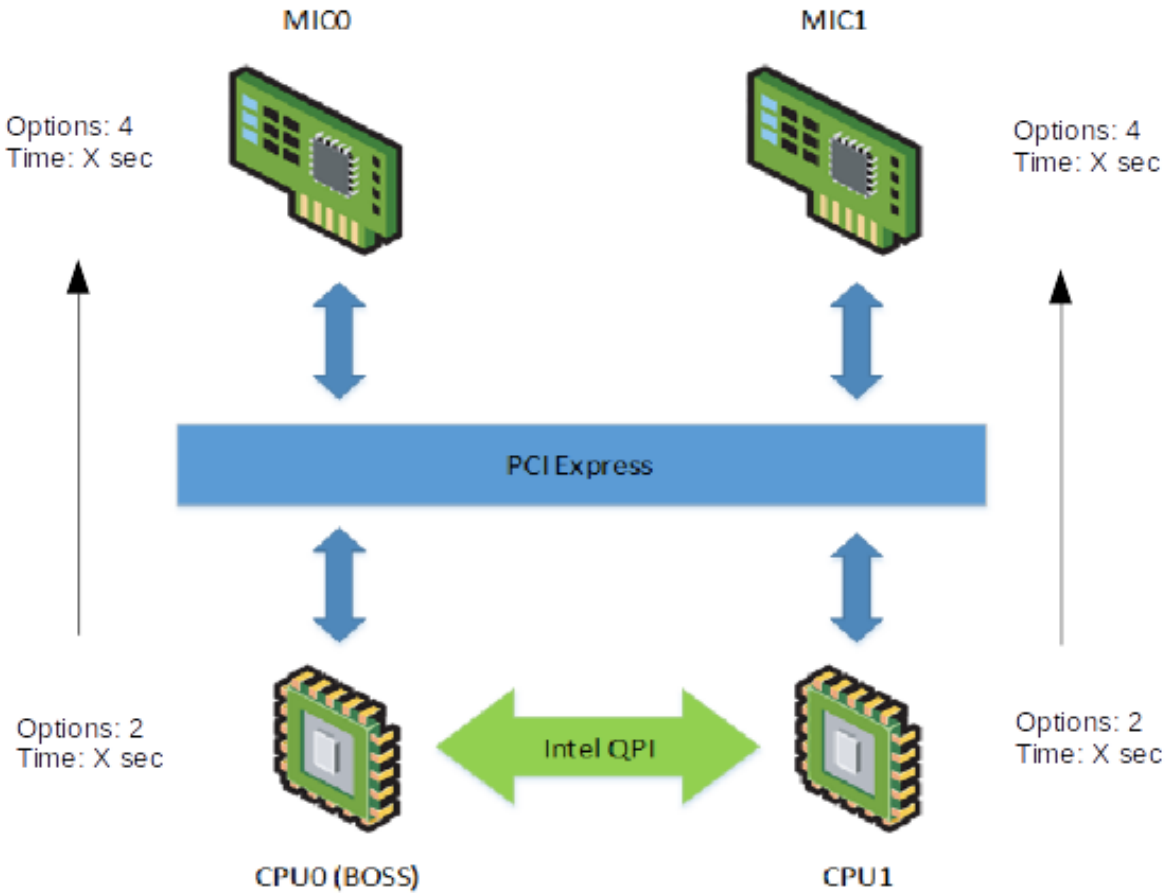
Xeon Phis can "adapt" by checking how much time is required by a CPU to compute a certain amount of options.

The time spent for computing a fixed amount of options and the number of options currently analyzed can be stored in coarray variables.

Learning Phase



Learning Phase



Experimental Platform

Tests run on Galileo, Tier 1 system operated by Cineca: the Italian supercomputing consortium.

Each node equipped with two 8-core Intel Haswell E5-2630 v3 @ 2.40 GHz.

