Parallel Performance Studies for an Elliptic Test Problem on the Stampede2 Cluster and Comparison of Networks

Kritesh Arora[1], Carlos Barajas[2], and Matthias K. Gobbert[2] (gobbert@umbc.edu)

[1]Department of Information Systems, University of Maryland, Baltimore County
[2]Department of Mathematics and Statistics, University of Maryland, Baltimore County

Technical Report HPCF–2018–10, hpcf.umbc.edu > Publications

Abstract

We study the parallel performance of dual-socket compute nodes with Intel Xeon Platinum 8160 Skylake CPUs with 24 cores and 192 GB of memory, connected by a 100 Gbps Intel Omni-Path (OPA) interconnect. The experimenets use the classical test problem of a Poisson equation in two spatial dimensions, discretized by the finite difference method to give a very large and sparse system of linear equations that is solved by the conjugate gradient method. The tests are performanced on the Skylake nodes of Stampede2 in the Texas Advanced Computing Center (TACC) at The University of Texas at Austin. This national supercomputer is funded by National Science Foundation (NSF) and can be accessed through the XSEDE program. We also compare the performance of the test code using different inter-node networks, Omni-Path (OPA), InfiniBand (IB), and Ethernet, on test clusters graciously provided to us by Dell. The results demonstrate excellent scalability when using more nodes due to the low latency of the high-performance interconnect and good speedup when using all cores of the multi-core CPUs. Comparison to past results brings out that core per core performance improvements have stalled, but that node per node performance continues to improve due to the larger number of cores available on a node.

1 Introduction

The UMBC High Performance Computing Facility (HPCF) is the community-based, interdisciplinary core facility for scientific computing and research on parallel algorithms at UMBC. Started in 2008 by more than 20 researchers from ten academic departments and research centers from all three colleges, it is supported by faculty contributions, federal grants, and the UMBC administration. The facility is open to UMBC researchers at no charge. Researchers can contribute funding for long-term priority access. System administration is provided by the UMBC Division of Information Technology, and users have access to consulting support provided by dedicated full-time graduate assistants. See hpcf.umbc.edu for more information on HPCF and the projects using its resources.

In 2017, the user community, represented by 51 researchers from 17 academic departments and research centers across UMBC, was successful for a third time to secure a grant from the National Science Foundation through its MRI program (grant no. OAC–1726023) for the extension and state-of-the-art update of HPCF. The tests in this report were performed to support the purchase decision in 2017–2018 that ultimately resulted in choosing to order a new cluster from Dell. The tests reported here use Stampede2 in the Texas Advanced Computing Center (TACC) at The University of Texas at Austin to test the CPUs themselves and several choices of environment variables, and we used two Dell evaluation clusters to compare three choices of networks.

This report uses the test problem that has been used repeatedly to test cluster performance, including in [7, 8] on maya, which were updates of previous reports [1, 4, 9, 10], which considered the same problem on previous clusters. The problem is the numerical solution of the Poisson equation with homogeneous Dirichlet boundary conditions on a unit square domain in two spatial dimensions. Discretizing the spatial derivatives by the finite difference method yields a system of linear equations with a large, sparse, highly structured, symmetric positive definite system matrix. This linear system is a classical test problem for iterative solvers and contained in several textbooks including [3, 5, 6, 12]. The parallel, matrix-free implementation of the conjugate gradient method as appropriate iterative linear solver for this linear system involves necessarily communications both collectively among all parallel processes and between pairs of processes in every iteration. Therefore, this method provides an excellent test problem for the overall, real-life performance of a parallel computer on a memory-bound algorithm. The results are not just applicable to the conjugate gradient method, which is important in its own right as a representative of the class of Krylov subspace methods, but to all memory-bound algorithms.

The results demonstrate excellent scalability when using more nodes due to the low latency of the high-performance interconnect and good speedup when using all cores of the multi-core CPUs. Comparison to past results brings out that core per core performance improvements have stalled, but that node per node performance continues to improve due to the larger number of cores available on a node.

2 The Elliptic Test Problem

We consider the classical elliptic test problem of the Poisson equation with homogeneous Dirichlet boundary conditions (see, e.g., [12, Chapter 8])

$$\begin{aligned} -\triangle u &= f \qquad \text{in } \Omega, \\ u &= 0 \qquad \text{on } \partial\Omega, \end{aligned} \tag{2.1}$$

on the unit square domain $\Omega = (0,1) \times (0,1) \subset \mathbb{R}^2$. Here, $\partial\Omega$ denotes the boundary of the domain Ω and the Laplace operator in is defined as $\triangle u = \frac{\partial^2 u}{\partial x_1^2} + \frac{\partial^2 u}{\partial x_2^2}$. 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

$$\frac{\partial^2 u(x_{k_1}, x_{k_2})}{\partial x_1^2} + \frac{\partial^2 u(x_{k_1}, x_{k_2})}{\partial x_2^2} \approx \frac{u_{k_1-1,k_2} - 2u_{k_1,k_2} + u_{k_1+1,k_2}}{h^2} + \frac{u_{k_1,k_2-1} - 2u_{k_1,k_2} + u_{k_1,k_2+1}}{h^2} \tag{2.2}$$

for $k_i = 1, \ldots, N$, $i = 1, \ldots, d$, for the approximations at the interior points. Using this approximation together with the homogeneous boundary conditions (2.1) gives a system of N^2 linear equations for the finite difference approximations at the N^2 interior mesh points.

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, 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

$$A = \begin{bmatrix} S & T & & & \\ T & S & T & & \\ & \ddots & \ddots & \ddots & \\ & & T & S & T \\ & & & T & S \end{bmatrix} \in \mathbb{R}^{N^2 \times N^2} \tag{2.3}$$

with the tri-diagonal matrix $S = \operatorname{tridiag}(-1, 4, -1) \in \mathbb{R}^{N \times N}$ 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. The system matrix A is known to be symmetric positive definite and thus the method is guaranteed to converge for this problem. 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 $q = A\,p$ component-wise directly from the components of the input vector p by using the explicit knowledge of the values and positions of the non-zero components of A, but without assembling A as a matrix.

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 fixed problem sizes 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 $q = A\,p$ also computes only the block of the vector q that is local to each process. But since the matrix A has non-zero off-diagonal elements, each local block needs values of p that are local to the two processes that hold the neighboring blocks of p. 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 q that require data from p 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`.

