CLUSTER-LEVEL TUNING OF A SHALLOW WATER EQUATION SOLVER ON THE INTEL MIC ARCHITECTURE

Andrey Vladimirov[1] *and Cliff Addison*[2]

[1] *Colfax International*
[2] *University of Liverpool*

May 12, 2014

Abstract

The paper demonstrates the optimization of the execution environment of a hybrid OpenMP+MPI computational fluid dynamics code (shallow water equation solver) on a cluster enabled with Intel Xeon Phi coprocessors. The discussion includes:

1. Controlling the number and affinity of OpenMP threads to optimize access to memory bandwidth;
2. Tuning the inter-operation of OpenMP and MPI to partition the problem for better data locality;
3. Ordering the MPI ranks in a way that directs some of the traffic into faster communication channels;
4. Using efficient peer-to-peer communication between Xeon Phi coprocessors based on the InfiniBand fabric.

With tuning, the application has 90% percent efficiency of parallel scaling up to 8 Intel Xeon Phi coprocessors in 2 compute nodes. For larger problems, scalability is even better, because of the greater computation to communication ratio. However, problems of that size do not fit in the memory of one coprocessor.

The performance of the solver on one Intel Xeon Phi coprocessor 7120P exceeds the performance on a dual-socket Intel Xeon E5-2697 v2 CPU by a factor of 1.6x. In a 2-node cluster with 4 coprocessors per compute node, the MIC architecture yields 5.8x more performance than the CPUs.

Only one line of legacy Fortran code had to be changed in order to achieve the reported performance on the MIC architecture (not counting changes to the command-line interface).

The methodology discussed in this paper is directly applicable to other bandwidth-bound stencil algorithms utilizing a hybrid OpenMP+MPI approach.

Table of Contents

1. INTRODUCTION

This paper continues our cycle of publications on the optimization of HPC applications for computing systems enabled with Intel Xeon Phi coprocessors featuring the Many Integrated Core (MIC) architecture[1]. The application studied in this work is a computational fluid dynamics code (CFD), a shallow water equation solver. The source code of the application is freely available at [1]. The code is based on the numerical method developed by Robert Sadourny [3] and initially written by Paul Schwarztrauber of National Center for Atmospheric Research (NCAR). Thereafter, the code was modernized and adapted for hybrid parallelism in the OpenMP and MPI frameworks by Cliff Addison (University of Liverpool). This solver is a good candidate for acceleration on the Intel MIC architecture because it submits to parallelization and operates in the memory bandwidth-bound regime.

In our solver, just like in any MPI application designed for CPU-based clusters, initialization and data traffic were already implemented prior to porting to the Intel MIC architecture. Therefore, the porting procedure can be limited to recompilation of the code [4]. Furthermore, the solver is already optimized for compute nodes based on multi-core CPUs. It is expressed in the Fortran language, with OpenMP and array notation used to express parallelism. As a consequence, the node-level performance of the solver is also very efficient when the code is compiled for and run on an Intel Xeon Phi coprocessor. This means that the optimization process does not need to involve any code modification, and only the execution environment and parameters need to be tuned.

Optimization methods described here can be applied to other bandwidth-bound stencil codes on multidimensional Cartesian grids.

In Section 2 we outline the nature of the calculation and the methodology of work distribution across computing nodes in the cluster. Section 3 discusses the optimization of the MPI environment for executing the code on a single CPU-based node or a single Xeon Phi coprocessor. Cluster-level tuning is discussed in Section 4, and results are presented in Section 5.

[1] See, e.g., [2] for information

2. ENSTROPHY-CONSERVING SHALLOW WATER EQUATION SOLVER

The physical model of shallow water flow assumes a perfect incompressible fluid subject to the gravitational force. This model can be used to describe, for example, the movement of the ocean water on Earth or certain atmospheric phenomena.

The quantities that describe shallow water flow are the pressure $P(x,y)$ and the fluid velocity $\mathbf{V}(x,y)$. The components of vector $\mathbf{V}$ are denoted as $V_x \equiv u$ and $V_y \equiv v$. These quantities can be related to the height $H(x,y)$ of the water above the equilibrium level as $H = P + V^2/2$, where $V^2 \equiv u^2 + v^2$. The evolution of shallow water flow can be expressed with the set of equations (1)–(2):

$$\frac{\partial \mathbf{V}}{\partial t} + \eta \mathbf{N} \times (P\mathbf{V}) + \nabla \left(P + \frac{1}{2} V^2 \right) = 0, \quad (1)$$

$$\frac{\partial P}{\partial t} + \nabla \cdot (P\mathbf{V}) = 0. \quad (2)$$

Here $\eta \equiv [\partial v/\partial x - \partial u/\partial y]/P$ is the potential vorticity, and $\mathbf{N}$ is the unit vector normal to the plain domain S. The system of units is chosen so that the free fall acceleration is reduced to a dimensionless value $g = 1$, and the fluid density $\rho = 1$.