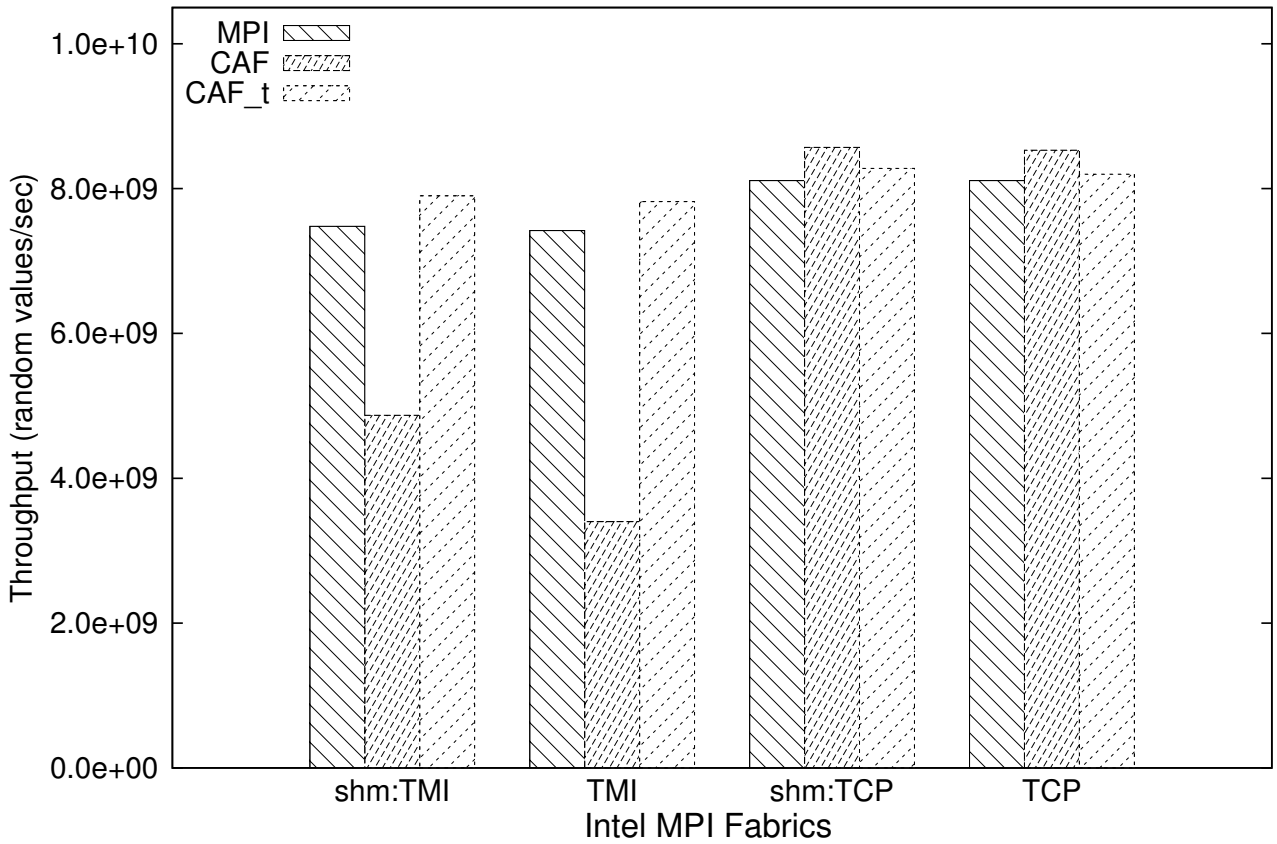
About half nodes are also equipped with two Intel Xeon Phi 7120p.

Code compiled with the Intel Fortran Compiler 15.0.2 and IntelMPI-5.0.2.

Coarray code based on OpenCoarrays-1.0.0, compiled with the Intel Compiler through the compiler wrapper module.

Note: The code is assumed to be optimized for the Intel Xeon and Xeon Phi architectures (Colfax International).