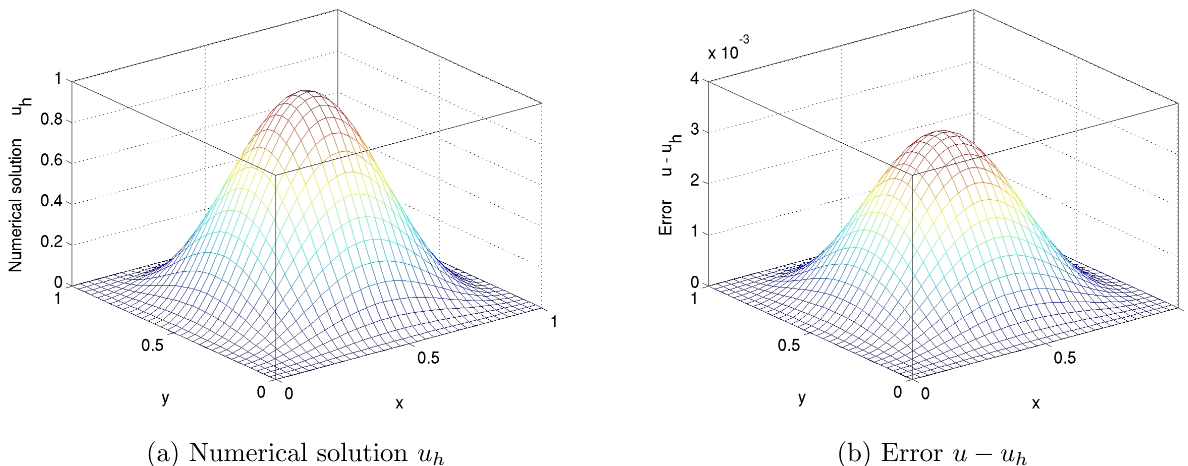
3 Convergence Study for the Model Problem

To test the numerical method and its implementation, we consider the elliptic problem (2.1) on the unit square $\Omega = (0,1) \times (0,1)$ with right-hand side function

$$f(x_1, x_2) = (-2\pi^2) \Big(\cos(2\pi x_1) \sin^2(\pi x_2) + \sin^2(\pi x_1) \cos(2\pi x_2) \Big), \tag{3.1}$$

for which the solution $u(x_1, x_2) = \sin^2(\pi x_1) \sin^2(\pi x_2)$ is known. On a mesh with 33×33 points and mesh spacing $h = 1/32 = 0.03125$, the numerical solution $u_h(x_1, x_2)$ can be plotted vs. (x_1, x_2) as a mesh plot as in Figure 3.1 (a). The shape of the solution clearly agrees with the true solution of the problem. At each mesh point, an error is incurred compared to the true solution $u(x_1, x_2)$. A mesh plot of the error $u - u_h$ vs. (x_1, x_2) is plotted in Figure 3.1 (b). We see that the maximum error occurs at the center of the domain of size about 3.2e–3, which compares well to the order of magnitude $h^2 \approx 0.98\text{e–}3$ of the theoretically predicted error.

To check the convergence of the finite difference method as well as to analyze the performance of the conjugate gradient method, we solve the problem on a sequence of progressively finer meshes. The conjugate gradient method is started with a zero vector as initial guess and the solution is accepted as converged when the Euclidean vector norm of the residual is reduced to the fraction 10^{-6} of the initial residual. Table 3.1 lists the mesh resolution N of the $N \times N$ mesh, the number of degrees of freedom N^2 (DOF; i.e., the dimension of the linear system), the norm of the finite difference error $\|u - u_h\| \equiv \|u - u_h\|_{L^\infty(\Omega)}$, the ratio of consecutive errors $\|u - u_{2h}\| / \|u - u_h\|$, the number of conjugate gradient iterations `#iter`, the observed wall clock time in HH:MM:SS and in seconds, and the predicted and observed memory usage in GB for studies performed in serial. More precisely, the runs used the parallel code run on one process only, on a dedicated node (no other processes running on the node), and with all parallel



(a) Numerical solution u_h (b) Error $u - u_h$

Figure 3.1: Mesh plots of (a) the numerical solution u_h vs. (x_1, x_2) and (b) the error $u - u_h$ vs. (x_1, x_2).

Table 3.1: Convergence study on Stampede2 with serial code except where noted.

N	DOF	$\|u - u_h\|$	Ratio	#iter	wall clock time		memory usage (GB)	
					HH:MM:SS	seconds	predicted	observed
1024	1,048,576	3.1266e–06	—	1,581	00:00:07	6.65	< 1	< 1
2048	4,194,304	7.8019e–07	4.07	3,192	00:01:26	85.98	< 1	< 1
4096	16,777,216	1.9366e–07	4.03	6,452	00:12:44	764.39	< 1	< 1
8192	67,108,864	4.7392e–08	4.09	13,033	01:46:34	6,393.86	2	2.18
16384	268,435,456	1.1548e–08	4.10	26,316	14:29:52	52,192.29	8	8.48
*32768	1,073,741,824	3.0870e–09	3.74	53,141	*00:19:55	*1,195.35	32	*95.01
**65536	4,294,967,296	8.9964e–10	3.43	107,261	**02:38:11	**9,491.42	128	**196.84

*The case $N = 32768$ uses 32 cores on 32 nodes; the observed memory is the total over all processes.

**The case $N = 65536$ uses 32 cores on 32 nodes; the observed memory is the total over all processes.

communication commands disabled by if-statements. The wall clock time is measured using the `MPI_Wtime` command (after synchronizing all processes by an `MPI_Barrier` command). 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 requires 8 bytes; dividing this result by 1024^3 converts its value to units of GB, as quoted in the table. The memory usage is observed in the code by checking the `VmRSS` field in the the special file `/proc/self/status`. The case $N = 32768$ and $N = 65536$ require more time to complete than the wall time available on a Stampede2 with 48 hours. For these cases, 32 cores on 32 nodes are used, with observed memory summed across all running processes to get the total usage.

In nearly all cases, the norms of the finite difference errors in Table 3.1 decrease by a factor of about 4 each time that the mesh is refined by a factor 2. This confirms that the finite difference method is second-order convergent, as predicted by the numerical theory for the finite difference method [2, 6]. The fact that this convergence order is attained also confirms that the tolerance of the iterative linear solver is tight enough to ensure a sufficiently accurate solution of the linear system. For the two finest mesh resolutions, the reduction in error appears slightly more erratic, which points to the tolerance not being tight enough beyond these resolutions. The increasing numbers of iterations needed to achieve the convergence of the linear solver highlights the fundamental computational challenge with methods in the family of Krylov subspace methods, of which the conjugate gradient method is the most important example: Refinements of the mesh imply more mesh points, where the solution approximation needs to be found, and makes the computation of each iteration of the linear solver more expensive. Additionally, more of these more expensive iterations are required to achieve convergence to the desired tolerance for finer meshes. And it is not possible to relax the solver tolerance too much, because otherwise its solution would not be accurate enough and the norm of the finite difference error would not show a second-order convergence behavior, as required by its theory. For the cases up to $N \leq 16384$, the observed memory usage in units of GB rounds to within 0.5 GB of the predicted usage; for $N = 32768$ and 65536, the observed memory shows the total of memory observed on each of the 1024 procsses, which leads to a significant duplication of overhead, thus the observed are quite a bit larger than the predicted memory figures. The good agreement between predicted and observed memory usage in the last two columns of the table indicates that the implementation of the code does not have any unexpected memory usage in the serial case. The wall clock times and the memory usages for these serial runs indicate for which mesh resolutions this elliptic test problem becomes challenging computationally. Notice that the very fine meshes show very significant runtimes and memory usage; parallel computing clearly offers opportunities to decrease runtimes as well as to decrease memory usage per process by spreading the problem over the parallel processes.