The numerical method for solving these equations proposed by Sadourny [3] conserves enstrophy

$$Z = \frac{1}{2} \int_S \eta^2 P dS,$$

which is a desirable stability property. Enstrophy conservation avoids nonlinear instability that leads to an energy catastrophe in the simulation. The time update step in this method is expressed as

$$\frac{\partial u}{\partial t} - \langle \eta \rangle_x \left\langle \langle V \rangle_x \right\rangle_y + \delta_x H = 0, \quad (3)$$

$$\frac{\partial v}{\partial t} + \langle \eta \rangle_y \left\langle \langle U \rangle_y \right\rangle_x + \delta_y H = 0, \quad (4)$$

$$\frac{\partial P}{\partial t} + \delta_x U + \delta_y V = 0. \quad (5)$$

Here δ_x and δ_y are derivation operators on the staggered simulation grid (see Section 2 in [3]), triangular brack-

ets $\langle\rangle$ are operators of averaging along the direction indicated by their subscript (equivalent to the overline operator in [3]), and $U \equiv \langle P \rangle_x u$ and $V \equiv \langle P \rangle_y v$ are mass fluxes. These operators use only adjacent cells for derivation and averaging, and so the algorithm can be expressed as a 2-dimensional stencil operation.

In the Fortran code expressing equations (3)–(5), quantities u, v, U, V, H and P are defined on a Cartesian grid of $N \times N$ cells, with x-axis being the inner matrix dimension and y-axis being the outer. Periodic boundary conditions are assumed. Parallelism is achieved by distributing the grid columns (y-dimension) across OpenMP threads and exposing the x-dimension to automatic vectorization by the compiler.

In the distributed-memory (MPI) version of the code, the simulation grid is partitioned into H blocks in the x-dimension W in the y-dimension, as shown in Figure 1. Each block is assigned to one of the $R = H \times W$ MPI processes. All blocks contain nearly the same number of cells. MPI proceses exchange boundary cells in order to perform the calculation.

Listing 1 and Figure 1 illustrate the approach taken in the numerical solver to expose parallelism[2]. Listing 2 shows how MPI convenience functions are used to partition the grid into blocks for distribution across the processes of the MPI job.

```
!$OMP PARALLEL DO schedule(static)
DO J = sy,ey
  ! Array notation below is automatically
  ! vectorized by the compiler in the
  ! first index dimension (x-dimension).
  UNEW(sx+1:ex+1,J) = UOLD(sx+1:ex+1,J) + &
    TDTS8*(Z(sx+1:ex+1,J+1) + Z(sx+1:ex+1,J))* &
    (CV(sx+1:ex+1,J+1) + CV(sx:ex,J+1) + &
     CV(sx:ex,J) + CV(sx+1:ex+1,J)) - &
    TDTSDX*(H(sx+1:ex+1,J) - H(sx:ex,J))
  VNEW(sx+1:ex+1,J) = VOLD(sx+1:ex+1,J) + &
    TDTS8*(Z(sx+1:ex+1,J+1) + Z(sx:ex,J+1))* &
    (CU(sx+1:ex+1,J+1) + CU(sx:ex,J+1) + &
     CU(sx:ex,J) + CU(sx+1:ex+1,J)) - &
    TDTSDY*(H(sx:ex,J+1) - H(sx:ex,J))
  PNEW(sx:ex,J) = POLD(sx:ex,J) - &
    TDTSDX*(CU(sx+1:ex+1,J) - CU(sx:ex,J)) - &
    TDTSDY*(CV(sx:ex,J+1) - CV(sx:ex,J))
END DO
!$OMP END PARALLEL DO
```

Listing 1: Time update step in the solver in Fortran with OpenMP and array syntax. Iterations along the y-dimension are distributed across OpenMP threads, and iterations along the x-dimension can be automatically vectorized by the compiler. The length of the inner loop, $ex - sx + 1$, depends on the problem size and on the partitioning of the problem between MPI processes. The corresponding numerical model is expressed by Equations (3)–(5).

3. SINGLE-NODE PERFORMANCE OPTIMIZATION

We run all benchmarks on a cluster with the configuration summarized below:

1) Two compute nodes Colfax ProEdge™ SXP8600p interconnected with Gigabit Ethernet interconnects and with Mellanox MHQH19B-XTR HCAs (QDR, ConnectX-2 VPI), one per node;
2) Each compute node contains a dual-socket Intel E5-2697 v2 processor (12 cores per socket, Ivy Bridge architecture) with 128 GB of DDR3 RAM at 1600 MHz;
3) In each node, four 61-core Intel Xeon Phi 7120P coprocessors are installed;
4) The software configuration includes the CentOS 6.5 Linux operating system with kernel version 2.6.32-431.el6.x86_64, Intel Cluster Studio XE 2013 SP1 and Intel MPSS 3.2.1 with the OFED-1.5.4.1 stack.

For consistency of benchmarks, all power management features are disabled on Intel Xeon Phi coprocessors.

In the following discussion, we use the traditional terminology in the field of computing accelerators. By CPU, processor, host, or host system, we generally

```
dims(1) = 0
dims(2) = 0
call MPI_DIMS_CREATE( numprocs, 2, dims, ierr )
call MPI_CART_CREATE( MPI_COMM_WORLD, 2, dims, &
     periods, .false., comm2d, ierr )
call MPI_CART_GET( comm2d, 2, dims, periods, &
     coords, ierr )
call MPE_DECOMP1D(m, dims(1), coords(1), sx, ex)
call MPE_DECOMP1D(n, dims(2), coords(2), sy, ey)
```

Listing 2: Partitioning the simulation domain between MPI processes using MPI convenience functions.

[2]Code listings in this paper are given in the free format. The original code is expressed in the fixed-column format.

