Heterogeneous Clustering with Homogeneous Code: Accelerate MPI Applications Without Code Surgery Using Intel Xeon Phi Coprocessors

Andrey Vladimirov and Vadim Karpusenko
Colfax International

December 9, 2013

Abstract

This paper reports on our experience with a heterogeneous cluster execution environment, in which a distributed parallel application utilizes two types of compute devices: those employing general-purpose processors, and those based on computing accelerators known as Intel Xeon Phi coprocessors.

Unlike general-purpose graphics processing units (GPGPUs), Intel Xeon Phi coprocessors are able to execute native applications. In this mode, the application runs in the coprocessor's operating system, and does not require a host process executing on the CPU and offloading data to the accelerator (coprocessor). Therefore, for an application in the MPI framework, it is possible to run MPI processes directly on coprocessors. In this case, coprocessors behave like independent compute nodes in the cluster, with an MPI rank, peer-to-peer communication capability, and access to a network-shared file system. With such configuration, there is no need to instrument data offload in the application in order to utilize a heterogeneous system comprised of processors and coprocessors. That said, an MPI application designed for a CPU-only cluster can be used on coprocessor-enabled clusters without code modification.

We discuss the issues of portable code design, load balancing and system configuration (networking and MPI) necessary in order for such a setup to be efficient. An example application used for this study carries out a Monte Carlo simulation for Asian option pricing. The paper includes the performance metrics of this application with CPU-only and heterogeneous cluster configurations.

Table of Contents

1. Introduction: To Offload or To Take It Easy, Aye There's the Point

Intel Xeon Phi coprocessors, featuring the Intel Many Integrated Core (MIC) architecture, are computing accelerators bearing considerable similarity to general-purpose graphics processing units (GPGPUs):

i) both types of accelerators are connected to a CPU-based system via a PCIe bus;
ii) they both require data and task parallelism to deliver greater performance and better power economy than general-purpose processors;
iii) coprocessors and GPGPUs support the so-called offload programming model.

In the offload programming model, the application is launched on the host system (i.e., using the CPU), and initialization of data also takes place on the host. Then the application uses the PCIe bus to push ("offload") a part of the data and specialized executable code to the device for processing. After processing, results are pulled back to the host[1]. In the Nvidia CUDA framework, offload is effected through specialized functions and streams. For Intel Xeon Phi coprocessors, the programmer uses compiler pragmas to initiate offload. Finally, for both accelerators, OpenCL means of offload are available. In order to instrument offload in an application designed for general-purpose CPUs, the programmer must devise a work-sharing strategy, prepare data structures for offload, and outfit the code with offload directives. Offload programming can be used on standalone machines, as well as in clusters, where on each machine, one or several MPI processes are launched, each performing offload (see Figure 1).

However, two substantial circumstances set Intel Xeon Phi coprocessors apart from GPGPUs:

i) Xeon Phi coprocessors run a limited version of a Linux operating system, called uOS[2], which makes them IP-addressable devices with a virtual file system, capable of running workhorse HPC services including SSH, NFS and MPI.
ii) Coprocessors also support native programming model, in which the application is launched directly on the device, and all data initialization and I/O take place there.

Native programming opens possibilities for architecting distributed applications in ways not possible with GPGPUs:

- Distributed applications in the MPI framework may run exclusively on coprocessors, leaving the host CPU free for other tasks;
- Alternatively, one may place MPI processes on host CPUs as well as on coprocessors, for a computation on a heterogeneous platform (see Figure 2).

The latter option is especially attractive for enabling existing MPI applications to use Intel Xeon Phi coprocessors, because in this model, the offload mechanism does not have to be instrumented. Processes running natively on Intel Xeon Phi coprocessors will initialize themselves and participate in communication just like processes running on hosts.

Naturally, such porting with zero programming effort will not always yield accelerated performance "out of the box". In this paper, we analyze the types of applications that can be efficiently executed on heterogeneous clusters, and the prerequisites for their implementation. We also describe the system configuration that enables heterogeneous execution with Intel Xeon Phi coprocessors. As a proof of concept, we implement a Monte Carlo method of Asian stock option pricing in the C language. This code is designed for a CPU-based cluster, but can be executed on a heterogeneous cluster with Intel Xeon Phi coprocessors with zero coding effort. The latter condition means that nothing in the code of the application indicates that it is designed to use the Intel Xeon Phi architecture, and yet, significant acceleration is observed when coprocessors are added to the hardware configuration.

2. Asian Option Pricing

In Section 3, we discuss the implementation of a distributed application in the MPI framework. This application is a Monte Carlo simulation that solves the

[1] Hereafter, "host" means the operating system running on the CPU-based platform, or the CPU platform itself, and "accelerator" or "device" mean the GPGPU or coprocessor.

[2] uOS is a common spelling of μOS, which stands for "micro operating system"

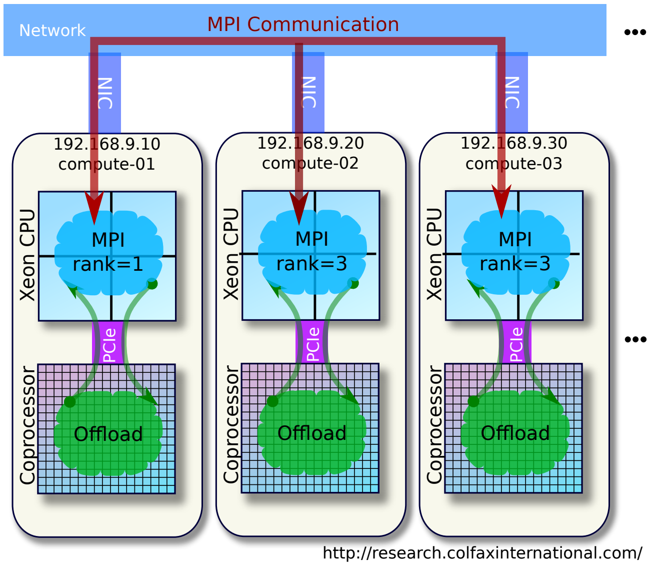

Figure 1: MPI communication in a traditional cluster with offload to coprocessors. Hosts can communicate with remote hosts using MPI and with *local* coprocessors using offload.

