Performance Comparison of a Two-Dimensional Elliptic Test Problem on Intel Xeon Phis

REU Site: Interdisciplinary Program in High Performance Computing

Ishmail A. Jabbie[1], George Owen[2], Benjamin Whiteley[3],
Graduate assistant: Jonathan S. Graf[1],
Faculty mentor: Matthias K. Gobbert[1], and
Client: Samuel Khuvis[4]

[1]Department of Mathematics and Statistics, UMBC,
[2]Department of Mathematics, Louisiana State University,
[3]Department of Engineering & Aviation Sciences, University of Maryland, Eastern Shore,
[4]ParaTools, Inc.

Technical Report HPCF–2016–16, hpcf.umbc.edu > Publications

Abstract

The Intel Xeon Phi is a many-core processor with a theoretical peak performance of approximately 1 TFLOP/s in double precision. This project contrasts the performance of the second-generation Intel Xeon Phi, code-named Knights Landing (KNL), to the first-generation Intel Xeon Phi, code-named Knights Corner (KNC), as well as to a node with two CPUs as baseline reference. The benchmark code solves the classical elliptic test problem of the two-dimensional Poisson equation with homogeneous Dirichlet boundary that is prototypical for the computational kernel in many numerical methods for partial differential equations. The results show that the KNL can perform approximately four times faster than the KNC in native mode or two CPUs, provided the problem fits into the 16 GB of on-chip MCDRAM memory of the KNL.

1 Introduction

Knights Landing (KNL) is the code name for the second-generation Intel Xeon Phi many-core processor, that was announced in June 2014 [8] and began shipping in July 2016. The change in hardware represents a significant improvement over the first-generation of Phi, giving the KNL the potential to be even more effective for memory-bound problems. This is Intel's response to the trend in parallel computing using more cores.

The most important improvement of the second-generation KNL Phi is the 16 GB MCDRAM memory on board the chip. The Phi can also access the node's DDR4 memory whose size is system dependent, but MCDRAM is directly on the chip and is approximately 5x faster than DDR4. The MCDRAM is also nearly 50% faster than the GDDR5 memory, which is the memory on board the first-generation Knights Corner (KNC) chip. The computational cores in the KNL are connected by a 2D mesh structure that allows for significantly more bandwidth since there are more channels for cores to communicate than the bi-directional ring bus on the KNC. The KNL can have up to a maximum of 72 cores, while KNC has up to 61 cores; we have access to an early KNL model with 68 cores. Another dramatic change is that the KNL is capable of serving as primary or sole processor in a node, while the KNC is only a co-processor.

We compare performance of three hardwares, namely KNL, KNC, and CPUs. The KNC model contains 61 cores in a bi-directional ring bus structure with 8 GB of GDDR5 memory on the chip. We use native mode on this Phi for full comparison, which allows access to the GDDR5 memory on the chip only, and remark on the symmetric and offload modes. For reference, we also tested on a compute node with two 8-core CPUs, giving it 16 cores total. Section 2 specifies the hardware in detail.

Table 1.1: Comparison of observed wall clock times in units of MM:SS for $N \times N = 8{,}192 \times 8{,}192$ mesh on CPU node with two 8-core CPUs, KNC with 61 cores, and KNL with 68 cores, accessing the available memory, and a choice of `KMP_Affinity=scatter` or `compact`.

(a) CPU node with DDR4 with total of 16 threads					
MPI proc	1	2	4	8	16
Threads per proc	16	8	4	2	1
DDR4–scatter	36:18	21:57	21:58	22:02	21:48

(b) KNC with GDDR5 with total of 240 threads										
MPI proc	1	2	4	8	15	16	30	60	120	240
Threads per proc	240	120	60	30	16	15	8	4	2	1
GDDR5–scatter	28:24	28:20	27:51	23:08	23:06	23:00	22:24	22:45	22:43	25:37

(c) KNL with DDR4 and MCDRAM with total of 272 threads										
MPI proc	1	2	4	8	16	17	34	68	136	272
Threads per proc	272	136	68	34	17	16	8	4	2	1
DDR4–scatter	26:02	25:07	24:38	24:25	24:24	36:29	37:40	37:54	39:06	41:00
MCDRAM–scatter	05:49	05:43	05:39	05:35	05:36	08:22	08:49	08:41	08:37	08:57
MCDRAM–compact	05:27	05:14	05:11	05:11	05:13	07:44	08:13	08:02	08:08	09:33

As test problem, we use the Poisson equation with homogeneous Dirichlet boundary conditions

$$\begin{aligned} -\Delta u &= f \quad \text{in } \Omega, \\ u &= 0 \quad \text{on } \partial\Omega, \end{aligned} \tag{1.1}$$

on the domain $\Omega = (0,1)\times(0,1) \subset \mathbb{R}^2$. The equation is discretized by the finite difference method on a $N \times N$ mesh and the resulting system of N^2 linear equations solved by the conjugate gradient method. The numerical method is parallelized in C using hybrid MPI and OpenMP code. Section 3 shows the discretized equations and provides implementation details. We use a self-written special-purpose code, but (1.1) is a very well-known standard example (e.g., [1, Section 6.3], [3, Subsection 9.1.1], [9, Chapter 12], and [13, Section 8.1]) and researchers should be easily able to recreate the results.

The performance studies in Section 4 use all available computational cores in each hardware, since that provides best total performance. With 4 threads per core supported on the Phis, this gives the KNL a total of 272 threads, KNC 240 (with 1 core reserved for the operating system), and the compute node with two CPUs 16 total. We test combinations of MPI processes and OpenMP threads. The KNL studies also compare the two types of memory available, MCDRAM and DDR4, and examine two settings for an environment variable, `KMP_AFFINITY=scatter` and `KMP_AFFINITY=compact`. Finally, the KNL results take advantage of the larger on-chip memory by solving the problem for a larger resolution than possible on the KNC.

Table 1.1 summarizes the comparison of the results of observed wall clock time (in units of MM:SS = minutes:seconds) for the case of the $N \times N = 8{,}192 \times 8{,}192$ mesh, which is the largest case we use for comparison here. We find that the KNL using the MCDRAM is dramatically faster than KNC and CPUs in all cases. Despite DDR4 being a slower form of memory, KNL using DDR4 is comparable in most cases to KNC using its GDDR5. For MCDRAM and DDR4 on the KNL, using more threads than MPI processes is significantly faster than the inverse. In final analysis, with the best combination of MPI processes and OpenMP threads per process, the KNL can be approximately 4x faster than either KNC or two CPUs, provided the problem fits into its 16 GB of on-chip MCDRAM memory, which is a signficant amount of memory and sufficient for many applications.