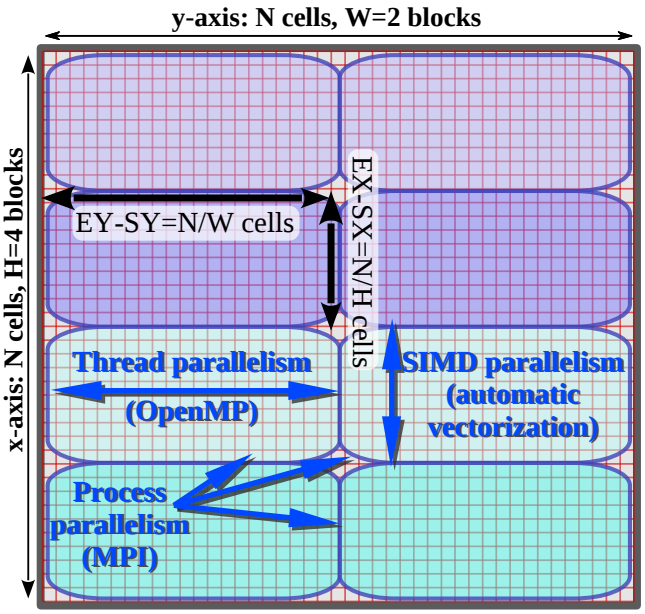

Figure 1: Simulation domain of $N \times N$ cells is partitioned into H blocks x-wise and W blocks y-wise, and each block is assigned to an MPI process. Within each MPI process, OpenMP is used to distribute iterations in the y-dimension between threads, and automatic vectorization is applied in the x-dimension.

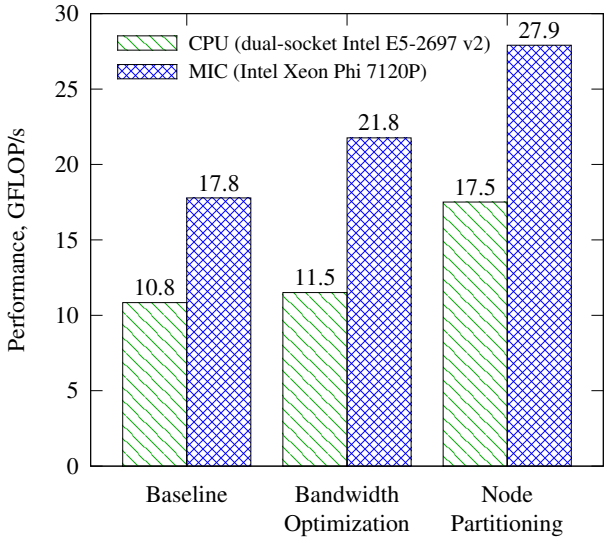

Figure 2: Single-node performance: dual-socket Intel Xeon E5-2697v2 processor and Intel Xeon Phi 7120P coprocessor. See Section 3 for discussion.

mean the Intel Xeon CPU-based computing system. By coprocessor, device or accelerator we mean the Intel Xeon Phi coprocessor. We will also use the term "compute device" in contexts where the term applies to both the coprocessor and the CPU.

3.1. Baseline

Without any platform-specific configuration of the execution environment, the code can be run on the CPUs of the cluster compute nodes, as well as on the Intel Xeon Phi coprocessors. We use `mpirun` to start the calculation with a single MPI process first on the dual-socket CPU, then on one of the Xeon Phi coprocessors, as shown in Listing 3. By default, the hybrid OpenMP/MPI code spawns as many OpenMP threads as there are logical cores on the respective device: 48 threads on the CPU and 244 threads on the coprocessor.

The attained performance is measured internally in the code by dividing the number of arithmetic operations involved in the calculation by the elapsed wall clock time. The result is reported in terms of GFLOP/s, which is a metric specific to this particular solver. For the baseline test, the CPU attained 10.8 GFLOP/s and the coprocessor 17.8 GFLOP/s. This corresponds to the first set of bars in Figure 2.

3.2. Optimizing MPI Process Pinning for Memory Bandwidth

Most modern Intel processor architectures support hyper-threading, which is hardware support for operating more than one software thread per physical core. While hyper-threading improves the performance of applications bound by memory access latency, it is counter-productive for bandwidth-bound workloads that stream contiguous data from memory to cores.

In most cases (see, e.g., [5]), a bandwidth-bound application performs optimally on the Intel architecture when both of the following conditions are met:

1) one software thread is used per core, and
2) thread migration from core to core is forbidden at the operating system level.

Therefore, the first step towards improving the performance of our shallow water equation solver is limiting the number of OpenMP threads on the coprocessor,

```
cfx@c001-n001$ mpirun -host c001-n001 \
> -np 1 /home/cfx/shallow/ncar-solver1 10000
 ...
 WALL CLOCK TIME FOR JOB =   29.97804 seconds
 EXPECTED GFLOPS RATE    =    10.84127

cfx@c001-n001$ export I_MPI_MIC=1
cfx@c001-n001$ mpirun -host c001-n001-mic0 \
> -np 1 /home/cfx/shallow/ncar-solver1.MIC 10000
 ...
 WALL CLOCK TIME FOR JOB =   18.28000 seconds
 EXPECTED GFLOPS RATE    =    17.77900
```

Listing 3: Baseline of code performance: execution on a single node (CPU and MIC) without tuning.