We finally note that the results for the finite difference error and the conjugate gradient iterations in Table 3.1 agree with past results for this problem; see [8] and the references therein. This ensures that the parallel performance studies in the next section are practically relevant in that a correct solution of the test problem is computed.

4 Performance Studies on Skylake Nodes with Omni-Path Interconnect on Stampede2 Using MPI-Only Code

This section describes the parallel performance studies for the solution of the test problem on the Skylake portion of Stampede2. There are 1,736 compute nodes, each with two Intel Xeon Platinum 8160 Skylake CPUs with 24 cores at 2.1 GHz and 192 GB of memory. Each CPU has 24 cores making it a total of 48 cores per node; with hyperthreading enabled, there can be up to 96 threads per node. There is 32 kB L1 cache per core, 1 MB L2 cache per core, and 33 MB L3 cache per socket. The nodes in Stampede2 are connected by a 100 Gbps Intel Omni-Path (OPA) interconnect.

The results in this section use the Intel compiler 18.0.0 and Intel MPI 18.0.0. We use the compile options `-xCORE-AVX2 -axCORE-AVX512,MIC-AVX512` to produce multi-architecture binary to allow it run on both Skylake and KNL nodes. The `-xCORE-AVX2` generates a binary for Skylake nodes and `MIC-AVX512` produces a binary for KNL nodes.

The SLURM submission script uses the `ibrun` command to start the job. The number of nodes are controlled by the `--nodes` option in the SLURM submission script, and the number of processes per node by the `--ntasks-per-node` option. Each node that is used is dedicated to the job with remaining cores idling, if not all of them are used, using the `--exclusive` flag.

We included the `OMP_PLACES` and `OMP_PROC_BIND` or `KMP_AFFINITY` environment variables[1] in the SLURM script. The environment variable `OMP_PLACES` is used to list the places threads can be pinned on. `OMP_PROC_BIND` chooses the order of places in the pinning. `KMP_AFFINITY` is used to assign threads to processors but works only for Intel runtime library. The value `OMP_PROC_BIND=close` means that the assignment goes successively through the available places, while `OMP_PROC_BIND=spread` spreads the threads over the places.

We conduct numerical experiments of the test problem for six progressively finer meshes of $N = 1024$, 2048, 4096, 8192, 16384, 32768, 65536. This results in progressively larger systems of linear equations with system dimensions ranging from about 1 million for $N = 1024$ to over 1 billion for $N = 32768$ and over 4 billion for $N = 65536$; the results for the latter two resolutions are contained in Table 3.1. For each mesh resolution, the parallel implementation of the test problem is run on all possible combinations of nodes from 1 to 16 by powers of 2 and processes per node from 1 to 32 by powers of 2. We choose to limit us to not using hyperthreading and to powers of 2, thus 16 processes per CPU is the maximum, with 8 cores idling per CPU. Per node, this allows for up to 32 processes of the available 96 processes, resulting from hyperthreading enabled on the available 48 cores.

Tables 4.1, 4.2, and 4.3 collect the results of the performance studies on Stampede2. The tables summarize the observed wall clock time (total time to execute the code) in HH:MM:SS (hours:minutes:seconds) format. The upper-left entry of each subtable contains the runtime for the serial run of the code for that particular mesh. The lower-right entry of each subtable lists the runtime using 32 cores of both 24-core processors, i.e., not using all available cores, on 16 nodes for a total of 512 parallel processes working together to solve the problem.

Table 4.1 reports the studies using the `OMP_PLACES=cores` and `OMP_PROC_BIND=close` environment variables in the slurm script. When we double the size of the mesh, we can see that the run time increases by a factor of about 8 for corresponding entries in the sub-tables. The run times in each row of each sub-table show that doubling the number of nodes reduces the run time by about a factor 2 each time. The run times in each column of each sub-table show that the run times decrease by a factor of about 2 each time, when going from 1 to 2, from 2 to 4, and from 4 to 8 processes per node used; the decrease is not optimal any more from 8 to 16 and from 16 to 32 processes per node, but the run times still decrease, making the use of all available cores advisable. Also, in each sub-table we can see that the value for anti-diagonals are about equal, that is, the run time for nodes=2 and and processes per node=1 is almost same as for nodes=1 and processes per node=2, for instance.

Table 4.2 reports the studies using the `OMP_PLACES=cores` and `OMP_PROC_BIND=spread` environment variables in the slurm script. The run times were about equal to the run time in Table 4.1.

Table 4.3 reports the studies using the `KMP_AFFINITY=scatter` environment variable in the slurm script. The run times were about same as Tables 4.1 and 4.2.

From the studies using different choices of `OMP_PLACES=cores` and `OMP_PROC_BIND=close`, `OMP_PLACES=cores` and `OMP_PROC_BIND=spread` and, `KMP_AFFINITY=scatter`, we see that the run time are almost same for same values of nodes and processor used. This should have been expected, since the environment variables tested are supposed to influence the placement of OpenMP threads, yet the code used here is an MPI-only parallel code.

[1] http://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-affinity.html

Table 4.1: Wall clock time in HH:MM:SS on Skylake nodes on the Stempede2 cluster using OmniPath network using 1 thread per hardware core with `OMP_PLACES=cores` and `OMP_PROC_BIND=close` using the Intel compiler 18.0.0 with Intel MPI 18.0.0.