MPI vs. CAF Dynamic Load Balancing

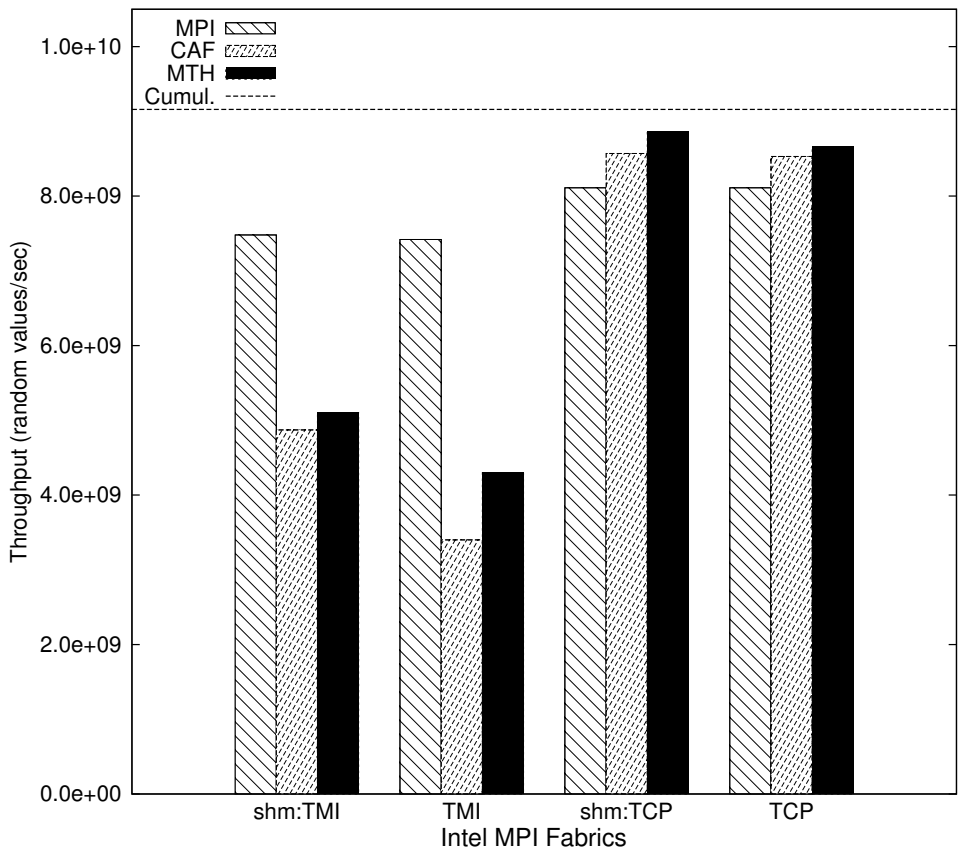


The MPI implementation capabilities and the network fabric are critical factors.

MT Hybrid approach

How about using MO only on one CPU (CPU1) and MT on the remaining devices?

Reduces communication costs and has acceptable impact on schedule granularity (it forces 8 options on CPU1).



Conclusions

- The ease of programming provided by coarrays and their one-sided semantics allowed us to focus more on the algorithm design rather than its implementation (no low level details).
- We were able to manage heterogeneous situations, where two different versions of the same code run, at the same time, on the hw more suitable for the performance needs.
- Although the same results can be obtained by using MPI one-sided routines, coarrays make easier to program more complex parallel algorithms.
- The one-sided support provided by Intel MPI is strongly related with the "network" fabric employed.

Thanks

MPI Asynchronous Progress

MPI One-Sided Progress

One-sided functions exposed by MPI-3.0, are 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.

MPI progress is guaranteed when the application invokes some MPI routines.

Progress Approaches

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

- Manual progress is performed by the programmer by manually invoking MPI routines during the program execution (adopted by OpenCoarrays).
- Thread-based progress requires a thread-safe MPI implementation; it uses an helper thread that continuously invokes the MPI library.
- Communication offload delegates the MPI progress to hardware components; although this solution allows independent progress, it may become a bottleneck because of the lower performance of the embedded processors compared to regular CPUs.