and enforcing the pinning of each thread to a respective physical core. This is done by passing the environment variable `OMP_NUM_THREADS=24` to the MPI process running on the host, or `OMP_NUM_THREADS=61` to the MPI process on the coprocessor, as shown in Listing 4.

```
cfx@c001-n001$ mpirun -host c001-n001 \
> -np 1 -env "OMP_NUM_THREADS=24" \
> /home/cfx/shallow/ncar-solver1 10000
 ...
 WALL CLOCK TIME FOR JOB =   28.23723 seconds
 EXPECTED GFLOPS RATE    =    11.50963

cfx@c001-n001$ export I_MPI_MIC=1
cfx@c001-n001$ mpirun -host c001-n001-mic0 \
> -np 1  -env "OMP_NUM_THREADS=61" \
> /home/cfx/shallow/ncar-solver1.MIC 10000
 ...
 WALL CLOCK TIME FOR JOB =   14.93185 seconds
 EXPECTED GFLOPS RATE    =    21.76555
```

Listing 4: Memory bandwidth is optimized by setting the number of OpenMP threads to the number of physical cores. Affinity of threads and processes to specific cores is set by the Intel MPI library automatically.

Bandwidth optimization by restricting the number of threads improves performance on the CPU by 6%, and on the coprocessor by 22% compared to the baseline. This is shown by the second set of bars in Figure 2.

Note that without MPI, the optimization of OpenMP environment for bandwidth-bound applications requires setting `OMP_NUM_THREADS` to the number of physical cores, and then setting `KMP_AFFINITY=scatter` to prevent thread migration across cores. With MPI, the latter step is not necessary, as the pinning functionality of Intel MPI takes care of binding threads to cores. Pinning decisions made by Intel MPI can be diagnosed by setting `I_MPI_DEBUG=4` prior to running a job. Environment variables for pinning control are described in the Intel MPI Library Reference Manual [6].

3.3. Node Partitioning

Further improvement of the shallow water equation solver may come from using multiple MPI processes per device with proportionally fewer OpenMP threads per process. This tweak in the configuration improves performance for two reasons.

1) On the host system, the two CPU sockets form a Non-Uniform Memory Access (NUMA) system, in which each CPU socket can access memory attached to it faster than memory attached to the other CPU. This leads to performance penalties when one MPI process with multiple threads operates on a large data set, and thread affinity to data is not set. On the other hand, with two (or, in general, with an even number of) MPI processes on a dual-socket system, pinning will ensure that each process addresses only its local memory.
2) On both the CPU and the coprocessor, partitioning the dataset leads to better cache traffic in the stencil operation expressed by the code in Listing 1. Indeed, the inner loop expressed by array notation has `ex-sx+1` iterations. This value is equal to the grid size N for only one MPI process, but for a job with several MPI processes distributed across the simulation box, the value of `ex-sx+1` will be smaller. Smaller length of the inner loop improves data reuse. Indeed, arrays `Z` and `H` used in the calculation of `UNEW` are subsequently used to compute `VNEW`; and arrays `CU` and `CV` re-used in the calculation of `PNEW`. Furthermore, because the stencil operation uses adjacent `y`-values, some arrays are re-used in subsequent `J`-iterations. For a small enough value of `ex-sx+1`, the re-used arrays may still be in the processor's cache when they are accessed a second time. Thus, reducing `ex-sx+1` by increasing the number of MPI processes per compute device improves cache utilization.

The phenomenon described above is often exploited in shared-memory applications (e.g., [7]) through an optimization known as loop tiling, or loop blocking. Loop tiling improves performance by increasing the temporal locality of data access. Our solver would benefit from loop tiling, however, it is not implemented. At the same time, a similar pattern of data re-use occurs naturally when OpenMP is used in tandem with MPI to partition the simulation box.

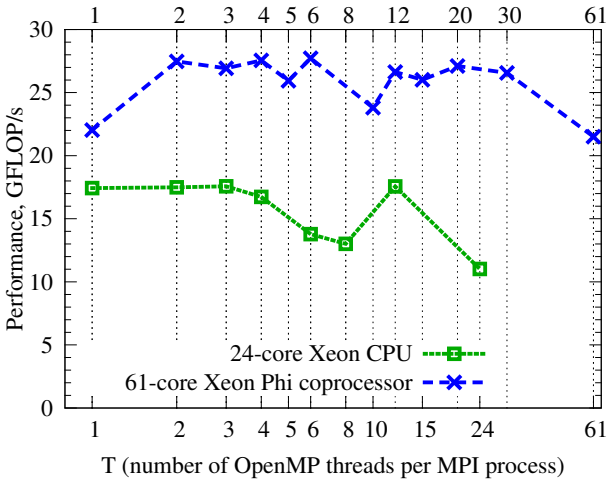

Figure 3: Impact of node partitioning between several MPI processes on performance. Data obtained on the host and on the coprocessor. In each run, the number of MPI processes is chosen as $24/T$ on the CPU, and as $61/T$ on the coprocessor (with rounding).

Figure 3 demonstrates the dependence of the code performance on the number of OpenMP threads per MPI process. On the host CPU, using fewer threads and many processes leads to better performance than using only one 24-threaded process. On the coprocessor, exploratory trials are needed to determine the optimal number of threads. The case of $T = 2$ in Figure 3 corresponds to the third set of bars in Figure 2.