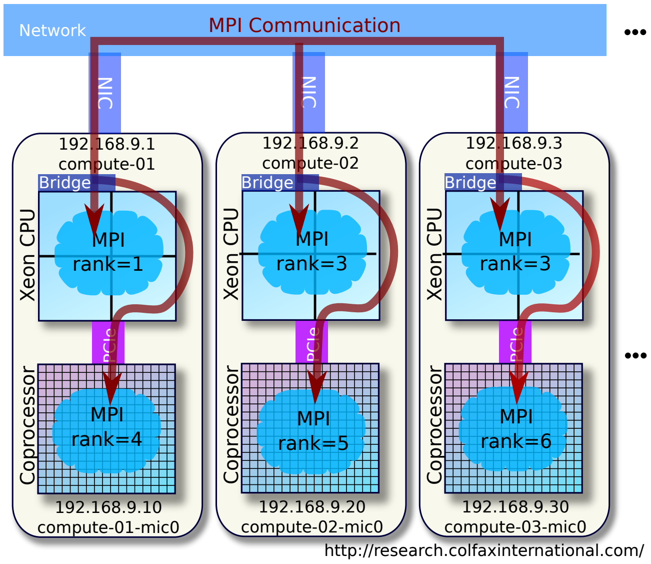

Figure 2: MPI communication in a heterogeneous cluster. Hosts can communicate with remote hosts and with *local or remote* coprocessors via MPI messages.

Asian option pricing problem. In this section we outline the financial problem and the mathematical model involved in the simulation. Readers interested only in the HPC aspect of this work may skip to Section 3.

Options are contracts which allow one party (called "beneficiary") to buy ("call option") or sell ("put option"), on some future date ("option expiration date"), a stock market asset from/to the other party ("grantor") at a "strike price" agreed upon the signing of contract. A contract to buy is called a "call option", and a contract to sell is a "put option". Unlike a futures contract, an option gives the beneficiary the right to choose whether to exercise the transaction. This choice is typically made based on the market price of the asset at the option expiration date. For example, if at a call option expiration date, the stock market price of the asset is higher than in the option contract, the beneficiary will elect to buy the asset from the grantor, and sell it on the market, which yields a profit called a "payoff". Otherwise, the beneficiary does not exercise the option, but the grantor retains the option fee. A style of options called Asian options has the feature that the payoff is calculated based on the mean price (arithmetic or geometric) of the asset, sampled at prearranged instances. This reduces the risks associated with market volatility and short-term market manipulation.

In order to perform risk analysis and to price an Asian option, a Monte Carlo simulation method can be used. In this method, multiple stochastic histories of the asset price are simulated based on the available information on the asset volatility.

Variable $S(t)$ is the price of the underlying asset for the option, which is assumed to evolve in time according to the stochastic equation

$$dS(t) = \mu S(t)dt + \sigma S(t)dB(t). \tag{1}$$

In this equation, μ is the drift of the asset, σ is the option volatility, and $B(t)$ is a standard Brownian motion.

The solution of this stochastic differential equation can be written as

$$S(t_i) = S(t_{i-1})e^{(\mu-\sigma^2/2)\Delta t+\sigma\sqrt{\Delta t}\,\mathcal{X}}, \tag{2}$$

where $\mathcal{X}$ is a normally distributed random variable with zero mean and unit standard deviation, and $\Delta t = t_i - t_{i-1}$.

In order to calculate the Asian option payoff for this asset, the asset price is averaged over the expiration time T at N instants:

$$\langle S\rangle_{\text{arithm}} = \frac{1}{N}\sum_{i=0}^{N-1} S(t_i), \quad (3)$$

$$\langle S\rangle_{\text{geom}} = \exp\left(\frac{1}{N}\sum_{i=0}^{N-1}\log S(t_i)\right), \quad (4)$$

$$(5)$$

for the arithmetic and geometric means, respectively, where $t_i = T \times i/(N-1)$. The corresponding payoffs for the "call" and "put" options for strike price K are

$$P_{\text{put}} = e^{-rT}\max\{0; K - \langle S\rangle\}, \quad (6)$$

$$P_{\text{call}} = e^{-rT}\max\{0; \langle S\rangle - K\}, \quad (7)$$

where $\langle S\rangle$ can be either $\langle S_{\text{arithm}}\rangle$, or $\langle S_{\text{geom}}\rangle$, depending on the contract, and r is the risk-free rate.

In order to numerically determine the mathematical expectation of the Asian option payoff for a set of parameters $\{S, K, r, \mu, v, T, N\}$ and averaging rules (arithmetic or geometric), a Monte Carlo simulation may be used. The simulation plays out a large number M of random paths, each of which stochastically evolves the option price from $t = 0$ to $t = T$ according to Equation (2), and computes the means over N time points according to Equation (3) or (4). These means are then used to calculate the put and call option payoffs according to Equations (6) and (7). Finally, these payoffs are averaged over the M random paths, producing a Monte Carlo estimate of their mathematical expectation.

3. Implementation for a Traditional Cluster

Like any Monte Carlo simulation, our Asian option pricing application has a parallel nature. The first opportunity for parallelism is that, when multiple sets of parameters $\{S, K, r, \mu, v, T, N\}$ need to be processed, each parameter set can be studied independently. In our implementation, we use this property to distribute distinct parameter sets across MPI processes. The second parallel property is that, for each parameter set, the simulated stochastic paths of asset evolution are independent from one another. Our implementation takes advantage of this fact by distributing the simulated paths across the processor cores and across the SIMD (Single Instruction Multiple Data) lanes of each core. The latter mode of parallelism is also known as vectorization.

Section 3.1 demonstrates the C language code of a thread-parallel calculation in which a single set of option parameters is analyzed on a single shared-memory compute device. In Section 3.2, we discuss the corresponding work distribution scheme, which scales the application across MPI processes in a distributed-memory system.

3.1. Thread Parallelism and Vectorization

Listing 1 is an outline of the performance-critical part of the calculation. Only the "put" option with arithmetic mean is shown. Full code is available for download [1]. This code is optimized for a multi-core processor with a SIMD instruction set, such as an Intel Xeon CPU.