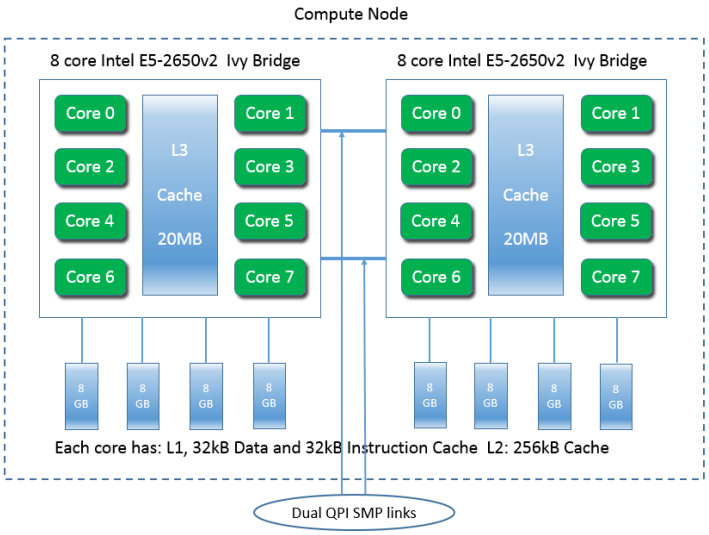


Figure 2.1: Schematic of one compute node on maya.

Section 5 summarizes our conclusions and suggests opportunities for more studies.

At the time of this writing (December 2016), the KNL has just begun to be available to the general public, and few performance comparisons are available. One report is an announcement [11] by Intel of results of the HPCG Benchmark that characterizes speedup of KNL over two 18-core CPUs (Intel E5-2697v4) as approximately 2x. The HPCG Benchmark [4] is essentially a more sophisticated implementation of a discretization of the 3-D version of (1.1) and designed to provide a modern reference for performance [2]. Its results are reported at `hpcg-benchmark.org`. Our test problem has analogous features and its implementation is purposefully chosen even simpler to allow for easier reproducibility. We hope that our results round out the characterization of [11] with concrete performance data and give initial guidance how to run memory-bound code on the KNL.

2 Hardware

2.1 CPU Hardware

The CPU baseline results were obatained on Stampede at the Texas Advanced Computing Center (TACC). One compute node contains two eight-core 2.7 GHz Intel E5-2680 Sandy Bridge CPUs and 32 GB memory.

The scalability studies using CPUs with MPI only were conducted on maya. The maya cluster is a heterogeneous cluster with equipment acquired between 2009 and 2013. It contains a total of 324 nodes, 38 GPUs and 38 Intel Xeon Phi coprocessors, and over 8 TB of main memory. Our work focuses on the newer 2013 section which has 72 nodes in Dell 4220 cabinets with 34 PowerEdge R620 CPU-only compute nodes, 19 PowerEdge R720 CPU/GPU compute nodes, and 19 PowerEdge R720 CPU/Phi compute nodes. All nodes have two Intel E5-2650v2 Ivy Bridge (2.6 GHz, 20 MB cache), processors with eight cores apiece, for a total of 16 cores per node. Each core runs one thread making 16 threads per node. The two CPUs in a node are connected by two Quick Path Interconnects (QPI) for cross communication. CPU/GPU nodes have two NVIDIA K20 GPUs, while CPU/Phi have two Intel Phi 5110P processors. All nodes have 64 GB of main memory, except those designated as user nodes which have 128 GB, and 500 GB of local hard drive.

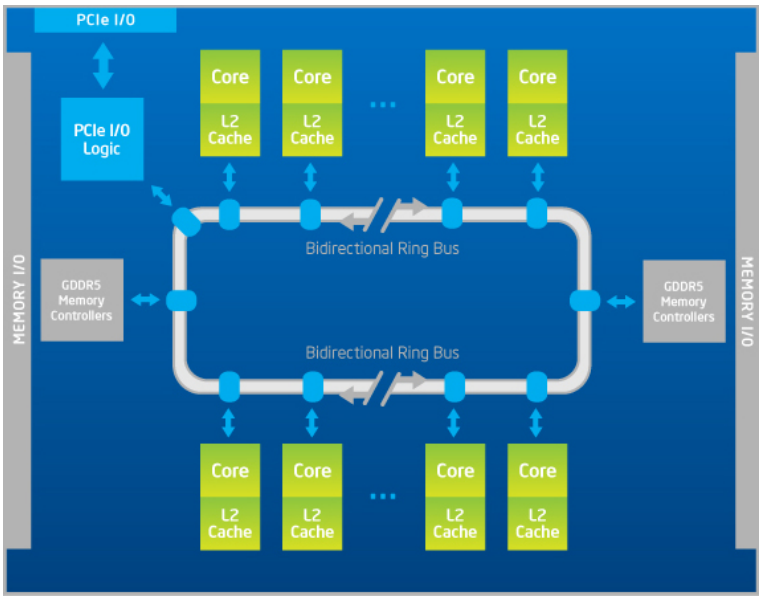


Figure 2.2: Schematic of the Intel Xeon Phi Knights Corner (KNC).

The Xeon E5-2650 v2 CPUs are Ivy Bridge processors based off a 22 nm FinFET architecture [7]. (FinFET is a technology using a 3 dimensional transistor rather than a planer one.) They have 8 cores which amounts to 16 cores per node. They have a base clock of 2.6 GHz with a boost clock of 3.4 GHz.

2.2 Intel Xeon Phi Knights Corner (KNC)

The Intel Xeon Phi is a co-processor designed by Intel to compete with GPU such as the Tesla cards by NVIDIA or the FirePro cards by AMD. It was announced in June of 2011, and the first units were delivered Q4 2012. The cores on a Phi are more capable than the cores in a GPU. GPU cores, such as a CUDA core found in a NVIDIA Tesla card, are quite low powered but numerous at 2496 cores in a NVIDIA K20. In contrast, the first-generation Intel Xeon Phi, codenamed Knights Corner (KNC), has cores more equivalent to that of a CPU. Production model KNCs are also known as Xeon Phi x100 series. Figure 2.2, sourced from Intel, shows the schematic of the KNC [5]. KNC processors can have up to 61 cores. Each core has 4 threads equalling 244 threads. The processors in the KNC are based off the 22 nm Ivy Bridge architecture. As rough comparison, this equates to about 60 basic laptop CPUs connected on a bi-direction ring bus. The cores in KNC each have one 512-bit Vector Processing Unit (VPU). This VPU can do 16 single or 8 double floating point operations per cycle. Also connected to this bus is the 8 GB of GDDR5 memory, the same kind as used in GPUs. These can be seen on the sides of the ring in Figure 2.2. The 2 memory channels split between the cores on the ring evenly. This bus also has a PCIe Gen 3 x16 slot, this is how the KNC is connected to the rest of the system. This connection can be seen on the top-left of Figure 2.2. Most importantly the PCIe bus connects the KNC to the DDR4. The KNC only has a Linux micro-OS, this means it can only be run as a co-processor. Our model of the KNC is on the Stampede cluster in TACC.