4. CLUSTER-LEVEL OPTIMIZATION CAVEATS

At this point, the performance of the solver on a single node is tuned, and one Xeon Phi coprocessor performs 1.6x faster than the dual-socket Xeon host. Considering that each of our compute nodes contains 4 coprocessors, we can expect to run up 6.4x faster on the MIC accelerators than on the CPUs. Because in this case, the performance of Xeon Phi coprocessors dwarfs the performance of host CPUs, from this point on, we will focus only on optimization for coprocessors.

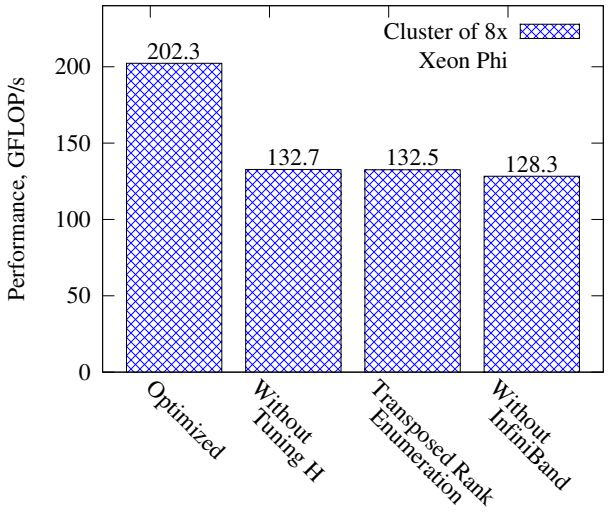

Figure 4: Tuned performance and degraded performance of eight Xeon Phi coprocessors in two compute nodes of the cluster. See Section 4 for discussion.

4.1. DOMAIN DECOMPOSITION TUNING

Scaling across multiple compute nodes with MPI can be done by modifying the arguments of `mpirun` and using a machine file as illustrated in Listing 5. There we configure MPI to run the code on 8 coprocessors installed in two compute nodes: `c001-n001` and `c001-n002`. We use 4 threads per process and 15 processes per Xeon Phi (a total of $15 \times 8 = 120$ MPI ranks). This is one of the optimal numbers of threads as shown in Figure 3. We could not use 2 threads per process, because the job would exceed the maximum number of ranks supported in our cluster.

As Listing 5 reports, we obtain a performance of 132.7 GFLOP/s from 8 coprocessors, which is 4.8x greater than the performance of one coprocessor. This is not a satisfactory scalability metric, and more investigation is required.

```
cfx@c001-n001$ cat allmics.txt
c001-n001-mic0:15
c001-n001-mic1:15
c001-n001-mic2:15
c001-n001-mic3:15
c001-n002-mic0:15
c001-n002-mic1:15
c001-n002-mic2:15
c001-n002-mic3:15
cfx@c001-n001$ export I_MPI_MIC=1
cfx@c001-n001$ mpirun  \
> -machine allmics.txt -env "OMP_NUM_THREADS=4"\
> /home/cfx/shallow/ncar-solver1.MIC 10000
 ...
 WALL CLOCK TIME FOR JOB =     2.44866 seconds
 EXPECTED GFLOPS RATE    =   132.72564
```

Listing 5: Running the solver on a cluster of 2 compute nodes with 4 coprocessors each, 6 threads per process and 10 processes per coprocessor.

Going back to Figure 1, we recognize that with a greater number of MPI ranks, the simulation grid is partitioned into smaller blocks. This drives the calculation away from the optimal value of the x-loop length `ex-sx+1` that was achieved by tuning the number of threads T on one compute device (see Section 3.3). Consequently, one needs to tune the number of threads per MPI rank, T, for the multi-node run, rather than rely on the tuned value of T for a single-node run.

Furthermore, one can infer from Listing 2 that the code allows MPI to choose the decomposition of the simulation domain into blocks both in the x-dimension (parameter H) and in the y-dimension (parameter W). Considering how important the value of `ex-sx+1` is for performance on Xeon Phi coprocessors (see Figure 3), we should consider tuning the dimensions of the blocks H and W. This can be done by changing the call to `MPI_DIMS` as shown in Listing 6.

Of course, the total number of blocks $W \times H$ must be equal to the number of MPI ranks in the simulation, R (otherwise, the call to `MPI_DIMS` will fail). Therefore, we should only probe values of H that are divisors of R. The value of R is computed as $R = (60/T) * M$, where T is the number of threads per rank, and M is the number of coprocessors in the calculation.

A thorough scan of parameter space (T, H) allows to tune our solver to the cluster size on which it is run. Figure 5 shows the performance as a function of these two parameters. For $M = 8$ coprocessors, we restricted the search to values of $T \in 1, 2, 3, 4, 5, 6, 10, 12, 15, 20, 30, 60$ and $H \in 1, 2, 4, 8$.

```
1  ! Value of H taken from command-line arguments
2  dims(1) = H
3  dims(2) = 0
4  call MPI_DIMS_CREATE( numprocs, 2, dims, ierr )
5  ! ...
```

Listing 6: Adjusting the decomposition of simulation grid into blocks by setting a value of H (number of blocks in the x-dimension), instead of allowing MPI to choose a value. MPI will choose the second decomposition parameter, W, so that $H \times W = R$ (R is the total number of MPI processes). This is the only code change that was necessary for optimization on the MIC architecture. Compare to Listing 2.