(a) Mesh resolution $N \times N = 1024 \times 1024$, system dimension 1048576					
	1 node	2 nodes	4 nodes	8 nodes	16 nodes
1 process per node	00:00:07	00:00:03	00:00:02	00:00:01	00:00:00
2 processes per node	00:00:03	00:00:02	00:00:01	00:00:00	00:00:00
4 processes per node	00:00:02	00:00:01	00:00:00	00:00:00	00:00:00
8 processes per node	00:00:01	00:00:00	00:00:00	00:00:00	00:00:00
16 processes per node	00:00:01	00:00:00	00:00:00	00:00:00	00:00:00
32 processes per node	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00
(b) Mesh resolution $N \times N = 2048 \times 2048$, system dimension 4194304					
	1 node	2 nodes	4 nodes	8 nodes	16 nodes
1 process per node	00:01:25	00:00:38	00:00:16	00:00:07	00:00:04
2 processes per node	00:00:38	00:00:14	00:00:07	00:00:03	00:00:02
4 processes per node	00:00:20	00:00:08	00:00:04	00:00:02	00:00:01
8 processes per node	00:00:11	00:00:05	00:00:02	00:00:01	00:00:00
16 processes per node	00:00:06	00:00:02	00:00:01	00:00:00	00:00:00
32 processes per node	00:00:05	00:00:01	00:00:00	00:00:00	00:00:00
(c) Mesh resolution $N \times N = 4096 \times 4096$, system dimension 16777216					
	1 node	2 nodes	4 nodes	8 nodes	16 nodes
1 process per node	00:13:04	00:06:16	00:02:57	00:01:18	00:00:33
2 processes per node	00:06:15	00:02:57	00:01:19	00:00:32	00:00:14
4 processes per node	00:03:12	00:01:32	00:00:42	00:00:17	00:00:08
8 processes per node	00:01:41	00:00:49	00:00:22	00:00:10	00:00:04
16 processes per node	00:01:08	00:00:33	00:00:14	00:00:05	00:00:02
32 processes per node	00:01:00	00:00:29	00:00:10	00:00:03	00:00:01
(d) Mesh resolution $N \times N = 8192 \times 8192$, system dimension 67108864					
	1 node	2 nodes	4 nodes	8 nodes	16 nodes
1 process per node	01:47:42	00:53:44	00:26:35	00:13:07	00:06:10
2 processes per node	00:54:12	00:26:43	00:13:04	00:06:09	00:02:44
4 processes per node	00:27:42	00:13:41	00:06:41	00:03:12	00:01:28
8 processes per node	00:14:38	00:07:14	00:03:31	00:01:43	00:00:48
16 processes per node	00:09:49	00:04:53	00:02:23	00:01:09	00:00:30
32 processes per node	00:08:36	00:04:14	00:02:06	00:01:02	00:00:21
(e) Mesh resolution $N \times N = 16384 \times 16384$, system dimension 268435456					
	1 node	2 nodes	4 nodes	8 nodes	16 nodes
1 process per node	14:29:41	07:17:16	03:37:51	01:49:40	00:54:18
2 processes per node	07:18:18	03:39:57	01:49:43	00:54:02	00:26:39
4 processes per node	03:46:30	01:54:28	00:56:30	00:28:15	00:13:44
8 processes per node	01:58:00	00:59:28	00:29:40	00:14:43	00:07:15
16 processes per node	01:20:27	00:40:49	00:19:57	00:10:01	00:04:57
32 processes per node	01:10:40	00:35:53	00:17:37	00:08:40	00:04:16

Table 4.2: Wall clock time in HH:MM:SS on Skylake nodes on the Stempede2 cluster using OmniPath network using 1 thread per hardware core with `OMP_PLACES=cores and OMP_PROC_BIND=spread` using the Intel compiler 18.0.0 with Intel MPI 18.0.0.

(a) Mesh resolution $N \times N = 1024 \times 1024$, system dimension 1048576	1 node	2 nodes	4 nodes	8 nodes	16 nodes
1 process per node	00:00:07	00:00:03	00:00:02	00:00:01	00:00:00
2 processes per node	00:00:03	00:00:02	00:00:01	00:00:00	00:00:00
4 processes per node	00:00:02	00:00:01	00:00:00	00:00:00	00:00:00
8 processes per node	00:00:01	00:00:00	00:00:00	00:00:00	00:00:00
16 processes per node	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00
32 processes per node	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00
(b) Mesh resolution $N \times N = 2048 \times 2048$, system dimension 4194304	1 node	2 nodes	4 nodes	8 nodes	16 nodes
1 process per node	00:01:26	00:00:37	00:00:15	00:00:07	00:00:03
2 processes per node	00:00:38	00:00:14	00:00:07	00:00:03	00:00:02
4 processes per node	00:00:20	00:00:08	00:00:04	00:00:02	00:00:01
8 processes per node	00:00:10	00:00:04	00:00:02	00:00:01	00:00:00
16 processes per node	00:00:07	00:00:02	00:00:01	00:00:00	00:00:00
32 processes per node	00:00:05	00:00:01	00:00:01	00:00:00	00:00:00
(c) Mesh resolution $N \times N = 4096 \times 4096$, system dimension 16777216	1 node	2 nodes	4 nodes	8 nodes	16 nodes
1 process per node	00:13:01	00:06:18	00:02:58	00:01:18	00:00:34
2 processes per node	00:06:17	00:02:56	00:01:18	00:00:33	00:00:15
4 processes per node	00:03:07	00:01:31	00:00:42	00:00:17	00:00:08
8 processes per node	00:01:41	00:00:49	00:00:22	00:00:09	00:00:04
16 processes per node	00:01:09	00:00:33	00:00:14	00:00:05	00:00:02
32 processes per node	00:01:00	00:00:31	00:00:10	00:00:03	00:00:01
(d) Mesh resolution $N \times N = 8192 \times 8192$, system dimension 67108864	1 node	2 nodes	4 nodes	8 nodes	16 nodes
1 process per node	01:45:43	00:53:10	00:26:30	00:13:06	00:06:09
2 processes per node	00:55:12	00:27:06	00:13:02	00:06:11	00:02:45
4 processes per node	00:27:13	00:13:44	00:06:41	00:03:12	00:01:28
8 processes per node	00:14:39	00:07:15	00:03:32	00:01:42	00:00:48
16 processes per node	00:09:57	00:04:53	00:02:25	00:01:09	00:00:30
32 processes per node	00:08:55	00:04:16	00:02:06	00:01:02	00:00:21

Table 4.3: Wall clock time in HH:MM:SS on Skylake nodes on the Stempede2 cluster using OmniPath network using 1 thread per hardware core with `KMP_AFFINITY=scatter` using the Intel compiler 18.0.0 with Intel MPI 18.0.0.