2.3 Intel Xeon Phi Knights Landing (KNL)

The new second-generation architecture of the Phi is codenamed Knights Landing (KNL). The production models of KNL models are known as Xeon Phi x200 series. For our testing we had access

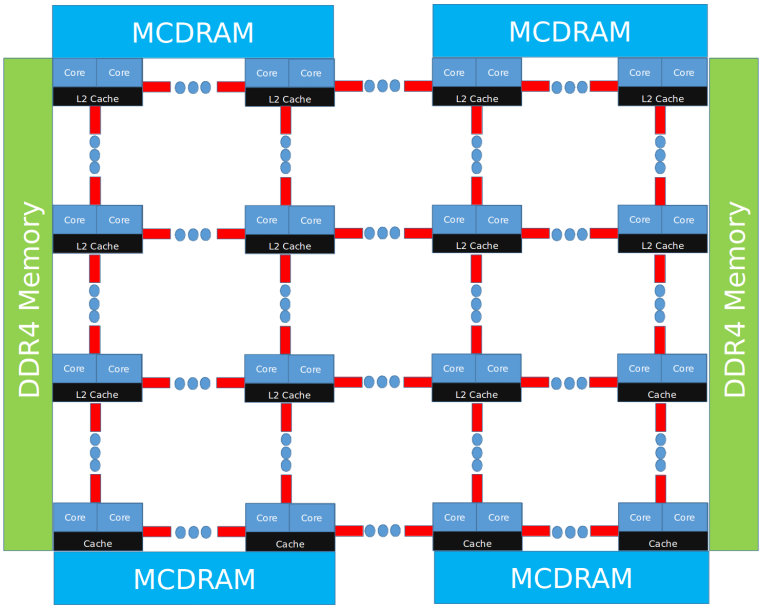


Figure 2.3: Schematic of the Intel Xeon Phi Knights Landing (KNL).

to Grover, a KNL testing server, at the University of Oregon. The model of the KNL in Grover is the Xeon Phi 7250 [6]. This new generation can have up to 72 cores. The pre-production model 7250 we have access to has access to 68 cores. Grover is a single node server with a Intel Xeon Phi 7250 engineering sample running as an independent processor. The Figure 2.3 shows the internal schematic of the KNL processor. These processors are based on a 14 nm Airmont process technology, similar to CPUs found in low power tablet computers. The processors are also now connected on a more complicated 2D mesh network instead of a bi-direction ring. This mesh is similar to the topology of a GPU and allows multitudes more connections between cores. The mesh arranges the 68 cores into 34 tiles with 2 cores each. The 2 cores on each of these tiles share an L2 cache. These cores are out of order to make better use of resources like memory. Each core has 4 threads adding up to 272 threads. The KNL also has two different types of memory to access, DDR4 and MCDRAM. The DDR4 is system dependent, meaning that this is the same DDR4 that every node has access to. In the Grover server we are using it is over 98 GB of DDR4 in the single node. MCDRAM (Multi-Channel DRAM) is a new form of HMC (Hybrid Memory Cube) also known as stacked memory. MCDRAM allows speeds of up to 500 GB/s to 16 GB of RAM across 4 channels. This is 5 times faster bandwidth than DDR4 and nearly 50% more bandwidth compared to GDDR5 used in KNC and GPUs like the K20. This is thanks to the MCDRAM being on each KNL chip. The MCDRAM on Grover is configured in flat mode. This means that MCDRAM is used as addressable memory. MCDRAM may also be used as cache in cache mode or as a combination of addressable memory and cache in hybrid mode. Another major improvement is the doubling of Vector Processing Units (VPUs), from one per KNC core to two per KNL core. Each VPU is 512 bits wide, allowing for 16 single or 8 double precision operations per clock cycle; thus, this allows 16 double precision additions to happen at the same time per core [12]. The KNL also runs a full Linux-based OS. This allows it to be run as either a processor or co-processor. As a co-processer the KNL functions similar to a KNC or GPU over a PCIe Gen 3 x16 connection. As a full processor, the KNL replaces a CPU in a node and runs through the LGA 3647 socket connection. KNL models releasing Q4 2016 will have an optional fiber connection in addition to the socket allowing for faster connections.

3 Test Problem

We consider the classical elliptic test problem of the Poisson equation with homogeneous Dirichlet boundary conditions in (1.1), as used in many Numerical Linear Algebra textbooks as standard example, e.g., [1, Section 6.3], [3, Subsection 9.1.1], [9, Chapter 12], and [13, Section 8.1], and researchers should be easily able to recreate the results. Using $N+2$ mesh points in each dimension, we construct a mesh with uniform mesh spacing $h = 1/(N+1)$. Specifically, define the mesh points $(x_{k_1}, x_{k_2}) \in \overline{\Omega} \subset \mathbb{R}^2$ with $x_{k_i} = h\,k_i$, $k_i = 0, 1, \ldots, N, N+1$, in each dimension $i = 1, 2$. Denote the approximations to the solution at the mesh points by $u_{k_1,k_2} \approx u(x_{k_1}, x_{k_2})$. Then approximate the second-order derivatives in the Laplace operator at the N^2 interior mesh points by

$$\frac{\partial^2 u(x_{k_1}, x_{k_2})}{\partial x_1^2} + \frac{\partial^2 u(x_{k_1}, x_{k_2})}{\partial x_2^2} \approx \frac{u_{k_1-1,k_2} - 2u_{k_1,k_2} + u_{k_1+1,k_2}}{h^2} + \frac{u_{k_1,k_2-1} - 2u_{k_1,k_2} + u_{k_1,k_2+1}}{h^2} \quad (3.1)$$

for $k_i = 1, \ldots, N$, $i = 1, 2$, for the approximations at the interior points. Using this approximation together with the homogeneous boundary conditions (1.1) as determining conditions for the approximations u_{k_1,k_2} gives a system of N^2 linear equations

$$-u_{k_1,k_2-1} - u_{k_1-1,k_2} + 4u_{k_1,k_2} - u_{k_1+1,k_2} - u_{k_1,k_2+1} = h^2\, f(x_{k_1}, x_{k_2}), \quad k_i = 1, \ldots, N, \quad i = 1, 2, \quad (3.2)$$

for the finite difference approximations u_{k_1,k_2} at the N^2 interior mesh points $k_i = 1, \ldots, N$, $i = 1, 2$.

Collecting the N^2 unknown approximations u_{k_1,k_2} in a vector $u \in \mathbb{R}^{N^2}$ using the natural ordering of the mesh points with $k = k_1 + N(k_2 - 1)$ for $k_i = 1, \ldots, N$, $i = 1, 2$, we can state the problem as a system of linear equations in standard form $A\,u = b$ with a system matrix $A \in \mathbb{R}^{N^2 \times N^2}$ and a right-hand side vector $b \in \mathbb{R}^{N^2}$. The components of the right-hand side vector b are given by the product of h^2 multiplied by right-hand side function evaluations $f(x_{k_1}, x_{k_2})$ at the interior mesh points using the same ordering as the one used for u_{k_1,k_2}. The system matrix $A \in \mathbb{R}^{N^2 \times N^2}$ can be defined recursively as block tri-diagonal matrix with $N \times N$ blocks of size $N \times N$ each. Concretely, we have $A = \text{block-tridiag}(T, S, T) \in \mathbb{R}^{N^2 \times N^2}$ with the tri-diagonal matrix $S = \text{tridiag}(-1, 4, -1) \in \mathbb{R}^{N \times N}$ for the diagonal blocks of A and with $T = -I \in \mathbb{R}^{N \times N}$ denoting a negative identity matrix for the off-diagonal blocks of A.