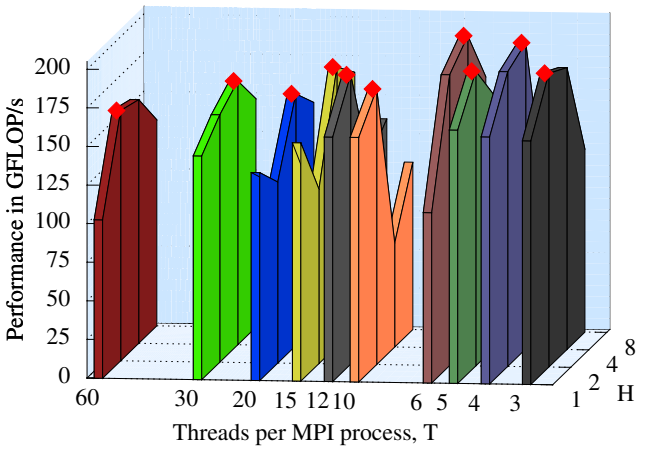

Figure 5: Two parameters must be empirically determined for each problem size N and cluster size M: (i) T, the number of threads per process, and (ii) H, the number of blocks in the x-dimension for grid partitioning. The best-performing pair of (T, H) can then be used in production runs. Shown are results of calibration for $N = 10000$, $M = 8$. Red markers indicate the best values of H for a given T.

The best combination of the tuning parameters for problem size $N = 10000$ on $M = 8$ coprocessors is $T = 6$, $H = 4$ (see Table 1). Compared to similarly tuned performance of the solver on one coprocessor, this is a 7.2x speedup. This performance is reflected in the first bar of Figure 4, while the performance before tuning T and H is the second bar in this Figure.

```
cfxuser@c001-n001$ export I_MPI_DEBUG=3
cfxuser@c001-n001$ cat mics1.txt
mic0:3
mic1:3
[cfxuser@c001-n001$ mpirun \
> -machine mics1.txt -env "OMP_NUM_THREADS=20" \
> /home/cfx/shallow/ncar-solver1.MIC 10000
...
[0] MPI startup(): Rank  Pid     Node name
[0] MPI startup(): 0     20647   c001-n001-mic0
[0] MPI startup(): 1     20648   c001-n001-mic0
[0] MPI startup(): 2     20649   c001-n001-mic0
[0] MPI startup(): 3     15250   c001-n001-mic1
[0] MPI startup(): 4     15251   c001-n001-mic1
[0] MPI startup(): 5     15252   c001-n001-mic1
...
```

Listing 7: Specifying the number of processes per node in the machine file assignes a contiguous chunk of ranks to each compute device.

```
cfxuser@c001-n001$ export I_MPI_DEBUG=3
cfxuser@c001-n001$ cat mics2.txt
mic0
mic1
cfxuser@c001-n001$ mpirun -np 6 \
> -machine mics1.txt -env "OMP_NUM_THREADS=20" \
> /home/cfx/shallow/ncar-solver1.MIC 10000
...
[0] MPI startup(): Rank  Pid     Node name
[0] MPI startup(): 0     20809   c001-n001-mic0
[0] MPI startup(): 1     15413   c001-n001-mic1
[0] MPI startup(): 2     20810   c001-n001-mic0
[0] MPI startup(): 3     15414   c001-n001-mic1
[0] MPI startup(): 4     20811   c001-n001-mic0
[0] MPI startup(): 5     15415   c001-n001-mic1
...
```

Listing 8: Specifying the total number of MPI processes using the argument `-np`, and listing compute nodesin the machine file assigns nodes to ranks in the round-robin order.

4.2. MPI Rank Numbering

Even though at this point, we have provided an optimized solution to scaling the shallow water equation solver, we will demonstrate some caveats on cluster-level tuning, based our experience in this project.

In Listing 5 we used a machine file in which each line has format `host:ppn`, where `host` is the hostname of a coprocessor, and `ppn` is the number of MPI processes on that compute device. In this case, MPI rank numbers are assigned to each coprocessor in contiguous chunks. In Listing 7 we illustrate rank numbering for 6 MPI processes on 2 coprocessors.

At the same time, it is possible to start a calculation in a different way, as shown in Listing 8. In that case, we list one compute device per line in the machine file, and add the argument `-np=R` to `mpirun`, where `R` is the number of MPI ranks equal to 6 in this case. As apparent from the listing, now compute devices are assigned to ranks in the round-robin order.

Figure 6 illustrates the rank assignment pattern in these two cases. In our solver, MPI communication is set up in such a way that each process communicates only with the processes that operate on adjacent grid blocks (left/right and top/bottom neighbors). Consequently, when the left and right neighbors are local (i.e., on the same coprocessor) rather than remote (i.e., on a different device), MPI communication will take less time. That is because intra-coprocessor communication will be performed using the shared-memory copy protocol `shm`, relieving the strain on the PCIe and InfiniBand subsystems that carry inter-coprocessor traffic with the protocol `dapl` (see [8] for details). Thus, the configuration on the left-hand side of Figure 6 (with host file in format `host:ppn`) is better optimized than the configuration on the right-hand side (with argument `-np=R`).

Indeed, a benchmark shows that running our shallow water equation solver on a grid of processors transposed in this way yields only 132.5 GFLOP/s on eight compute devices (third bar in Figure 4), as opposed to 202.3 GFLOP/s in the optimized case.

4.3. Role of Communication Fabric

Our compute cluster uses InfiniBand interconnects for MPI communication. InfiniBand drivers are integrated with Intel MPSS in order to enable Coprocessor Communication Link (CCL). This functionality virtualizes an InfiniBand HCA on each coprocessor, which enables peer-to-peer messaging between Intel Xeon Phi coprocessors in different compute nodes, and also improves comunication between coprocessors within the same host. See [8] for more details.