The aspects of our implementation most important for performance are:

1. The workload is parallelized by distributing the calculations of random paths across threads using the OpenMP framework (see lines 3-4). The OpenMP scheduler takes care of load scheduling across threads. By default, all cores available on the CPU will be utilized for the calculation. Therefore, only one such process must be started per compute node.
2. Within each thread, multiple paths are computed concurrently in the SIMD lanes of the core if the compiler implements automatic vectorization in the loops with the index `k` (specifically, the performance-critical loop in line 13). The Intel C compiler is capable of this procedure. The instruction set for this data parallelism is chosen at the compile time according to the specifications of the platform on which the code is compiled.
3. Random number generation is performed using the Intel Math Kernel Library (MKL) in line 8. The Mersenne twister-based random number generator used in our implementation is vectorized, i.e., it takes advantage of the available SIMD instruction set. Each OpenMP thread is maintaining a private random number stream.

```
 1 /* The i-loop is thread-parallel, i.e.,
 2  distributed across the processor cores */
 3 #pragma omp parallel for schedule(guided) \
 4   reduction(+: payoff_arithm_put)
 5 for (int i = 0; i < nPaths/vecSize; i++) {
 6   for (int j = 1; j < nIntervals; j++) {
 7 /* Intel MKL random number generator */
 8     vsRngGaussian(
 9       VSL_RNG_METHOD_GAUSSIAN_BOXMULLER,
10       stream, vec_size, rands, 0.0f, 1.0f);
11 /* The k-loop is data-parallel thanks to
12  automatic vectorization by the compiler */
13     for (int k = 0; k < vecSize; k++) [
14       spot_prices[k] *=
15         exp2f(drift + vol*rands[k]);
16       sumsm[k] += spot_prices[k];
17     }
18   }
19   for (int k = 0; k < vecSize; k++) {
20     arithm_mean_put[k] =
21       K - (sumsm[k] * recipIntervals);
22     if (arithm_mean_put[k] < 0.0f)
23       arithm_mean_put[k] = 0.0f;
24   }
25 /* Reduction across vector lanes and across
26  OpenMP threads is automatically implemented
27  by the compiler */
28   for (int k = 0; k < vecSize; k++)
29     payoff_arithm_put += arithm_mean_put[k]*
30       expf(-r*T)/(float)nPaths;
31 }
```

Listing 1: A Monte Carlo method of Asian Option pricing.

4. There is no communication between threads during the calculation. However, at the end of the calculation, the results of the random paths are reduced across SIMD lanes and across OpenMP threads. The OpenMP library and the auto-vectorizer implement this operation expressed in lines 29-30 of the code.

All optimizations in the code in Listing 1 are performed for general-purpose multi-core processors. There is no indication in the code that it is specifically targeted to Intel Xeon Phi coprocessors. However, as we will see in Section 4.4, the same exact code can be efficiently run on these coprocessors after a recompilation pass. In this sense, the code in Listing 1 can be labelled “many-core-ready”[3].

3.2. Dynamic Load Balancing with the Boss-Worker Model

For a realistic application, more than one set of option parameters must be priced. In order to scale this application across a computing cluster, we partition the workload so that one or several sets of option parameters are processed on each compute node.

```
if (myRank == bossRank) {
  int nR = 0; /* Number of processed tasks */
  int iP = 0; /* Next task to assign */
  while (nR < nPars) {

 /* Wait for any worker to report for work */
    float buf[msgReportLength];
    MPI_Recv(&buf, msgReportLength,
      MPI_INT, MPI_ANY_SOURCE, msgReportTag,
      MPI_COMM_WORLD, &status);
    const int iW = status.MPI_SOURCE;

    if (buf[0] > 0.0f) {
 /* If worker reports with results of a
    previous task, record these results */
      nR++;
      const int iR = floorf(buf[1]);
      payoff_arithm_put [iR] = buf[2];
    }

    if (iP < nStrikes) {
 /* Assign the next task iP to worker iW */
      float buf[msgSchedLen] = {iP,
        M[iP], N[iP], K[iP], S[iP], /*...*/};
      MPI_Send((void*)&buf, msgSchedLen,
        MPI_FLOAT, iW, msgSchedTag,
        MPI_COMM_WORLD);
      iP++;
    }
  }
}
```

Listing 2: The boss process implementation. Load balancing is achieved dynamically, with the boss process receiving and satisfying workers’ requests for work items.

Generally, there is no guarantee that each parameter set takes the same amount of time to process on any computing device. Indeed, the calculation time for a parameter set is proportional to $M \times N$, where M is the number of Monte Carlo paths required to achieve the desired accuracy, and N is the number of time intervals for price averaging. The values of M and N may vary across the studied parameter sets. We resolve this problem by instrumenting a dynamic load balanc-

[3] Here, we adopt a convention where “multi-core” denotes the architecture of general-purpose processors featuring more than one core (such as Intel Xeon CPUs), and “many-core” to denote the Intel MIC architecture, specifically, Intel Xeon Phi coprocessors.

ing mechanism. This method of work scheduling will also be helpful when coprocessors are employed in a heterogeneous cluster with Xeon Phi coprocessors (see Section 4).

A relatively simple and well-known method of dynamic work scheduling is the "boss-worker model". In this model, one process ("boss") is occupied with the task of receiving and satisfying requests for work from multiple other processes ("workers"). When the boss receives a request for work, it sends to the reporting worker a "work item" to compute, which in our case is a set of option parameters for the Monte Carlo simulation. Once the worker processes the work item, it communicates to the boss again to report the results and request the next work item. Because workers are loaded with work as soon as they become idle, this scheduling mechanism balances the load across workers even for non-uniform costs of work items, or non-identical performance of workers. There are limitations on the scalability of the boss-worker scheduling method, as discussed in Section 5.2. However, this method is sufficient for our purposes.

For the Asian option pricing application, the functional part of the boss process code in the C language is shown in Listing 2. Full code is available for download [1]. In this implementation, the boss process is a single-threaded process with the MPI rank equal to 0, and workers are all other MPI processes in the MPI_WORLD communicator.

3.3. Compilation and CPU-Only Execution

We run the application on a test cluster consisting of a head node and two compute nodes with Intel Xeon processors (see Figure 3). The compute nodes NFS-import from the head node two directories: /opt/intel (to give the compute nodes access to the Intel MKL and Intel MPI libraries) and /nfs (to share the executable file and any necessary data files with the compute nodes).

As Figure 3 indicates, we are going to place one worker on each compute node. In addition, on node compute-01, we will place the boss worker. Listing 3 demonstrates how our simulation is compiled and executed on a test cluster.