For fine meshes with large N, iterative methods such as the conjugate gradient method are appropriate for solving this linear system [3]. The system matrix A is known to be symmetric positive definite and thus the method is guaranteed to converge for this problem [3]. In a careful implementation, the conjugate gradient method requires in each iteration exactly two inner products between vectors, three vector updates, and one matrix-vector product involving the system matrix A. In fact, this matrix-vector product is the only way, in which A enters into the algorithm. Therefore, a so-called matrix-free implementation of the conjugate gradient method is possible that avoids setting up any matrix, if one provides a function that computes as its output the product vector $v = A\,u$ component-wise directly from the components of the input vector u by using the explicit knowledge of the values and positions of the non-zero components of A, but without assembling A as a matrix. In fact, the formula for v_{k_1,k_2} is the left-hand side of (3.2).

Thus, without storing A, a careful, efficient, matrix-free implementation of the (unpreconditioned) conjugate gradient method only requires the storage of four vectors (commonly denoted as the solution vector x, the residual r, the search direction p, and an auxiliary vector q). In a parallel implementation of the conjugate gradient method, each vector is split into as many blocks as parallel processes are available and one block distributed to each process. That is, each parallel process possesses its own block of each vector, and normally no vector is ever assembled in full on any process. To understand what this means for parallel programming and the performance of the method, note that

an inner product between two vectors distributed in this way is computed by first forming the local inner products between the local blocks of the vectors and second summing all local inner products across all parallel processors to obtain the global inner product. This summation of values from all processes is known as a reduce operation in parallel programming, which requires a communication among all parallel processes. This communication is necessary as part of the numerical method used, and this necessity is responsible for the fact that for a strong scalability study with a fixed problem size eventually for very large numbers of processors the time needed for communication — increasing with the number of processes — will unavoidably dominate over the time used for the calculations that are done simultaneously in parallel — decreasing due to shorter local vectors for increasing number of processes. By contrast, the vector updates in each iteration can be executed simultaneously on all processes on their local blocks, because they do not require any parallel communications. However, this requires that the scalar factors that appear in the vector updates are available on all parallel processes. This is accomplished already as part of the computation of these factors by using a so-called Allreduce operation, that is, a reduce operation that also communicates the result to all processes. This is implemented in the MPI function `MPI_Allreduce`. Finally, the matrix-vector product $v = A\,u$ also computes only the block of the vector v that is local to each process. But since the matrix A has non-zero off-diagonal elements, each local block needs values of u that are local to the two processes that hold the neighboring blocks of u. The communications between parallel processes thus needed are so-called point-to-point communications, because not all processes participate in each of them, but rather only specific pairs of processes that exchange data needed for their local calculations. Observe now that it is only a few components of v that require data from u that is not local to the process. Therefore, it is possible and potentially very efficient to proceed to calculate those components that can be computed from local data only, while the communications with the neighboring processes are taking place. This technique is known as interleaving calculations and communications and can be implemented using the non-blocking MPI communications commands `MPI_Isend` and `MPI_Irecv`. In the hybrid MPI+OpenMP implementation, all expensive `for` loops with large trip counts are parallelized with OpenMP. This includes the loops in the computation of the matrix vector product, axpby operations, and computation of dot products. These `for` loops are parallelized with the OpenMP pragma `#pragma omp parallel for`, instructing each MPI process to divide the iterations of the loop between available threads and execute in parallel.

For Table 3.1, the memory usage of the code is predicted by noting that there are $4N^2$ double-precision numbers needed to store the four vectors of significant length N^2 and that each double-precision number needs 8 bytes. Taking this result and dividing it by 1024^3 converts the value to units of GB, as shown in the table. The largest mesh resolution that can be solved on all systems under comparison, including the KNC with its 8 GB of GDDR5 on-chip memory, is $N = 8{,}192$. On the KNL with 16 GB of MCDRAM, we can solve one larger mesh, with $N = 16{,}384$.

Table 3.1: Predicted memory for test problem on $N \times N$ mesh resulting in a linear system with n variables. Our matrix-free implementation of CG uses the optimal number of four vectors.

$N \times N$	$n = N^2$	# of doubles ($4N^2$)	memory (GB)
1,024 × 1,024	1,048,576	4,194,304	< 0.100
2,048 × 2,048	4,194,304	16,777,216	0.125
4,096 × 4,096	16,777,216	67,108,864	0.500
8,192 × 8,192	67,108,864	268,435,456	2.000
16,384 × 16,384	268,435,456	1,073,741,824	8.000
32,768 × 32,768	1,073,741,824	4,294,967,296	32.000
65,536 × 65,536	4,294,967,296	17,179,869,184	128.000

4 Results

Sections 4.1, 4.2, and 4.3 describe the parallel scalability studies using CPUs only on the 2013 portion of maya. This section reports the performance studies using hybrid CPU+OpenMP code for the solution of the test problem on one CPU node in Section 4.4, on one Intel Xeon Phi KNC in Section 4.5, and on one Intel Xeon Phi KNL in Section 4.6.

4.1 Performance Study for Mix of CPU and Phi Nodes without BLAS on maya

In this subsection we analyze the results of the time trial runs for the Poisson equation code running without excluding the Phi nodes from the 2013 portion of the maya cluster. Thus there is a mix of Phi and CPU's being used in these runs. We have tested this code on square matrices of $N = 1024$, 2048, and 4096 (i.e., of matrices of around 1 million, 4 million, and 16 million entries respectively)

Table 4.1 organizes the results in the form of a strong scalability study, that is, there is one row for each problem size, with columns for increasing number of parallel processes p. Table 4.1 (a) and (b) lists the raw timing data, organized by numbers of parallel processes p. Tables 4.1 (c) and (d) show the numbers for speedup and efficiency, respectively, that will be visualized in Figures 4.1 (a) and (b), respectively. It becomes clear that there are several choices for most values of p, such as for instance for $p = 4$, one could use 2 nodes with 2 processes per node or 1 node with 4 processes per node. For this study, for $p \geq 16$ we use 16 processes per node, increasing the number of nodes to achieve higher numbers of p. For $p < 16$, only one node is used, with the remaining cores idle. Comparing adjacent columns in the raw timing data in Table 4.1 (a) yields some interesting observations. For $N = 1024$ performance improvement is very good from 1 to 2 processes and from 2 to 4 processes, but not quite as good from 4 to 8 processes, but becomes quite good again from 8 to 16 processes. Since the table collects the results using 16 processes per node, the next columns double the numbers of nodes. We observe that when this happens the improvement in runtime becomes much smaller from 16 to 32 processes than from any previous jump in processes. From 32 to 64 processes and onwards we observe that runtimes actually increase as processes increase. This can be interpreted as the communication cost between nodes being more impactful than any savings in computation time gained through parallelization for $N = 1024$.