Without InfiniBand adapters, and without the specialized OFED stack, the cluster would be using the TCP protocol over Gigabit Ethernet for communication. We can emulate this situation by setting the environ-

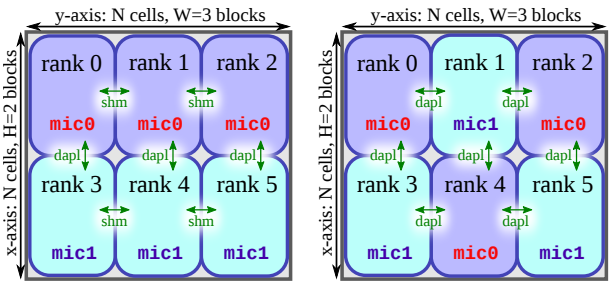

Figure 6: Rank assignment in a grid of $R = W \times H$ processes. *Left:* optimized rank enumeration with the `:ppn` suffix in the machine file (Listing 7). Most left and right neighbors are local (on the same coprocessor) and communicate via the shared-memory copy protocol `shm`. *Right:* sub-optimal rank enumeration with the `-np=R` argument of `mpirun` (Listing 8). All left/right and top/bottom neighbors are remote, communicating via the InfiniBand protocol `dapl`.

ment variable `I_MPI_FABRICS=tcp` for the calculation, which makes the Intel MPI library use the TCP protocol over the Gigabit Ethernet fabric.

The fourth bar in Figure 4 shows the performance of the tuned calculation on 8 coprocessors in 2 compute nodes with the TCP fabric. It achieves 128.3 GFLOP/s, which is only 63% of the performance with InfiniBand.

5. PARALLEL SCALABILITY

Scaling the shallow water equation solver across multiple compute nodes requires that for each cluster size M and problem size N, the tuning parameters (T, H) are empirically determined in a calibration run. T is the number of threads per MPI process, and H is the number of blocks in the x-dimension for partitioning the simulation domain between MPI processes. T must be a divisor of 60 (otherwise, several Xeon Phi cores may go unused), and H must be a divisor of the number of processors $R = (60/T) * M$. We have also determined that values of H greater than M are inefficient, due to the nature of the communication pattern discussed in Section 4.2.

Considering these pruning rules, there are usually several tens of pairs (T, H) that must be tested. Calibration runs take several seconds each, and therefore the process of tuning can be realistically completed on a time scale of several minutes.

Table 1 shows the results obtained for different problem sizes: $N \in 5000, 7000, 10000, 14000$. For $N = 10000$, the working dataset is around 9 GB, so this is the biggest problem in the studied set that fits into the 16 GB memory of a 7120P Xeon Phi coprocessor. Figure 7 illustrates the best performance as a function of the number of coprocessors.

Size N	MICs	Threads	H	GFLOP/s	Speedup
5000	1	6	2	27.1	1.00
5000	2	6	1	50.2	1.85
5000	4	6	1	92.0	3.40
5000	8	6	2	157.8	5.82
7000	1	15	2	27.0	1.00
7000	2	10	3	51.8	1.92
7000	4	12	4	94.7	3.51
7000	8	6	2	185.1	6.85
10000	1	2	3	28.1	1.00
10000	2	2	3	55.0	1.96
10000	4	3	4	107.0	3.81
10000	8	6	4	202.3	7.20
14000	1	n/a	n/a	n/a	n/a
14000	2	3	5	55.9	2.00*
14000	4	6	5	107.4	3.84*
14000	8	4	4	212.4	7.60*

Table 1: Optimized performance and values tuning parameters T (threads per MPI process) and H (grid blocks in x-dimension) for problem sizes $N \in 5000, 7000, 10000, 14000$ on $M = 1, 2, 4$ and 8 Intel Xeon Phi coprocessors.
* For $N = 14000$, the problem does not fit in memory of a single coprocessor, and reported speedup is normalized so that the performance for $M = 2$ has a speedup of 2.0.

We have found that the speedup approaches linear scaling law for larger problems, which is a common behavior in massively parallel systems. For our specific application, this is explained by the fact that the amount of calculation scales with the problem size as $O(N^2)$, while the amount of communication scales as $O(N)$. At the same time, problems that are large enough to demonstrate perfect scalability on 8 coprocessors do not fit into the memory of one coprocessor.

With calculations running only on CPUs, the performance is not as sensitive to the values tuning parameters. Scalability is easy to achieve with CPUs, especially because in our two-node cluster, we only have to scale across two CPU architecture devices as opposed to eight MIC accelerators.

The tuned performance results on both platforms are shown in Figure 8. In this plot, the second set of bars "Two Nodes/Eight MICs" shows the performance achieved within the same rack space (two nodes) with only CPUs and only MIC accelerators.