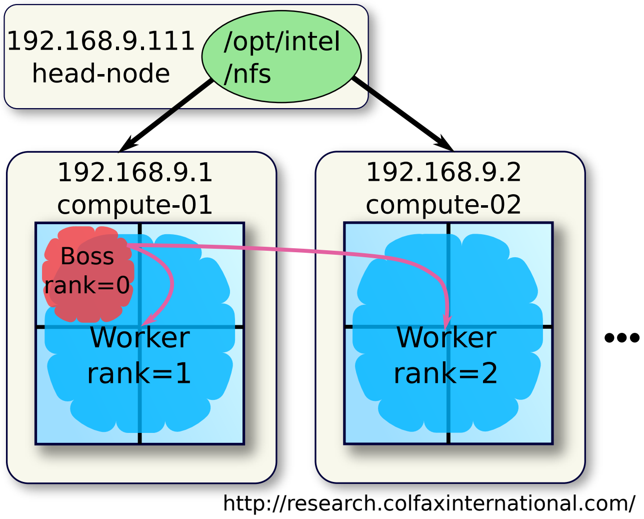

Figure 3: Cluster configuration and MPI run setup for the Asian option pricing application with boss-worker scheduling. Only CPU-based compute nodes are used in this homogeneous cluster.

```
# Intel MPI environment
[colfax@head-node]# source \
> /opt/intel/impi/4.1.1/intel64/bin/mpivars.sh
# Viewing the cluster configuration
[colfax@head-node]# cat /etc/hosts | grep 192
192.168.9.1     compute-01
192.168.9.2     compute-02
[colfax@head-node]# cat /etc/exports
/opt/intel 192.168.9.0/24(rw,no_root_squash)
/nfs 192.168.9.0/24(rw,no_root_squash)

# Compiling the code, sharing with compute nodes
[colfax@head-node]# mpiicc -std=c99 -mkl \
>    -openmp -xAVX -o options options.c
[colfax@head-node]# cp -v options /nfs/options/
'options' -> '/nfs/options/options'

# Starting the MPI job
[colfax@head-node]# cat ./machines-CPUs
192.168.9.1  # Boss process on compute-01
192.168.9.1  # Worker on compute-01
192.168.9.2  # Worker on compute-02
[colfax@head-node]# export I_MPI_PIN=0
[colfax@head-node]# mpirun \
>     -machinefile machines-CPUs \
>     /nfs/options/options
...
```

Listing 3: Launching a homogeneous MPI calculation using a machine file. The first compute node receives two MPI processes: the boss and one worker. All other nodes are assigned only one worker process.

In order to compile the application, we run the Intel MPI wrapper for the Intel C compiler, called `mpiicc`. The compiler options include `-openmp` and `-mkl` to enable runtime loading of the OpenMP and MKL libraries.

The execution of the application is done using the script `mpirun` provided by the Intel MPI library. We had prepared the list of hosts on which the calculation is run in the file `machines-CPUs`. The first line in this file corresponds to the MPI rank 0, which is the boss process. All other lines list the workers.

One important aspect of the execution is that we set the environment variable `I_MPI_PIN=0`. This setting indicates to the Intel MPI library that it should not pin MPI processes on their respective hosts. By default, if several MPI processes are started on one host, the MPI library restricts access of these processes to the CPU resources, effectively partitioning the system to share it between the processes. In our case, we place the boss process and the first worker on the same host, `compute-01`. With pinning, on our two-socket compute nodes, the boss would have access to one socket, and the worker to another. However, this is not optimal, because the boss process is single-threaded, and it will leave its assigned multi-core socket under-utilized. By disabling pinning, we allow the worker to use all of the CPU cores. The boss process is not CPU-intensive, and it does not hurt the overall performance to run it in addition to the worker.

4. Heterogeneous Clustering with Coprocessors

In this section we will demonstrate how the Asian option pricing application developed for a traditional cluster can be executed, without code modification, on a heterogeneous cluster containing Intel Xeon Phi coprocessors. Specifically, we put the coprocessors on the same network as the compute nodes, and then we place additional MPI processes directly on Intel Xeon Phi coprocessors. Our target setup is illustrated in Figure 4 (compare to Figure 3). We assume that the MIC Platform Software Stack (Intel MPSS) [2] has already been installed and configured with default settings.

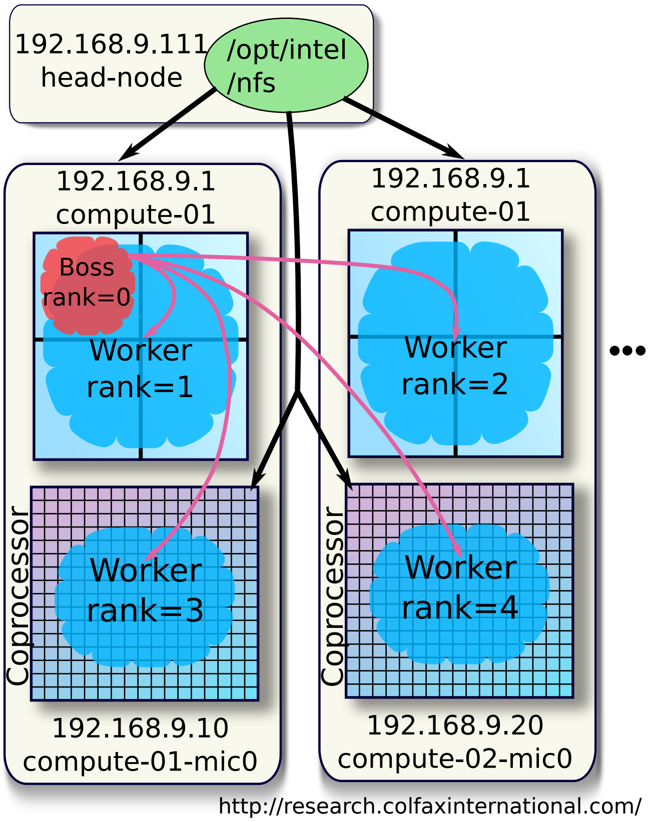

Figure 4: Cluster configuration and MPI run setup for a heterogeneous calculation. The application uses a heterogeneous cluster with CPU-based compute nodes and Intel Xeon Phi coprocessors that act as independent compute nodes.