For $N = 2048$ we observe that the performance improvement is very good between 1 to 2 processes, but improvements are sub-optimal from 2 processes up to 16 processes, with decreasing efficiency. Then, from 16 to 32 processes we see a near halving of runtime, a large increase in efficiency. We see efficiency continue to increase up to $p = 64$. For $p = 128$ and $p = 256$ we see runtime start to stabilize around 2 seconds, with efficiency falling. This we can summarize at around $p = 64$ (i.e., 4 nodes with 16 processes per node) is when communication cost equalizes with increased computation power from more nodes.

For $N = 4096$ we observe that for $p \leq 16$ the performance improvement starts out very good from 1 to 2 processes, but that efficiency declines with each step up to $p = 16$. Then, from 16 to 32 processes run time roughly halves and efficiency remains stable. Up to $p \leq 256$ the halving of run times with each step continues and efficiency continues to remain consistent, even increasing slightly with the higher number of processes. This means that with a doubling of nodes we are achieving a halving of run time, a relatively optimal speedup performance. This goes to show that generally the larger the problem size the more effective parallelization will be.

The plots in Figures 4.1 (a) and (b) visualize the numbers in Tables 4.1 (c) and (d), respectively. These plots do not provide new data but simply provide a graphical representation of the results in Table 4.1. It is customary in results for fixed problem sizes that the speedup is better for larger problems, since the increased communication time for more parallel processes does not dominate

Table 4.1: Performance study for mix of CPU and Phi nodes, no BLAS.

(a) Observed wall clock time in seconds									
	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$	$p=256$
$N=1024$	9.65	3.39	1.78	1.20	0.55	0.43	0.97	0.69	0.98
$N=2048$	96.78	48.59	35.02	32.46	20.55	6.93	1.96	1.81	1.97
$N=4096$	824.92	397.77	236.91	273.15	161.73	83.34	41.90	16.59	7.11
(b) Observed wall clock time in HH:MM:SS									
	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$	$p=256$
$N=1024$	00:00:10	00:00:03	00:00:02	00:00:01	00:00:01	00:00:00	00:00:01	00:00:01	00:00:01
$N=2048$	00:01:37	00:00:49	00:00:35	00:00:32	00:00:21	00:00:07	00:00:02	00:00:02	00:00:02
$N=4096$	00:13:45	00:06:38	00:03:57	00:04:33	00:02:42	00:01:23	00:00:42	00:00:17	00:00:07
(c) Observed speedup									
	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$	$p=256$
$N=1024$	1.00	2.85	5.42	8.04	17.55	22.44	9.95	13.99	9.85
$N=2048$	1.00	1.99	2.76	2.98	4.71	13.97	49.38	53.47	49.13
$N=4096$	1.00	2.07	3.48	3.02	5.10	9.90	19.69	49.72	116.02
(d) Observed efficiency									
	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$	$p=256$
$N=1024$	1.00	1.42	1.36	1.01	1.10	0.70	0.16	0.11	0.04
$N=2048$	1.00	1.00	0.69	0.37	0.29	0.44	0.77	0.42	0.19
$N=4096$	1.00	1.04	0.87	0.38	0.32	0.31	0.31	0.39	0.45
(e) Observed total memory usage in MB									
	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$	$p=256$
$N=1024$	56	68	105	179	326	654	1344	2702	5555
$N=2048$	152	165	202	275	425	746	1414	2805	5771
$N=4096$	536	549	586	660	811	1136	1804	3203	6205

over the calculation time as quickly as it does for small problems. This is born out generally by both plots in Figure 4.1. Specifically, the speedup in Figure 4.1 (a) appears near-optimal up to $p = 256$ for problem size $N = 4096$, but behaves much differently for the smaller problem sizes. One would expect that the efficiency plot in Figure 4.1 (b) would not add much clarity, since its data are directly derived from the speedup data. But the efficiency plot can provide insight into behavior for small p, where the better-than-optimal behavior is noticeable now. This can happen due to experimental variability of the runs, for instance, if the single-process timing $T_1(N)$ used in the computation of $S_p = T_1(N)/T_p(N)$ happens to be slowed down in some way. Another reason for excellent performance can also be that runs on several processes result in local problems that fit better into the cache of each processor, which leads to fewer cache misses and thus potentially dramatic improvement of the run time, beyond merely distributing the calculations to more processes.

4.2 Performance Study excluding Phi Nodes without BLAS on maya

These are the performance studies that exclude the Phi nodes.

4.3 Performance Study excluding Phi Nodes with BLAS on maya

This section is the results when excluding the Phi nodes and adding BLAS to our code.

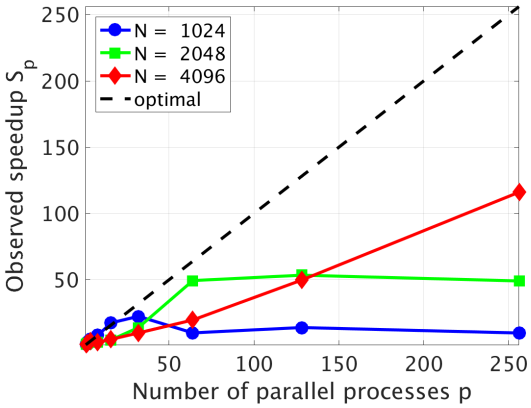
We put the compile option `-mkl` in the `CFLAGS` and the define `-DBLAS` in the `DEFS` of the Makefile.

Table 4.2: Performance study for CPUs, excluding Phi nodes, no BLAS.

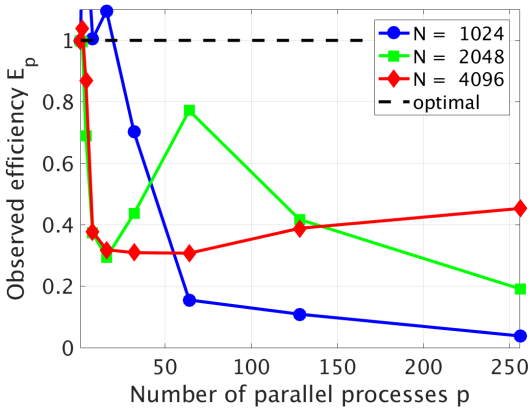
(a) Observed wall clock time in seconds									
	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$	$p=256$
$N=1024$	9.80	3.52	1.80	1.17	1.24	0.52	0.53	0.66	0.87
$N=2048$	97.76	48.29	35.46	33.03	19.77	7.07	2.00	3.10	2.36
$N=4096$	876.33	538.52	235.49	254.26	163.14	81.84	41.81	16.28	7.50
(b) Observed wall clock time in HH:MM:SS									
	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$	$p=256$
$N=1024$	00:00:10	00:00:04	00:00:02	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01
$N=2048$	00:01:38	00:00:48	00:00:35	00:00:33	00:00:20	00:00:07	00:00:02	00:00:03	00:00:02
$N=4096$	00:14:36	00:08:59	00:03:55	00:04:14	00:02:43	00:01:22	00:00:42	00:00:16	00:00:08
(c) Observed speedup									
	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$	$p=256$
$N=1024$	1.00	2.78	5.44	8.38	7.90	18.85	18.49	14.85	11.26
$N=2048$	1.00	2.02	2.76	2.96	4.94	13.83	48.88	31.54	41.42
$N=4096$	1.00	1.63	3.72	3.45	5.37	10.71	20.96	53.83	116.84
(d) Observed efficiency									
	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$	$p=256$
$N=1024$	1.00	1.39	1.36	1.05	0.49	0.59	0.29	0.12	0.04
$N=2048$	1.00	1.01	0.69	0.37	0.31	0.43	0.76	0.25	0.16
$N=4096$	1.00	0.81	0.93	0.43	0.34	0.33	0.33	0.42	0.46
(e) Observed total memory usage in MB									
	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$	$p=256$
$N=1024$	56	69	105	179	329	658	1335	2701	5560
$N=2048$	152	165	202	275	424	753	1427	2817	5766
$N=4096$	536	549	586	660	812	1145	1805	3191	6180