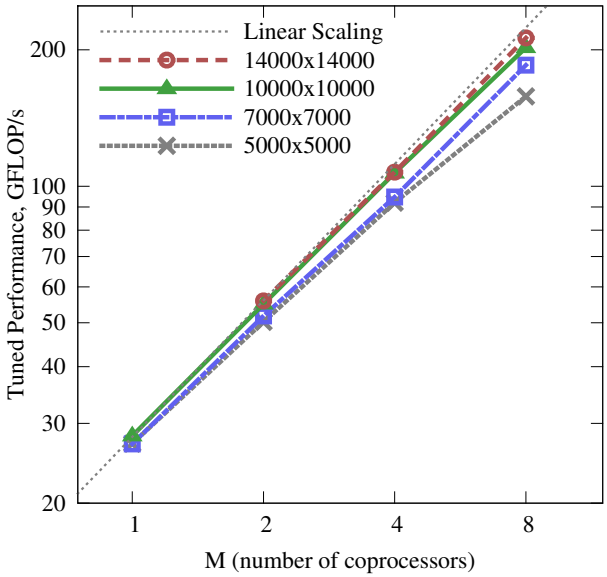

Figure 7: Parallel scalability with tuning of the parameter set (T, H) (see Table 1) of the shallow water equation solver on a cluster of Intel Xeon Phi coprocessors for different problem sizes.

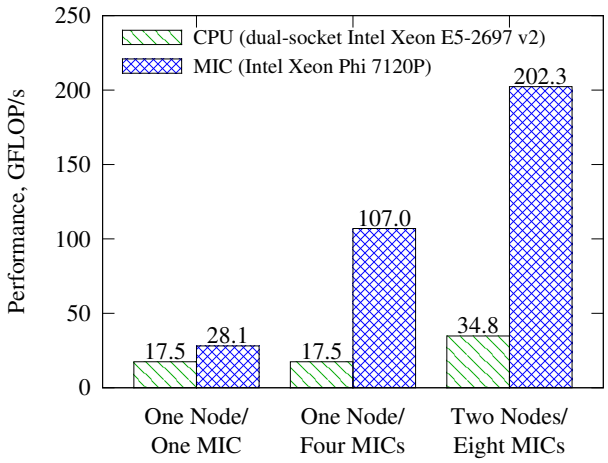

Figure 8: Scalability across multiple coprocessors and multiple CPUs. In this plot, the second and third set of bars show the performance achieved within the same rack space with only CPUs and only MIC accelerators (one node in the second set of bars and two nodes in the third set).

6. CONCLUSIONS

We analyzed the performance of a shallow water equation solver on a MIC-enabled computing cluster. The numerical method is a 2-dimensional stencil code operating in the memory bandwidth-bound regime.

We demonstrated how tuning the number of threads per process T improves performance by increasing the re-use of cached data. A second tuning parameter is required for optimal performance with Intel Xeon Phi coprocessor. This parameter, H, tweaks the size of the inner loops by changing the number of partitions of the simulation domain.

In the parameter calibration process, it was apparent that Xeon Phi coprocessors are less forgiving of sub-optimal parameters than Xeon CPUs. However, after tuning, we achieved a 7.2x performance increase with eight coprocessors compared to one (202.3 GFLOP/s on eight Xeon Phi 7120P). On two compute nodes with dual-socket E5-2697v2 CPUs we achieved 34.8 GFLOP/s. This amounts to a $202.3/34.8 = 5.8$x speedup from the MIC architecture over using the CPUs of two compute nodes alone.

Our optimizations for the MIC architecture required a modification of only one line of code (not counting modifications of the code interface), which controls the partitioning of the simulation domain between MPI ranks (see Listing 6). The original code in Fortran, optimized for CPU-based clusters, performs well as long as the MPI environment was optimized.

It is important to note that the code was not specifically tuned for Intel Xeon Phi coprocessors prior to this work. In fact, the last updates to the code relevant to performance date to the year 2003, when Intel MIC architecture was not in existence. This confirms once again the notions that

i) an application that efficiently uses multi-core CPUs is likely to also perform well on Intel Xeon Phi coprocessors without code modification,

ii) Intel MIC architecture is a viable platform for acceleration and modernization of time-tested legacy applications.

REFERENCES

[1] Landing page for this paper "Cluster-Level Tuning...".
http://research.colfaxinternational.com/post/2014/05/12/Shallow-Water.aspx.

[2] Primer on Computing with Intel Xeon Phi Coprocessors. Slides from a presentation, with links to additional resources.
http://research.colfaxinternational.com/post/2014/03/06/Geant4-Tutorial.aspx.

[3] Robert Sadourny. The Dynamics of Finite-Difference Models of Shallow-Water Equations. *Journal of Atmospheric Sciences*, 32:680–689, 1974.

[4] 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.

[5] Andrey Vladimirov. Terabyte RAM Servers: Memory Bandwidth Benchmark and How to Boost RAM Bandwidth by 20% with a Single Command .
http://research.colfaxinternational.com/post/2012/01/04/Terabyte-RAM-Servers-Memory-Bandwidth-Benchmark.aspx.

[6] Intel MPI Library Reference Manual for Linux* OS.
http://software.intel.com/sites/products/documentation/hpc/ics/impi/41/lin/Reference_Manual/index.htm.

[7] Andrey Vladimirov. Multithreaded Transposition of Square Matrices with Common Code for Intel Xeon Processors and Intel Xeon Phi Coprocessors.
http://research.colfaxinternational.com/post/2013/08/12/Trans-7110.aspx.

[8] Vadim Karpusenko and Andrey Vladimirov. Configuration and Benchmarks of Peer-to-Peer Communication over Gigabit Ethernet and InfiniBand in a Cluster with Intel Xeon Phi Coprocessors.
http://research.colfaxinternational.com/post/2014/03/11/InfiniBand-for-MIC.aspx.