4.1. SSH Keys for Coprocessors

In order to run MPI processes natively on Intel Xeon Phi coprocessors, the head node of the cluster must be able to SSH into the operating system on running on the coprocessor. Authentication in this process is done using SSH keys instead of passwords, just like with MPI client authentication on regular compute nodes. Unless this has already been done, the procedure shown in Listing 4 must be performed.

Note that we stop MPSS on each compute node, and reset the configuration of coprocessors. This is done for two reasons. First, it automates the copying of the SSH keys from the user's home directory on the host to the authorized keys file on all Intel Xeon Phi coprocessors. Second, stopping the MPSS service is required for the

```
[colfax@head-node]% ssh-keygen
(output omitted)
[colfax@head-node]% cat ~/.ssh/id_rsa.pub >>\
> ~/.ssh/authorized_keys
# The following steps must be repeated
# for all compute nodes in the cluster.
[colfax@head-node]% scp ~/.ssh/id_rsa \
> ~/.ssh/id_rsa.pub ~/.ssh/authorized_keys \
> compute-01:~/
[colfax@head-node]% sudo su
[root@head-node]# ssh compute-01
[root@compute-01]# service mpss stop
Shutting down MPSS stack:    [ OK ]
[root@compute-01]% micctrl --cleanconfig
[root@compute-01]% micctrl --initdefaults
```

Listing 4: Configuring passwordless access to coprocessors in the cluster using SSH keys.

subsequent steps discussed in Sections 4.2 and 4.3.

4.2. Bridged Network Configuration

As discussed in Section 3.2, the boss process running on `compute-01` must be able to communicate to the worker processes on `compute-01` and `compute-02`. With the addition of compute devices `compute-01-mic0` and `compute-02-mic0` (coprocessor), the boss process must also communicate to them. The difficulty here is that the boss process is running on `compute-01`, however, `compute-02-mic0` is located on a remote host.

In order to enable such communication, the MPSS supports a bridged network configuration, in which coprocessors are assigned IP addresses on the same subnet as the compute hosts. This is done by creating a network bridge on each compute node, and configuring the coprocessors to connect to the network via that bridge. What happens when `compute-01` sends a message to `compute-02-mic0`, is

1) the message travels from the network interface controller (NIC) of `compute-01` to the NIC of `compute-02`,
2) then, with the help of the operating system and the coprocessor driver on `compute-02`, the message travels across the PCIe bus to the coprocessor `compute-02-mic0`.

See Figure 2 for an illustration of this path. This setup, however, is completely transparent to the application. That is, `compute-01` is oblivious of the fact that `compute-02-mic0` is not a real host, but an Intel Xeon Phi coprocessor on a remote host.

In order to set up bridging, first, a network bridge must be created on each compute node. In the OS that we are using (CentOS 6.4), this is done by creating a configuration as shown in Listing 5.

```
[root@compute-02]% cat \
> /etc/sysconfig/network-scripts/ifcfg-eth0
DEVICE=eth0
HWADDR="00:1E:67:56:B7:4B"
NM_CONTROLLED="no"
ONBOOT="yes"
UUID="ac3cfdfd-b25f-4493-8bad-3f7b9d51d0d1"
BRIDGE=br0
MTU=9000
[root@compute-02]%
[root@compute-02]% cat \
> /etc/sysconfig/network-scripts/ifcfg-br0
DEVICE=br0
TYPE=Bridge
ONBOOT=yes
DELAY=0
NM_CONTROLLED="no"
MTU=9000
BOOTPROTO=static
IPADDR=192.168.9.2
NETMASK=255.255.255.0
NOZEROCONF=yes
```

Listing 5: A virtual bridge on a compute host is created in the network configuration files.

In this configuration, `ifcfg-br0` is a new file that we created. With this file, we are configuring the host `compute-02` to use the virtual interface `br0` to connect to the network, and self-assign the IP address `192.168.9.2`[4]. The file `ifcfg-eth0` was created during the OS installation, and we modified it by adding the line `BRIDGE=br0` and removing the lines that assign the IP address to this device. This procedure must be repeated on each compute node, either manually, or using the cluster management software.

The second step in creating bridged networking for Intel Xeon Phi coprocessors is shown in Listing 6.

As we can see, the command `micctrl --network ...` has changed the IP addresses

[4] If the cluster has a DHCP server, it is acceptable to connect `br0` using DHCP.

```
[root@compute-02]% micctrl --addbridge=br0 \
> --type=external --ip=192.168.9.1 \
> --netbits=24
[root@compute-02]% micctrl --network \
> --bridge=br0 --ip=192.168.9.20
          mic0: Changing network to static
                bridge br0 at 192.168.9.20
          mic1: Changing network to static
                bridge br0 at 192.168.9.21
[root@compute-02]% cat /etc/hosts | grep 192
192.168.9.1  compute-01
192.168.9.2  compute-02
192.168.9.20 compute-02-mic0 mic0
192.168.9.21 compute-02-mic1 mic1
```

Listing 6: Configuring coprocessors on compute nodes to connect to an external network bridge. This makes coprocessors on remote machines IP-addressable.

of the two coprocessors present in this system, and it was reflected in /etc/hosts. Now that the coprocessors of this machine have IP addresses on the same subnet as the hosts, it is possible to ping, SSH into them and send MPI messages to them from remote machines on this subnet. Again, this procedure has to be repeated on all compute nodes of the cluster, including compute-01.

Further details on network configuration with Intel Xeon Phi coprocessors can be found in [3]

4.3. Network File Sharing with Coprocessors

With bridged network configuration, network file sharing (NFS) can be configured across the cluster with Intel Xeon Phi coprocessors, so that shared mounts also appear on the coprocessors. This is a convenience feature that allows easy application initialization and direct file I/O in MPI processes running on the coprocessors.

In Section 3, we already assumed that the head node is configured as an NFS server, exporting /opt/intel and /nfs to all hosts on the subnet 192.168.9.0/24. Therefore, the only remaining task is mounting these directories from the uOS running on each coprocessor.

In Listing 7 we show how the tool micctrl can be used to create the NFS entries in /etc/fstab on coprocessors that automatically mount when the coprocessor boots, and persist across system restarts. We mount /opt/intel and /nfs from the head node 192.168.9.111, and the mount locations on coprocessors have the same paths as on the head node.

```
[root@compute-01]% micctrl --addnfs=/opt/intel \
> --dir=/opt/intel --server=192.168.9.111
[root@compute-01]% micctrl --addnfs=/nfs \
> --dir=/nfs --server=192.168.9.111
[root@compute-01]% service mpss start
Starting MPSS Stack:    [ OK ]
[root@compute-01]%
```

Listing 7: Setting up two NFS mounts on coprocessors and restarting the MPSS.