(a) Mesh resolution $N \times N = 1024 \times 1024$, system dimension 1048576					
	1 node	2 nodes	4 nodes	8 nodes	16 nodes
1 process per node	00:00:07	00:00:03	00:00:02	00:00:01	00:00:00
2 processes per node	00:00:03	00:00:02	00:00:01	00:00:00	00:00:00
4 processes per node	00:00:02	00:00:01	00:00:00	00:00:00	00:00:00
8 processes per node	00:00:01	00:00:00	00:00:00	00:00:00	00:00:00
16 processes per node	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00
32 processes per node	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00
(b) Mesh resolution $N \times N = 2048 \times 2048$, system dimension 4194304					
	1 node	2 nodes	4 nodes	8 nodes	16 nodes
1 process per node	00:01:22	00:00:36	00:00:15	00:00:07	00:00:03
2 processes per node	00:00:38	00:00:15	00:00:07	00:00:03	00:00:02
4 processes per node	00:00:20	00:00:08	00:00:04	00:00:02	00:00:01
8 processes per node	00:00:10	00:00:05	00:00:02	00:00:01	00:00:00
16 processes per node	00:00:07	00:00:02	00:00:01	00:00:00	00:00:00
32 processes per node	00:00:04	00:00:01	00:00:01	00:00:00	00:00:00
(c) Mesh resolution $N \times N = 4096 \times 4096$, system dimension 16777216					
	1 node	2 nodes	4 nodes	8 nodes	16 nodes
1 process per node	00:12:40	00:06:19	00:02:58	00:01:18	00:00:34
2 processes per node	00:06:19	00:02:53	00:01:19	00:00:32	00:00:14
4 processes per node	00:03:11	00:01:31	00:00:42	00:00:18	00:00:08
8 processes per node	00:01:41	00:00:49	00:00:22	00:00:10	00:00:05
16 processes per node	00:01:08	00:00:33	00:00:14	00:00:05	00:00:02
32 processes per node	00:01:00	00:00:29	00:00:10	00:00:02	00:00:01
(d) Mesh resolution $N \times N = 8192 \times 8192$, system dimension 67108864					
	1 node	2 nodes	4 nodes	8 nodes	16 nodes
1 process per node	01:48:29	00:53:51	00:26:56	00:13:00	00:06:09
2 processes per node	00:53:54	00:26:50	00:13:07	00:06:10	00:02:45
4 processes per node	00:27:27	00:13:47	00:06:42	00:03:11	00:01:28
8 processes per node	00:14:30	00:07:11	00:03:30	00:01:45	00:00:48
16 processes per node	00:10:06	00:04:55	00:02:25	00:01:08	00:00:30
32 processes per node	00:08:45	00:04:25	00:02:05	00:01:02	00:00:21
(e) Mesh resolution $N \times N = 16384 \times 16384$, system dimension 268435456					
	1 node	2 nodes	4 nodes	8 nodes	16 nodes
1 process per node	14:17:25	07:16:00	03:39:24	01:49:35	00:54:49
2 processes per node	07:17:33	03:38:39	01:50:13	00:54:26	00:26:49
4 processes per node	03:43:21	01:52:55	00:56:06	00:28:11	00:13:51
8 processes per node	01:58:23	00:59:44	00:29:49	00:14:50	00:07:17
16 processes per node	01:20:31	00:40:30	00:20:26	00:10:15	00:04:59
32 processes per node	01:12:25	00:36:17	00:18:27	00:08:58	00:04:27

5 Comparisons

Table 5.1 contains a summary of the results obtained on the Stampede2 cluster as well as a comparison to HPCF clusters. The table reports results for the historical mesh resolution of $N = 4096$, which was the largest resolution that could be solved on kali in 2003 (using the extended memory of 4 GB on the storage node). Also, to maintain backward comparisons, this table is restricted to 32 nodes, since the old clusters kali and hpc had fewer nodes than maya. The first row of the table contains the results for cluster kali. This cluster was a 33-node distributed-memory cluster with 32 compute nodes including a storage node (with extended memory of 4 GB), containing the 0.5 TB central storage, each with two (single-core) Intel Xeon 2.0 GHz processors and 1 GB of memory, connected by a Myrinet interconnect, plus 1 combined user/management node. Note that for the case of all cores on 1 node, that is, for the case of both (single-core) CPUs used simultaneously, the performance was worse than for 1 CPU and hence the results were not recorded at the time. The second row of the table contains results for the cluster hpc which was a 35-node distributed-memory cluster with 33 compute nodes plus 1 development and 1 combined user/management node, each equipped with two dual-core AMD Opteron processors and at least 13 GB of memory, connected by a DDR InfiniBand network and with an InfiniBand-accessible 14 TB parallel file system. The third row contains results for the cluster tara which was an 86-node distributed-memory cluster with two quad-core Intel Nehalem processors and 24 GB per node, a QDR InfiniBand interconnect, and 160 TB central storage. This cluster is now part of the cluster maya as maya 2009, and its QDR InfiniBand network extends to the 2013 portion of maya. The fourth row of the table contains results for maya 2009, which recomputes the results from tara using the default compiler and MPI implementation in 2013. The fifth row of the table contains results for the DDR InfiniBand connected portion maya 2010, and the sixth row contains results for the QDR InfiniBand connected portion maya 2013. The seventh row contains results for the Omni-Path network connected Skylake portion of Stampede2 with each compute node having two Intel Xeon Platinum 8160 Skylake CPUs with 24 cores at 2.1 GHz and 192 GB of memory and `OMP_PLACES=cores` and `OMP_PROC_BIND=close` parameters in the slurm script.

On the cluster kali from 2003, we observe a factor of approximately 30 speedup by increasing the number of nodes from 1 to 32. However by using both cores on each node we only see a factor of approximately 25 speedup. We do not observe the expected 64 factor speedup, since both CPUs on the node share a bus connection to the memory, which leads to contention in essentially synchronized algorithms like Krylov subspace methods. Hence, it is actually faster to leave the second CPU idling rather than to use both [1]. Note that there are four cores on each node of cluster hpc from 2008, compared to just two on the cluster kali, since the CPUs are dual-core. We observe approximately fourfold speedup that we would expect by running it on four cores rather than one. By running on 32 nodes with one core per node we observe the expected speedup of approximately 32; more in detail, the speedup is slightly better than optimal, which is explained by the smaller portions of the subdivided problem on each node fitting better into the cache of the processors. We see this for the first time here, but it is a typical effect in strong performance studies, in which a problem that already fits on one node is divided into smaller and smaller pieces as the number of nodes grows. Finally, by using all cores on 32 nodes we observe a speedup of 76.01, less than the optimal speedup of 128 [4]. On the cluster tara from 2009, we observe a less than optimal speedup of approximately 5 by running on all 8 cores rather than on one, caused by the cores of a CPU competing for memory access. By running on 32 nodes with one core per node we observe a speedup of approximately 30. Finally, by using all 8 cores on 32 nodes we observe a speedup of 208, less than the optimal speedup of 256 [9]. On maya 2009, we observe that by running on all 8 cores on a single node rather than 1 core there is a speedup of approximately 3 rather than the optimal speedup of 8. By running on 32 nodes with one core per node we observe a speedup of approximately 29 of the possible 32, which is very good. Finally, by using all 8 cores on 32 nodes we observe a speedup of 128, half of the

Table 5.1: Runtimes (speedup) for $N = 4096$ on the clusters kali, hpc, tara, maya, and Stampede2.