Table 4.3: Performance study for CPUs, excluding Phi nodes, with BLAS.

(a) Observed wall clock time in seconds									
	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$	$p=256$
$N=1024$	9.61	12.89	14.43	17.64	22.18	14.93	13.98	14.16	15.32
$N=2048$	146.49	143.99	119.95	111.85	137.77	82.73	51.91	36.66	33.49
$N=4096$	1195.05	1183.65	1202.67	1177.18	1073.76	550.93	301.70	182.47	122.47
(b) Observed wall clock time in HH:MM:SS									
	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$	$p=256$
$N=1024$	00:00:10	00:00:13	00:00:14	00:00:18	00:00:22	00:00:15	00:00:14	00:00:14	00:00:15
$N=2048$	00:02:26	00:02:24	00:02:00	00:01:52	00:02:18	00:01:23	00:00:52	00:00:37	00:00:33
$N=4096$	00:19:55	00:19:44	00:20:03	00:19:37	00:17:54	00:09:11	00:05:02	00:03:02	00:02:02
(c) Observed speedup									
	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$	$p=256$
$N=1024$	1.00	0.75	0.67	0.54	0.43	0.64	0.69	0.68	0.63
$N=2048$	1.00	1.02	1.22	1.31	1.06	1.77	2.82	4.00	4.37
$N=4096$	1.00	1.01	0.99	1.02	1.11	2.17	3.96	6.55	9.76
(d) Observed efficiency									
	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$	$p=256$
$N=1024$	1.00	0.37	0.17	0.07	0.03	0.02	0.01	0.01	0.00
$N=2048$	1.00	0.51	0.31	0.16	0.07	0.06	0.04	0.03	0.02
$N=4096$	1.00	0.50	0.25	0.13	0.07	0.07	0.06	0.05	0.04
(e) Observed total memory usage in MB									
	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$	$p=256$
$N=1024$	57	77	123	206	392	782	1551	2952	6141
$N=2048$	153	173	221	299	480	848	1656	3228	6517
$N=4096$	538	553	594	679	881	1246	1968	3468	7047

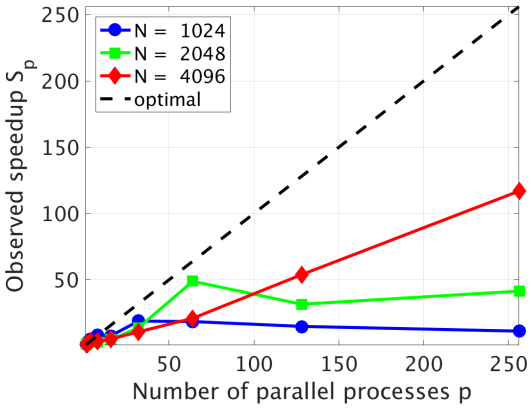


(a) Speedup

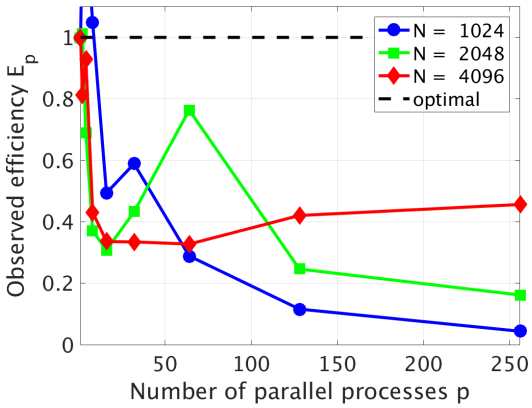


(b) Efficiency

Figure 4.1: Performance study for mix of CPU and Phi nodes, no BLAS.

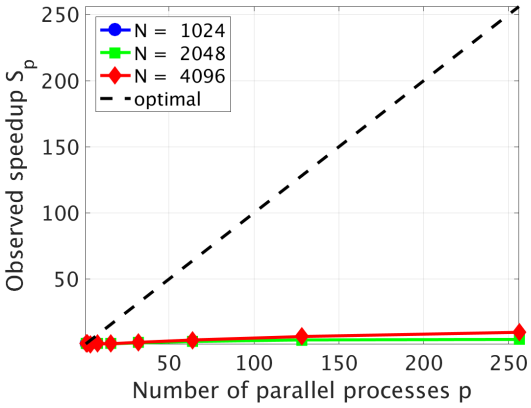


(a) Speedup

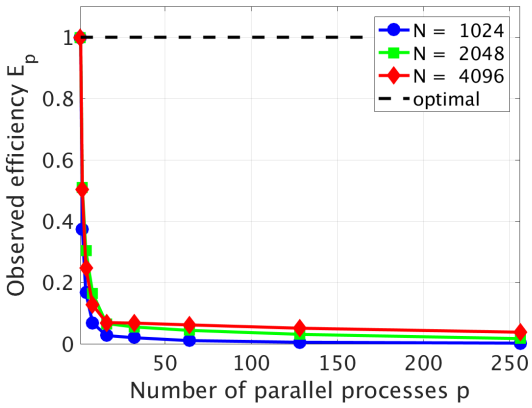


(b) Efficiency

Figure 4.2: Performance study for CPUs, excluding Phi nodes, no BLAS.



(a) Speedup



(b) Efficiency

Figure 4.3: Performance study for CPUs, excluding Phi nodes, with BLAS.

4.4 CPU Performance Studies on Stampede

Table 4.4 reports the baseline performance results using only state-of-the-art 8-core CPUs in one dual-socket node, for comparison with the Intel Xeon Phi studies. In the studies using 1 CPU node, we tested hybrid MPI+OpenMP code such that we utilized all cores on the node, so 2 CPUs with 8 threads per CPU for a total 16 threads, thus running 1 threads per core. These studies used the DDR4 RAM available on Stampede. The `KMP_Affinity=scatter` option was used. Table 4.4 demonstrates that different choices of MPI processes and OpenMP threads per process can potentially make a difference in performance. These results are taken from [10].