The NFS mount setup must be repeated on all compute nodes in the cluster. Note that, because adding NFS was the last step in the configuration of our heterogeneous cluster, it is time to start the MPSS service again, which we do in the last command in Listing 7.

4.4. Compilation and Heterogeneous Execution

With the heterogeneous cluster configured, we can recompile the MPI application for the MIC architecture and launch a coprocessor-assisted heterogeneous calculation. Listing 8 summarizes the procedure.

In order to understand what additional steps were taken this time, compare Listing 8 to Listing 3, where a traditional MPI run is launched:

a) We performed two compilation passes. One, with the flag -xAVX, produces the executable options, which is suitable for CPU architectures with the Intel Advanced Vector Extensions (AVX) instruction set, such as our compute nodes. The second pass, with the flag -mmic, produces a native executable for an Intel Xeon Phi coprocessor. The executable file name is options.MIC.

b) Our machine file now contains the IP addresses of coprocessors, in addition to the IP addresses of compute nodes. This means that MPI will place processes on these coprocessors. We include only one coprocessor per node, even though our system has two coprocessors in each system. This is done in order to enable fair comparison of performance. How-

```
[colfax@head-node]# source /opt/intel/impi/4.1.1/intel64/bin/mpivars.sh # Intel MPI environment
[colfax@head-node]# mpiicc -std=c99 -mkl -openmp -xAVX -o options options.c # Compile for CPU
[colfax@head-node]# mpiicc -std=c99 -mkl -openmp -mmic -o options.MIC options.c # Compile for MIC
[colfax@head-node]# cp -v options options.MIC /nfs/options/ # Copy to NFS-shared location
`options' -> `/nfs/options/options'
`options.MIC' -> `/nfs/options/options.MIC'

# Starting the MPI job
[colfax@head-node]# cat ./machines-HETEROGENEOUS # View the machine file
192.168.9.1  # Boss process on compute-01
192.168.9.1  # Worker on compute-01
192.168.9.2  # Worker on compute-02
192.168.9.10 # Worker on compute-01-mic0
192.168.9.20 # Worker on compute-02-mic0
[colfax@head-node]# export I_MPI_PIN=0  # Disable MPI process pinning
[colfax@head-node]# export I_MPI_MIC=on # Enable MPI for the MIC architecture
[colfax@head-node]# export I_MPI_MIC_POSTFIX=.MIC  # Postfix of the MIC architecture executable
[colfax@head-node]# export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$MIC_LD_LIBRARY_PATH # Enables MKL on MIC
[colfax@head-node]# mpirun -machinefile machines-HETEROGENEOUS /nfs/options/options  # Launch
```

Listing 8: Starting a heterogeneous MPI calculation, the work is distributed across CPUs and coprocessors. Coprocessors appear as independent compute nodes in the MPI machine file.

ever, the second coprocessor in each machine can be loaded trivially, by adding the corresponding IP address to the machine file.

c) We set the environment variable `I_MPI_MIC=on` in order to inform the Intel MPI library that native processes on Intel Xeon Phi coprocessors will be launched.

d) `I_MPI_MIC_POSTFIX=.MIC` is the postfix, which the MIC executable has (remember, we compiled `options` for the CPU architecture and `options.MIC` for the MIC architecture).

e) A trick with `LD_LIBRARY_PATH` enables easy loading of the Intel MKL library on coprocessors from the NFS-shared location `/opt/intel`

Note also that the arguments of `mpirun` are unchanged, with the exception of the machine file name. That completes our setup and launches a heterogeneous (coprocessor-assisted) calculation of Asian option pricing as illustrated in Figure 4.

5. Performance Results

All tests were performed on a cluster consisting of two compute nodes. The nodes are Colfax ProEdge SXP-7450 workstations. These are two-socket machines, with each socket containing an 8-core Intel Xeon E5-2687W V2 processor. The total amount of memory is 128 GB per server in 1600 MHz 16-GB memory modules. In each system, two Intel Xeon Phi QS-3120A active-cooled coprocessors are installed, of which only one per system was used in the benchmarks. The operating system is CentOS 6.4 with the Linux kernel 2.6.32-358. The Intel C Compiler version 13.1.3.192 was used for the tests, and Intel MPSS 2.1.6720-15 was installed. Toshiba MG03ACA100 hard disk drives are used on all machines.

5.1. Coprocessor-Assisted Calculation

Figure 5 shows three bars corresponding to the performance of three calculations:

1) A CPU-only calculation as set up in Listing 3,
2) A coprocessor-only calculation, for which the setup is not shown in the text, but which can be executed by removing the second and third line from the machine file in Listing 8, and
3) A coprocessor-assisted heterogeneous calculation as set up in Listing 8.

The performance is the number of option paths priced per second, with a breakdown of the contributions of each compute device (CPU or coprocessor). This performance metric was computed as

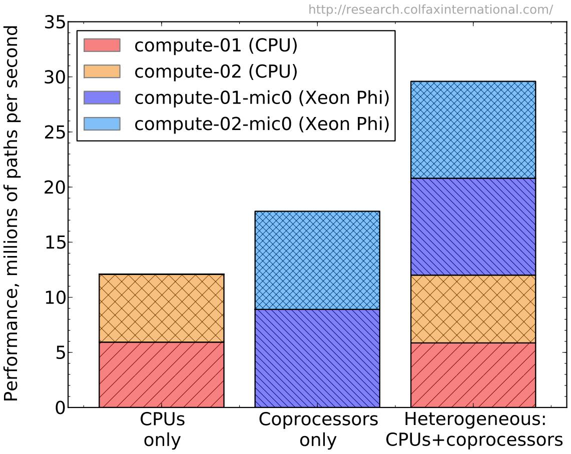

Figure 5: Performance in three configurations: only CPUs, only coprocessors, and both.

$M \times N \times Q/\tau$, where τ is the wall clock time of the calculation in which Q sets of parameters are processed on the cluster, and for each parameter set, M Monte Carlo paths are played out with N points for Asian option averaging. The parameters used for the calculation are $M = 2^{20}$, $N = 365$, $Q = 100$. Other parameters of the option used in the simulation are $S = 15.3$, $T = 1.0$, $\mu = r = .08$, $0.05 < v < 0.5$ and $10 < K < 20$. These parameters do not affect the application performance.