Cluster (year)	serial (1 core) time	1 node all cores time (speedup)	32 nodes 1 core per node time (speedup)	32 nodes all cores time (speedup)
kali (2003) [1]	02:00:49	N/A (N/A)	00:04:05 (29.59)	00:04:49 (25.08)
hpc (2008) [4]	01:51:29	00:32:37 (3.42)	00:03:23 (32.95)	00:01:28 (76.01)
tara (2009) [9]	00:31:16	00:06:39 (4.70)	00:01:05 (28.86)	00:00:09 (208.44)
maya 2009 [8]	00:17:05	00:05:55 (2.89)	00:00:35 (29.29)	00:00:08 (128.13)
maya 2010 [8]	00:17:00	00:05:48 (2.93)	00:00:34 (30.00)	00:00:06 (170.00)
maya 2013 [8]	00:12:26	00:02:44 (4.55)	00:00:15 (49.07)	00:00:12 (62.17)
Stampede2 (2018)	00:13:04	00:01:00 (13.07)	00:00:16 (49.00)	00:00:01 (784.00)

optimal speedup of 256 [8]. On maya 2010, we observe that by running on all 8 cores on a single node rather than 1 core there is a speedup of approximately 3 rather than the optimal speedup of 8. By running on 32 nodes with one core per node we observe a speedup of approximately 30. When combining the use of all cores with the use of 32 nodes, we observe a speedup of 170, short of the optimal speedup of 256 [8]. On maya 2013, we observe that by running on all 16 cores on a single node rather than on one core there is a speedup of approximately 5 rather than the expected speedup of 16. We observe a greater than optimal speedup of 49.07 by running on 32 nodes with one process per node; this is caused by the relatively small problem fitting into cache after dividing it onto 32 nodes, together with the quality of the QDR InfiniBand interconnect. As observed in [8, Table 4.1], using 8 processes per node on 32 nodes actually allows for a run time of 00:00:04 (speedup 186), which is faster than the 16 processes per node ("all cores" of 32 nodes) in Table 5.1. Thus, the speedup of 62.17 reported here is conservative [8].

Finally, the seventh and last row shows the Stampede2 data. The data for 1 node is from Table 4.1, which uses 32 cores in the column for "all cores". The results from Table 4.1 are extended for $N = 4096$ to 32 nodes in Table 5.1. We can now observe that in the past, the core per core performance improved each time. But this is not true any more now, as serial code on a single core in the Skylake processor is not faster than in the Ivy Bridge CPUs from 2013. This demonstrates that using multi-threading and/or MPI on a shared-memory node is absolutely vital to achieving optimal and scalable performance today and in the future. The possibility of this can be seen by the 13-fold speedup of the code that is possible when using "all cores" of 1 node, which actually used 32 cores out of the available 48 cores; see Table 4.1. In fact, when comparing the "all cores" results of Stampede2 to the previous rows, we see that they are the fastest in each column and a significant improvement over all past results. We also see that the Omni-Path network on Stampede2 provides for excellent speedup, as the 32-node results have better than linear speedup, both comparing the 1-core and the 32-core cases. For instance, the speedup of 784 indicates that much faster speed than the serial run using 1 core, which is an excellent speedup compared to the nominally expected using 32 nodes with 32 cores for a total of 1024 cores, but when comparing to the 32 cores ("all cores" column) on 1 node, the speedup is 61, which is much better than the nominally expected 32. We note that these better than optimal results typically result from the problem being divided into such small chunks on each MPI process that they fit in cache there, but the multi-node performance is still a testament to the low latency of the network.

6 Performance Studies on Skylake Nodes with Omni-Path, InfiniBand, and Ethernet Interconnects on Dell clusters

This section describes the parallel performance studies for the solution of the test problem on the Skylake nodes of Dell's zenith and rattler cluster using different networks. Dell has two different cluster, zenith and rattler. Some of the nodes in the zenith cluster are connected by Omni-Path (OPA) and some are connected by Ethernet. The nodes of the rattler cluster are connected by InfiniBand. We use up to 16 nodes, each with two Intel Xeon Gold 6148 Skylake CPUs with 20 cores at 2.4 GHz and 192 GB of memory. Each CPU has 20 cores making it a total of 40 cores per node; hyperthreading is not enabled. There is 32 kB L1 cache per core, 1 MB L2 cache per core, and 27.5 MB L3 cache per socket. Some nodes in zenith are connected by a 100 Gbps Intel Omni-Path (OPA) interconnect and some are connected by a 10 Gbps Ethernet. The nodes in rattler cluster are connected by a Mellanox EDR InfiniBand.

The results in this section use the Intel compiler 18.0.0 and Intel MPI 18.0.0.

The SLURM submission script uses the `mpirun` command to start the job. The number of nodes are controlled by the `--nodes` option in the SLURM submission script, and the number of processes per node by the `--ntasks-per-node` option. Each node that is used is dedicated to the job with remaining cores idling, if not all of them are used, using the `--exclusive` flag.

All studies in this section use the `OMP_PLACES=cores` and `OMP_PROC_BIND=close` environment variables, like they were used in Table 4.1. The studies also use the same mesh resolutions, numbers of nodes, and numbers of processes per node as in that table. We observe that the run times and the code performance and scalability are essentially the same as in Table 4.1. This indicates that for this test problem, which is memory-bound, all three networks allow for equivalent performance.

Table 6.1: Wall clock time in HH:MM:SS on Skylake nodes on the Dell zenith cluster using the Omni-Path network using 1 thread per hardware core with `OMP_PLACES=cores` and `OMP_PROC_BIND=close` using the Intel compiler 18.0.0 with Intel MPI 18.0.0.

(a) Mesh resolution $N \times N = 1024 \times 1024$, system dimension 1048576

	1 node	2 nodes	4 nodes	8 nodes	16 nodes
1 process per node	00:00:08	00:00:03	00:00:02	00:00:01	00:00:00
2 processes per node	00:00:03	00:00:02	00:00:01	00:00:00	00:00:00
4 processes per node	00:00:02	00:00:01	00:00:00	00:00:00	00:00:00
8 processes per node	00:00:01	00:00:00	00:00:00	00:00:00	00:00:00
16 processes per node	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00
32 processes per node	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00

(b) Mesh resolution $N \times N = 2048 \times 2048$, system dimension 4194304

	1 node	2 nodes	4 nodes	8 nodes	16 nodes
1 process per node	00:01:26	00:00:39	00:00:16	00:00:07	00:00:03
2 processes per node	00:00:40	00:00:16	00:00:07	00:00:03	00:00:02
4 processes per node	00:00:21	00:00:08	00:00:04	00:00:02	00:00:01
8 processes per node	00:00:12	00:00:04	00:00:02	00:00:01	00:00:01
16 processes per node	00:00:09	00:00:03	00:00:01	00:00:01	00:00:00
32 processes per node	00:00:05	00:00:01	00:00:00	00:00:00	00:00:00

(c) Mesh resolution $N \times N = 4096 \times 4096$, system dimension 16777216

	1 node	2 nodes	4 nodes	8 nodes	16 nodes
1 process per node	00:12:40	00:06:12	00:02:58	00:01:22	00:00:35
2 processes per node	00:06:15	00:02:59	00:01:22	00:00:34	00:00:14
4 processes per node	00:03:13	00:01:33	00:00:43	00:00:18	00:00:08
8 processes per node	00:01:43	00:00:50	00:00:24	00:00:10	00:00:04
16 processes per node	00:01:24	00:00:42	00:00:19	00:00:06	00:00:02
32 processes per node	00:01:06	00:00:30	00:00:12	00:00:03	00:00:01