Table 4.4: Observed wall clock times in units of MM:SS on one CPU node with two 8-core CPUs on Stampede using 16 threads and DDR4 memory.

CPU — Stampede — DDR4					
MPI proc	1	2	4	8	16
Threads per proc	16	8	4	2	1
1024×1024	00:02	00:01	00:01	00:01	00:01
2048×2048	00:35	00:20	00:20	00:20	00:20
4096×4096	05:11	02:42	02:42	02:42	02:42
8192×8192	36:18	21:57	21:58	22:02	21:48

4.5 KNC Performance Studies on Stampede

Table 4.5 collects the performance results using 1 Intel Phi KNC on Stampede with 61 computational cores. In these studies, we tested hybrid MPI+OpenMP code in native mode on 60 of the 61 KNC cores, leaving one core for the operating system. MPI processes times OpenMP threads were always equal 240, using 4 threads per core. We exclusively used the GDDR5 on board the KNC. We can see from Table 4.5 that for most combinations of MPI processes and threads, the KNC using GDDR5 is slightly slower than the CPU using DDR4 on Stampede.

Table 4.5: Observed wall clock times in units of MM:SS on 1 KNC of Stampede using 240 threads and GDDR5 memory using native mode.

KNC — Stampede — GDDR5										
MPI proc	1	2	4	8	15	16	30	60	120	240
Threads per proc	240	120	60	30	16	15	8	4	2	1
1024×1024	00:03	00:02	00:02	00:02	00:02	00:02	00:02	00:02	00:04	00:13
2048×2048	00:17	00:16	00:16	00:15	00:15	00:15	00:15	00:15	00:18	00:26
4096×4096	01:53	01:48	01:46	01:45	01:57	01:45	01:51	01:48	01:52	02:33
8192×8192	28:24	28:20	27:51	23:08	23:06	23:00	22:24	22:45	22:43	25:37

The KNC is only a co-processor and needs to be used in a hybrid node that also contains a CPU. A possible and typical arrangement is that of two CPUs and two KNC in one node. In this arrangement, besides using the KNC itself in native mode, it can also be used in symmetric and in offload mode. In symmetric mode, all resources of the node are used simultaneously. For the mesh resolution $N = 8{,}192$ problem, the best performance achieved on a hybrid node with two 8-core CPUs and two Phi KNCs is 09:51 [10, Table 2.7.1]. Performance in symmetric mode was achieved by running with 16 MPI processes and 15 OpenMP threads per process on each Phi and 16 MPI

processes and 1 OpenMP thread per process on each CPU. Using all available resources on a hybrid node thus results in a 2.27x improvement in runtime over native mode on 1 KNC. We mention the performance of 27:58 in offload mode, using both Phi in the node [10, Table 2.7.1]; the two CPUs only facilitate the MPI communication between the Phis in this mode, with all calculations taking place on the Phis. The poor performance of offload mode is due to the restriction of multithreading only, not MPI parallelism, in offload regions on the Phi. This explains why this performance is consistent with the result for 1 MPI process and 240 threads per process in Table 4.5.

Going forward, we focus on results in native mode for the KNC, since all runs on a KNL processor must, by definition, be in native mode.

4.6 KNL Performance Studies on Grover

These are the performance studies for the KNL in the Grover server, using both the DDR4 RAM on the node and the MCDRAM on board the KNL.

4.6.1 KNL DDR4 RAM

In the studies in Table 4.6, we tested hybrid MPI+OpenMP code such that we utilized all 68 cores of this model of the KNL. MPI processes times OpenMP threads were always equal 272, with 4 threads per core. We exclusively used the DDR4 RAM of the node. We can see from Table 4.6 that for all combinations of MPI processes and threads, the KNL using DDR 4 is slower than the best KNC results using GDDR5 RAM. We note that using more MPI processes than threads results in much slower run times than when using more threads than MPI processes.

Table 4.6: Observed wall clock times in units of MM:SS on 1 KNL of Grover using 272 threads and DDR4 memory, and `KMP_Affinity=scatter`.

KNL — DDR4 RAM — Scatter										
MPI proc	1	2	4	8	16	17	34	68	136	272
Threads per proc	272	136	68	34	17	16	8	4	2	1
1024×1024	00:03	00:03	00:02	00:02	00:02	00:03	00:02	00:03	00:04	00:16
2048×2048	00:25	00:24	00:24	00:23	00:24	00:34	00:34	00:34	00:37	00:43
4096×4096	03:26	03:19	03:08	03:10	03:15	04:50	04:55	04:45	04:55	05:15
8192×8192	26:02	25:07	24:38	24:25	24:24	36:29	37:40	37:54	39:06	41:00

4.6.2 KNL MCDRAM

In the studies in Table 4.7, we tested hybrid MPI+OpenMP code such that we utilized all 68 cores of the KNL. MPI processes times OpenMP threads were equal 272, using 4 threads per core. We exclusively used the MCDRAM available on board of the KNL. Comparing corresponding mesh resolutions up to $N = 8{,}192$, the largest mesh resolution reported in the previous tables, we can see from both Tables 4.7 (a) and (b) that for all combinations of MPI processes and threads, the KNL using MCDRAM is significantly faster than the KNC using GDDR5 RAM in native mode and also faster than CPU-KNC results using symmetric mode. This better performance allows to add the $N = 16{,}384$ mesh resolution to the study, since it does not require excessive run times any more, which is an additional row in each sub-table. We note that using more MPI processes than threads results in much slower run times than when using more threads than MPI processes.

Table 4.7 (a) shows results with the OpenMP option `KMP_Affinity=scatter`, while Table 4.7 (b) with the option `KMP_Affinity=compact`. `Compact`, as the name implies, tries to keep the threads closer together. `Scatter` tries instead to distribute the threads to cores as evenly as possible. We can see for our results up to the N = 8,192 mesh resolution that `compact` is slightly faster then `scatter`. `Scatter` was preferred in CPU runs, first because `compact` forces the threads to one CPU on the node first, limiting resources and leaving the other CPU idle. Secondly, the cores are in order on a CPU so logical neighbors share resources. On KNL, each core is capable of running 4 threads and has its own L1 cache, while each tile has two cores and has its own L2 cache. Using `compact` allows for increased cache utilization between threads on the same core or tile. Since `scatter` distributes threads, it is less likey that cache can be reused between neighboring threads. This explains why `compact` outperforms scatter for mesh resolutions up to N = 8,192. Interestingly, for the N = 16,384 mesh resolution `scatter` outperforms `compact`. It is unclear why this occurs but it is possible that since neighboring threads are further apart when using `scatter`, memory bandwidth may be increased. It is also possible that for this mesh resolution threads are distributed in a manner such that neighboring threads actually are able to reuse cache.