For the parameters that we chose, the frequency of MPI communication is low enough, and the number of work-items Q is large enough so that the load can be almost perfectly balanced across the four compute devices. As one would expect in this case, the net performance achieved CPUs and coprocessor is equal to the performance with CPUs only plus the performance with only coprocessors. See Section 6.2 for a discussion of the scheduling overhead.

5.2. Behind the Scenes: MPI and NFS Speed

In order to put the absolute performance metrics reported above in context, and to estimate the scalability limits of this approach, we performed additional tests of the hybrid cluster performance.

The first test is the MPI latency and bandwidth for communication within the cluster. This metric is important, because for the present problem, it determines the parallel scalability limits of the boss-worker model. In order to measure the MPI performance, we used the Intel MPI benchmark. We executed the “PingPong” test in several setups to measure the communication efficiency between the host `compute-01`, on which the boss process is running, and

1) `compute-01` (i.e., boss and worker are running on the same CPU node),
2) `compute-02` (worker on a remote compute node),
3) coprocessor `compute-01-mic0` (worker on a local MIC node),
4) coprocessor `compute-02-mic0` (worker on a remote MIC node).

The results are shown in Figure 6.

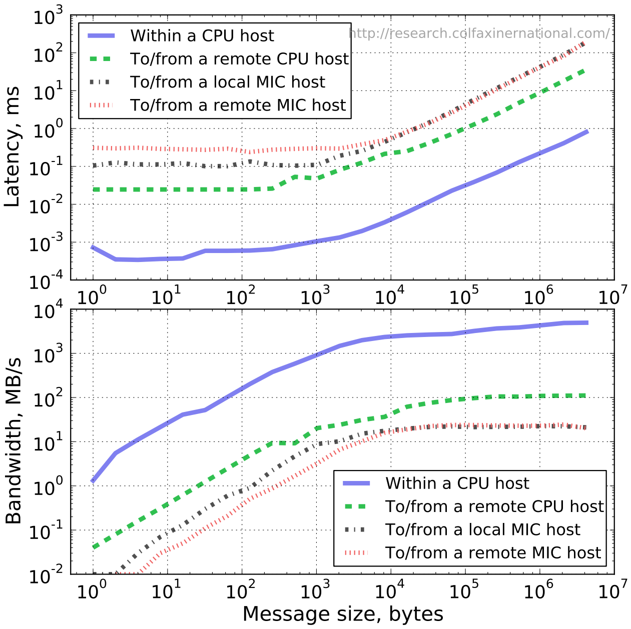

Figure 6: Intel MPI benchmarks in the PingPong mode measure the latency and bandwidth of MPI communication between a CPU-based host (simulating the boss process) and processes on other platforms (simulating the workers).

The second test is the performance of the NFS-exported file system. This metric is important in applications where heterogeneous MPI is used to initialize MPI processes on coprocessors from data files, thus avoiding initialization by the pragma-based offload facility. Additionally, when MPI processes output some data into files (instead of pushing it back to the host using the offload), it is important to know how fast this writing can be done. We measured the read and write speeds of large files

1) on a local hard drive (on the head node),
2) on a remote CPU host `compute-02`, and
3) on a remote MIC host `compute-02-mic0`.

This was done using the Linux tool `dd` to read or write a single file, 1 GB in size. In order to avoid cache effects, we flushed the disk cache before and after the write operation using `sync`. To purge the disk cache before the read operation, we unmounted and remounted the NFS share prior to reading. On the local file system, we purged disk caches by writing "3" into `/proc/sys/vm/drop_caches`. Figure 7 shows the performance results.

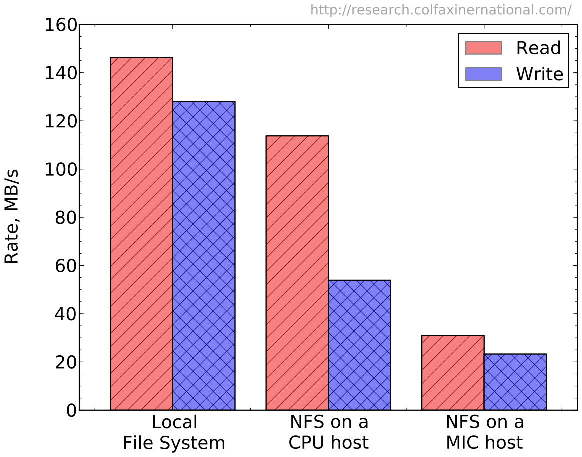

Figure 7: Speeds of reading and writing a 1 GB file into file systems, local and NFS-shared, with compute nodes and with the uOS on coprocessors.

The I/O performance of the local drive on the head node is 145 MB/s for reading and 130 MB/s for writing. On a remote CPU host (`compute-02`), the NFS speed reaches only 110 MB/s for reading and 50 MB/s for writing. This performance is limited by the bandwidth of the 1 Gigabit/s switch used to interconnect the machines. Finally, on a remote MIC host (`compute-02-mic0`), the NFS I/O speed reaches only 30 MB/s for reading and 25 MB/s for writing. It is logical to attribute this limitation to the implementation of the networking stack on the MIC architecture, because the bandwidth of the interconnect and of the PCIe bus are far greater.

6. Discussion

We have demonstrated that some applications designed for traditional CPU-based clusters can be executed on an Intel Xeon Phi coprocessor-accelerated heterogeneous cluster without code modification. Furthermore, significant performance improvement from adding the coprocessors was observed. To conclude this work, we discuss the prerequisites for the success of this approach, and the limitations of heterogeneous clustering method compared to the traditional offload-based acceleration.

6.1. Prerequisites for Improved Performance on Coprocessors

The Intel C/C++ and Fortran compilers are able to take high-level language code written for a general-purpose CPU and compile it into an executable for the many-core architecture of Xeon Phi coprocessors. However, as discussed in our previous publications (e.g., [4], [5], [6] and [7]), the coprocessor will yield a better performance than an Intel Xeon CPU-based system of comparable generation and wattage only if the code is designed according to parallel programming guidelines:

a) The algorithm is taking advantage of multiple cores through the OpenMP or Pthreads framework;
b) Synchronization and false sharing are insignificant, so that the application scales linearly up to 100 or more threads;
c) Data structures and loop implementations permit automatic vectorization by the compiler;
d) No hand-coded vector instructions are used (SSE and AVX instructions are not supported in Intel Xeon Phi coprocessors);

e) Memory traffic is either bandwidth-bound, or has good locality in time and space, so that the caches are used efficiently.