(d) Mesh resolution $N \times N = 8192 \times 8192$, system dimension 67108864

	1 node	2 nodes	4 nodes	8 nodes	16 nodes
1 process per node	01:45:21	00:52:40	00:26:07	00:12:53	00:06:07
2 processes per node	00:52:52	00:26:11	00:12:51	00:06:11	00:02:52
4 processes per node	00:27:14	00:13:35	00:06:41	00:03:14	00:01:31
8 processes per node	00:14:30	00:07:21	00:03:35	00:01:46	00:00:51
16 processes per node	00:11:42	00:05:45	00:02:57	00:01:25	00:00:42
32 processes per node	00:08:46	00:04:20	00:02:14	00:01:08	00:00:26

(e) Mesh resolution $N \times N = 16384 \times 16384$, system dimension 268435456

	1 node	2 nodes	4 nodes	8 nodes	16 nodes
1 process per node	14:22:52	07:06:16	03:34:39	01:46:57	00:52:53
2 processes per node	07:07:47	03:35:03	01:46:38	00:53:13	00:26:07
4 processes per node	03:41:04	01:50:33	00:55:28	00:27:33	00:13:37
8 processes per node	01:57:35	00:59:04	00:29:34	00:14:42	00:07:23
16 processes per node	01:31:52	00:46:42	00:23:31	00:12:15	00:05:57
32 processes per node	01:10:24	00:36:08	00:18:25	00:09:01	00:04:36

Table 6.2: Wall clock time in HH:MM:SS on Skylake nodes on the Dell rattler cluster using the InfiniBand network using 1 thread per hardware core with `OMP_PLACES=cores` and `OMP_PROC_BIND=close` using the Intel compiler 18.0.0 with Intel MPI 18.0.0.

(a) Mesh resolution $N \times N = 1024 \times 1024$, system dimension 1048576					
	1 node	2 nodes	4 nodes	8 nodes	16 nodes
1 process per node	00:00:07	00:00:03	00:00:02	00:00:01	00:00:00
2 processes per node	00:00:03	00:00:02	00:00:01	00:00:00	00:00:00
4 processes per node	00:00:02	00:00:01	00:00:00	00:00:00	00:00:00
8 processes per node	00:00:01	00:00:00	00:00:00	00:00:00	00:00:00
16 processes per node	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00
32 processes per node	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00

(b) Mesh resolution $N \times N = 2048 \times 2048$, system dimension 4194304					
	1 node	2 nodes	4 nodes	8 nodes	16 nodes
1 process per node	00:01:28	00:00:39	00:00:16	00:00:06	00:00:04
2 processes per node	00:00:40	00:00:16	00:00:06	00:00:03	00:00:02
4 processes per node	00:00:21	00:00:09	00:00:03	00:00:02	00:00:01
8 processes per node	00:00:12	00:00:04	00:00:02	00:00:01	00:00:00
16 processes per node	00:00:08	00:00:03	00:00:01	00:00:00	00:00:00
32 processes per node	00:00:06	00:00:01	00:00:01	00:00:00	00:00:00

(c) Mesh resolution $N \times N = 4096 \times 4096$, system dimension 16777216					
	1 node	2 nodes	4 nodes	8 nodes	16 nodes
1 process per node	00:12:51	00:06:17	00:03:00	00:01:22	00:00:35
2 processes per node	00:06:17	00:03:00	00:01:22	00:00:36	00:00:17
4 processes per node	00:03:16	00:01:34	00:00:44	00:00:19	00:00:08
8 processes per node	00:01:44	00:00:50	00:00:23	00:00:09	00:00:04
16 processes per node	00:01:30	00:00:42	00:00:21	00:00:06	00:00:02
32 processes per node	00:01:08	00:00:33	00:00:17	00:00:02	00:00:01

(d) Mesh resolution $N \times N = 8192 \times 8192$, system dimension 67108864					
	1 node	2 nodes	4 nodes	8 nodes	16 nodes
1 process per node	01:46:01	00:53:02	00:26:24	00:12:54	00:06:46
2 processes per node	00:53:11	00:26:25	00:12:56	00:06:10	00:03:10
4 processes per node	00:27:40	00:13:45	00:06:43	00:03:14	00:01:38
8 processes per node	00:14:48	00:07:24	00:03:39	00:01:46	00:00:53
16 processes per node	00:12:02	00:06:01	00:03:01	00:01:29	00:00:42
32 processes per node	00:09:22	00:04:46	00:02:56	00:01:09	00:00:27

Table 6.3: Wall clock time in HH:MM:SS on Skylake nodes on the Dell zenith cluster using the Ethernet network using 1 thread per hardware core with `OMP_PLACES=cores` and `OMP_PROC_BIND=close` using the Intel compiler 18.0.0 with Intel MPI 18.0.0.