Table 4.7: Observed wall clock times in units of MM:SS on 1 KNL of Grover using all 272 threads and MCDRAM memory, and two settings of `KMP_Affinity`.

(a) KNL — MCDRAM — Scatter

MPI proc	1	2	4	8	16	17	34	68	136	272
Threads per proc	272	136	68	34	17	16	8	4	2	1
1024×1024	00:01	00:01	00:01	00:01	00:01	00:02	00:01	00:02	00:02	00:13
2048×2048	00:07	00:07	00:07	00:07	00:07	00:09	00:09	00:10	00:10	00:11
4096×4096	00:49	00:48	00:47	00:47	00:47	01:05	01:04	01:04	01:06	01:12
8192×8192	05:49	05:43	05:39	05:35	05:36	08:22	08:49	08:41	08:37	08:57
16384×16384	41:17	40:52	40:47	40:40	40:39	59:36	61:30	62:05	64:10	63:24

(b) KNL — MCDRAM — Compact

MPI proc	1	2	4	8	16	17	34	68	136	272
Threads per proc	272	136	68	34	17	16	8	4	2	1
1024×1024	00:01	00:01	00:01	00:01	00:01	00:01	00:01	00:01	00:01	00:02
2048×2048	00:05	00:05	00:05	00:05	00:05	00:08	00:08	00:08	00:09	00:10
4096×4096	00:39	00:39	00:38	00:39	00:39	00:59	00:58	00:58	01:00	01:12
8192×8192	05:27	05:14	05:11	05:11	05:13	07:44	08:13	08:02	08:08	09:33
16384×16384	44:47	43:54	43:39	43:31	43:39	63:52	61:30	62:06	64:08	63:38

5 Conclusions

The results in Section 4 provide a comparison of performance using all available cores on CPUs, KNC, and KNL for the memory-bound test problem of the two-dimensional Poisson equation. We found that using MCDRAM on KNL is drastically faster than using CPUs, KNC, or KNL with DDR4 in all cases for this test problem. Despite DDR4 being a slower form of memory than GDDR5, we found that KNL using DDR4 is comparable in most cases to KNC. For MCDRAM and DDR4 on the KNL, using more threads than MPI processes is significantly faster than the inverse. Finally, we found that `KMP_Affinity=compact` was slightly faster than `KMP_Affinity=scatter` for all mesh resulutions except the largest available of N = 16,1384.

These results give only an introduction to how to run code on the KNL. The KNL has additional features that may be utilized to further improve performance. As mentioned in Section 2.3, each core of the KNL has 2 VPUs. As a result, proper vectorization of code is essential to good performance on the KNL. Even on the KNC which only had 1 VPU per core, [10, Table 1.6.2] shows a halving of runtime for a three-dimensional Poisson equation code by manually unrolling loops to improve vectorization. As mentioned in Section 2.3, the KNL may also be configured to use MCDRAM in cache and symmetric modes that increase the size of the cache but may increase latency of cache misses. The KNL may also be configured in different cache clustering modes. Grover was configured in all-to-all mode, meaning that memory addresses are uniformly distributed across the KNL. However, other modes that divide the KNL into hemispheres or quadrants are available that may improve cache locality and thus decrease latency of L2 cache misses. We have provided initial guidance on how to run memory-bound codes on the KNL, but these additional features of provide further opportunities to tune codes for the KNL.

Acknowledgments

These results were obtained as part of the REU Site: Interdisciplinary Program in High Performance Computing (hpcreu.umbc.edu) in the Department of Mathematics and Statistics at the University of Maryland, Baltimore County (UMBC) in Summer 2016. This program is funded by the National Science Foundation (NSF), the National Security Agency (NSA), and the Department of Defense (DOD), with additional support from UMBC, the Department of Mathematics and Statistics, the Center for Interdisciplinary Research and Consulting (CIRC), and the UMBC High Performance Computing Facility (HPCF). HPCF is supported by the U.S. National Science Foundation through the MRI program (grant nos. CNS–0821258 and CNS–1228778) and the SCREMS program (grant no. DMS–0821311), with additional substantial support from UMBC. Co-author Ishmail Jabbie was supported, in part, by the UMBC National Security Agency (NSA) Scholars Program through a contract with the NSA. Graduate assistant Jonathan Graf was supported by UMBC. The authors would like to thank the Performance Research Laboratory, University of Oregon for providing access to the KNL Hardware. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI–1053575. We acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC resources that have contributed to the research results reported within this paper.

References

[1] James W. Demmel. *Applied Numerical Linear Algebra*. SIAM, 1997.

[2] Jack Dongarra and Michael A. Heroux. Toward a new metric for ranking high performance computing systems. Technical Report SAND2013–4744, Sandia National Laboratories, June 2013. https://software.sandia.gov/hpcg/doc/HPCG-Benchmark.pdf, accessed on November 02, 2016.

[3] Anne Greenbaum. *Iterative Methods for Solving Linear Systems*, volume 17 of *Frontiers in Applied Mathematics*. SIAM, 1997.

[4] Michael A. Heroux, Jack Dongarra, and Piotr Luszczek. HPCG technical specification. Technical Report SAND2013–8752, Sandia National Laboratories, October 2013. https://software.sandia.gov/hpcg/doc/HPCG-Specification.pdf, accessed on November 02, 2016.

[5] Intel Xeon Phi coprocessor block diagram. http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-coprocessor-block-diagram.html.

[6] Intel Xeon Phi processor 7250 (16 GB, 1.40 GHz, 68 core) specifications. http://ark.intel.com/products/94035/Intel-Xeon-Phi-Processor-7250-16GB-1_40-GHz-68-core.

[7] Intel Xeon processor E5-2650 v2 (20 MB cache, 2.60 GHz) specifications. http://ark.intel.com/products/75269/Intel-Xeon-Processor-E5-2650-v2-20M-Cache-2_60-GHz.

[8] Intel re-architects the fundamental building block for high-performance computing — Intel newsroom. https://newsroom.intel.com/news-releases/intel-re-architects-the-fundamental-building-block-for-high-performance-computing/, June 23 2014. Accessed November 30, 2016.

[9] Arieh Iserles. *A First Course in the Numerical Analysis of Differential Equations*. Cambridge Texts in Applied Mathematics. Cambridge University Press, second edition, 2009.

[10] Samuel Khuvis. *Porting and Tuning Numerical Kernels in Real-World Applications to Many-Core Intel Xeon Phi Accelerators*. Ph.D. Thesis, Department of Mathematics and Statistics, University of Maryland, Baltimore County, 2016.

[11] Intel Measured Results. *High Performance Conjugate Gradients*. www.umbc.edu/~gobbert/papers/KNL_UMBC.PNG, April 2016.

[12] Avinash Sodani. Intel Xeon Phi processor "Knights Landing" architectural overview. https://www.nersc.gov/assets/Uploads/KNL-ISC-2015-Workshop-Keynote.pdf, 2015. Accessed November 30, 2016.

[13] David S. Watkins. *Fundamentals of Matrix Computations*. Wiley, third edition, 2010.