These guidelines are discussed in great detail in [8], and we will not elaborate on them in this publication.

The guidelines shown above are also applicable to HPC applications for multi-core CPUs. Therefore, an important prerequisite for the success of a "recode-nothing" approach is having a highly optimized multi-core CPU implementation in the first place.

6.2. Limitations of Heterogeneous MPI

Even though most MPI applications can be translated to Intel Xeon Phi coprocessors in the way demonstrated here, not every application will be accelerated on a hybrid cluster without optimization or restructuring. The list below summarizes the prerequisites for successful heterogeneous execution.

a) The amount of MPI communication should not be overwhelming. In the configuration presented here, we used a commodity 1 Gigabit Ethernet switch, and MPI messages were passed over the TCP/IP protocol. As Figure 6 shows, message passing from a host to a local coprocessor achieves a bandwidth of only 20 MB/s. This is 300x lower than the practically achievable bandwidth of the traditional offload data traffic, which is currently 6-7 GB/s [8]; it is also 5x slower than the measured bandwidth between CPU-based compute nodes. Therefore, by making remote coprocessors conveniently IP-addressable, the cluster compromises a great deal of communication bandwidth between a CPU and a local coprocessor. Whether this is a limiting factor or not, is determined by the ratio between data movement and computation in any given application.

b) Communication latency in a heterogeneous cluster may be an issue in applications with frequent synchronization. As Figure 6 shows, the greatest communication latency is the message passing from a CPU host to a remote coprocessor. It takes 0.3 ms to ping-pong a short message between these end points, as compared to 0.02 ms for ping-pong between CPU hosts. Furthermore, communication latency becomes important even for embarrassingly parallel applications that could run without any synchronization on traditional, homogeneous clusters. This is because for heterogeneous clusters, load balancing in some form is required. In the boss-worker model, where the boss hands out work items to the workers, communication latency puts a limit on the parallel scalability of the application. Indeed, the maximum number of workers in the boss-worker model must be considerably lower than the ratio of τ (the average compute time of a work-item on a single worker) to the scheduling latency.

$$N_W \ll \frac{\tau}{0.3\,\mathrm{ms}}. \qquad (8)$$

If the application must be scaled beyond the limit set by Equation (8), the programmer must increase the size of a work-item or adjust the size of work-items dynamically to achieve a compromise between the scheduling overhead and load balance. For problems where the compute time for any work item is predictable, it may be possible to avoid dynamic scheduling, and instead, statically schedule a balanced workload by assigning to coprocessors more work proportionally to their relative performance. Section 4.7 in [8] provides examples of such scheduling modes. In complex cases, some sort of collective work scheduling must be employed, such as a hierarchical structure of boss processes or a work-stealing algorithm.

c) File I/O on coprocessors via NFS cannot be too intensive. NFS is a useful convenience feature, enabling coprocessors to initialize from files and output results. However, due to a limited speed of the current NFS implementation for Intel Xeon Phi coprocessors, it cannot be relied upon for high-speed I/O. As Figure 7 shows, reading or writing inside an NFS-shared directory on a coprocessor achieves only $\sim$20% of the available 1 Gigabit/s interconnect bandwidth. If initialization of MPI processes on coprocessors from files becomes a bottleneck of the calculation, the application may need to be restructured to traditional offload-based acceleration.

6.3. Having Your Cake and Eating It Too

As we have seen, the heterogeneous cluster configuration, with Intel Xeon Phi coprocessors acting as independent compute nodes, enables, in some cases, acceleration "out of the box". A significant factor in the convenience of this approach is that it does not require any specialized networking hardware. The toy cluster tested in this work used a common 1 Gigabit/s switch to interconnect the nodes. However, the convenient bridged network configuration comes at the cost of increased communication latency and reduced data transfer bandwidth.

At the same time, for communication-bound workloads, the InfiniBand fabric provides reduced latency, decreased CPU overhead, and advanced technologies, such as Remote Direct Memory Access (RDMA). The Intel TrueScale product line is an InfiniBand fabric, which, in combination with the OpenFabrics Enterprise Distribution (OFED), integrates with the Intel MPSS. TrueScale and may be able to provide improved performance with simplified programming techniques. In future publications on Colfax Research, these advanced HPC technologies will be showcased and examined.

References

[1] Hetetogeneous Clustering with Homogeneous Code (landing page for this paper).
http://research.colfaxinternational.com/post/2013/10/17/Heterogeneous-Clustering.aspx.

[2] Intel Manycore Platform Software Stack (MPSS).
http://software.intel.com/en-us/articles/intel-manycore-platform-software-stack-mpss.

[3] System Administration for the Intel Xeon Phi Coprocessor.
http://software.intel.com/en-us/articles/system-administration-for-the-intel-xeon-phi-coprocessor.

[4] Andrey Vladimirov. Auto-Vectorization with the Intel Compilers: is Your Code Ready for Sandy Bridge and Knights Corner?
http://research.colfaxinternational.com/post/2012/03/12/AVX.aspx.

[5] Andrey Vladimirov and Vadim Karpusenko. Test-driving Intel Xeon Phi coprocessors with a basic N-body simulation.
http://research.colfaxinternational.com/post/2013/01/07/Nbody-Xeon-Phi.aspx.

[6] Vadim Karpusenko and Andrey Vladimirov. How to Write Your Own Blazingly Fast Library of Special Functions for Intel Xeon Phi Coprocessors.
http://research.colfaxinternational.com/post/2013/05/03/Fast-Library-Xeon-Phi.aspx.

[7] Andrey Vladimirov. How to Write Your Own Blazingly Fast Library of Special Functions for Intel Xeon Phi Coprocessors.
http://research.colfaxinternational.com/post/2013/08/12/Trans-7110.aspx.

[8] Colfax International. *Parallel Programming and Optimization with Intel Xeon Phi Coprocessors*. ISBN: 978-0-9885234-1-8. Colfax International, 2013.
http://www.colfax-intl.com/xeonphi/book.html.