(a) Mesh resolution $N \times N = 1024 \times 1024$, system dimension 1048576					
	1 node	2 nodes	4 nodes	8 nodes	16 nodes
1 process per node	00:00:07	00:00:03	00:00:02	00:00:01	00:00:01
2 processes per node	00:00:03	00:00:02	00:00:01	00:00:01	00:00:01
4 processes per node	00:00:02	00:00:01	00:00:01	00:00:01	00:00:01
8 processes per node	00:00:01	00:00:01	00:00:00	00:00:01	00:00:01
16 processes per node	00:00:00	00:00:00	00:00:00	00:00:01	00:00:01
32 processes per node	00:00:00	00:00:00	00:00:00	00:00:01	00:00:01
(b) Mesh resolution $N \times N = 2048 \times 2048$, system dimension 4194304					
	1 node	2 nodes	4 nodes	8 nodes	16 nodes
1 process per node	00:01:27	00:00:40	00:00:16	00:00:07	00:00:04
2 processes per node	00:00:39	00:00:15	00:00:07	00:00:04	00:00:03
4 processes per node	00:00:21	00:00:09	00:00:04	00:00:03	00:00:02
8 processes per node	00:00:11	00:00:05	00:00:02	00:00:02	00:00:02
16 processes per node	00:00:09	00:00:03	00:00:02	00:00:01	00:00:01
32 processes per node	00:00:05	00:00:02	00:00:01	00:00:01	00:00:02
(c) Mesh resolution $N \times N = 4096 \times 4096$, system dimension 16777216					
	1 node	2 nodes	4 nodes	8 nodes	16 nodes
1 process per node	00:12:42	00:06:14	00:03:00	00:01:24	00:00:37
2 processes per node	00:06:13	00:03:00	00:01:24	00:00:38	00:00:16
4 processes per node	00:03:13	00:01:35	00:00:46	00:00:19	00:00:10
8 processes per node	00:01:44	00:00:51	00:00:25	00:00:11	00:00:06
16 processes per node	00:01:25	00:00:44	00:00:21	00:00:07	00:00:05
32 processes per node	00:01:02	00:00:34	00:00:14	00:00:04	00:00:04
(d) Mesh resolution $N \times N = 8192 \times 8192$, system dimension 67108864					
	1 node	2 nodes	4 nodes	8 nodes	16 nodes
1 process per node	01:45:23	00:52:52	00:26:07	00:12:54	00:06:14
2 processes per node	00:52:39	00:26:09	00:12:54	00:06:18	00:03:01
4 processes per node	00:27:11	00:13:35	00:06:47	00:03:22	00:01:37
8 processes per node	00:14:20	00:07:16	00:03:41	00:01:49	00:00:54
16 processes per node	00:11:25	00:05:43	00:02:59	00:01:29	00:00:48
32 processes per node	00:08:46	00:04:28	00:02:18	00:01:09	00:00:31
(e) Mesh resolution $N \times N = 16384 \times 16384$, system dimension 268435456					
	1 node	2 nodes	4 nodes	8 nodes	16 nodes
1 process per node	14:11:13	07:08:15	03:36:56	01:46:27	00:52:51
2 processes per node	07:07:23	03:33:57	01:47:02	00:53:08	00:26:39
4 processes per node	03:40:16	01:50:24	00:55:12	00:27:40	00:14:37
8 processes per node	01:56:45	00:58:51	00:29:32	00:15:01	00:07:51
16 processes per node	01:32:07	00:45:58	00:23:13	00:11:41	00:06:04
32 processes per node	01:10:05	00:35:18	00:18:04	00:09:07	00:04:49

7 Performance Studies on Skylake Nodes with Omni-Path Interconnect on Stampede2 Using Hybrid MPI+OpenMP Code

This section describes the parallel performance studies for the solution of the test problem on the Skylake portion of Stampede2 using a hybrid MPI+OpenMP implementation of the code. In this implementation, OpenMP multi-threading is used to spawn several software threads within each MPI process. This OpenMPI multi-threading is implemented using `#pragma` lines for performance-critical for-loops and other parallel portions of the code. The studies in this section use the environment variables `OMP_PLACES=cores` and `OMP_PROC_BIND=close`.

Table 7.1 collects the results of the performance study using 1 thread per hardware core and 32 total threads per node. That is, 32 hardware cores per node are in use in each entry of the table, with the specified number of MPI processes and OpenMP threads per node, whose multiple is 32 in each case. With respect to the processes and threads per node, this study constitues a weak scalability study, since the run times should nominally be the same, independent of how the software threads access the same number of hardware cores. With respect to the increasing number of nodes used, the study is a strong scalability study, since the amount of resources used increases and the run times should nominally halve from column to column with the doubling of nodes used. Consider first concretely the Table 7.1 (b) for the 8192×8192 mesh. In the first result column for 1 node used, we observe essentially identical run times for all true hybrid cases, that is, those with at least 2 MPI processes and at least 2 OpenMP threads, which is the nominally expected result. On the one hand, using shared-memory multi-threading only (1 MPI process, 32 threads), we see a run time that is about 50% increased, which results from inefficiencies in accessing hardware cores on both CPUs without the benefit of MPI parallelism. On the other hand, using only MPI parallelism (32 MPI processes, 1 thread), the run time is slightly better than for any hybrid run, making it certainly no disadvantage to use MPI-only code for this implementation of the test problem. When reading results along each row of the table, run times approximately halve whenever the number of nodes is doubled, indicating excellent strong scalability of the code across the high-performance network. In fact, from 8 nodes to 16 nodes as well as from 16 nodes to 32 nodes, each row exhibits better than nominally optimal speedup, indicating that the problem size has descreased sufficiently after these sub-divisions to allow for more efficient use of cache than on fewer nodes. The same oservations hold for Table 7.1 (a) for the 4096×4096 mesh, except that the better than nominally optimal speedup is achieved from 2 nodes to 4 nodes as well as from 4 nodes to 8 nodes, after which nearly no improvement is observed. Note that the times for 32 MPI processes and 1 OpenMP thread per node in the last row of each sub-table rerun and confirm the results with 32 processes per node of MPI-only code in Table 4.1.

Table 7.2 collects the results of the performance study using 2 threads per hardware core and 64 total threads per node, which is possible here, since hyperthreading is enabled on the Skylake nodes of Stampede2. That is, each entry of Table 7.2 uses 64 hardware threads, with the specified number of MPI processes and OpenMP threads per node, whose multiple is 64 in each case. We note that the distribution of the 64 software threads to the 48 available hardware cores is not fully determined, since it is possible that some cores only have one thread on them, while others might still be idling; we did not investigate this detail more. It is notable that the runs with multithreading only in the first results row of each sub-table in Table 7.2 are faster than the corresponding row in Table 7.1, which might indicate a benefit of hyperthreading for multithreading-only code. But in all other cases, it is in fact slower for this implementation of the test problem to use hyperthreading, since run times in the other rows in Table 7.2 are slower than any entry for the corresponding columns in Table 7.1. This demonstrates that at least for implementation of the test problem, code using MPI (whether MPI-only or in hybrid MPI+OpenMP code) is fastest, and one should not use more software threads than hardware cores are available.

Table 7.1: Wall clock time in HH:MM:SS on Skylake nodes on the Stampede2 cluster using 1 thread per hardware core with `OMP_PLACES=cores` and `OMP_PROC_BIND=close` using the Intel compiler 18.0.0 with Intel MPI 18.0.0.

(a) Mesh resolution $N \times N = 4096 \times 4096$, system dimension 16777216							
Processes and threads per node		1 node	2 nodes	4 nodes	8 nodes	16 nodes	32 nodes
1 MPI process,	32 OpenMP threads	00:01:38	00:00:48	00:00:19	00:00:04	00:00:01	00:00:01
2 MPI processes,	16 OpenMP threads	00:01:04	00:00:31	00:00:10	00:00:03	00:00:01	00:00:01
4 MPI processes,	8 OpenMP threads	00:01:01	00:00:31	00:00:10	00:00:02	00:00:01	00:00:01
8 MPI processes,	4 OpenMP threads	00:01:01	00:00:29	00:00:10	00:00:03	00:00:01	00:00:01
16 MPI processes,	2 OpenMP threads	00:01:02	00:00:28	00:00:10	00:00:02	00:00:02	00:00:01
32 MPI processes,	1 OpenMP thread	00:00:59	00:00:28	00:00:10	00:00:03	00:00:01	00:00:01
(b) Mesh resolution $N \times N = 8192 \times 8192$, system dimension 67108864							
Processes and threads per node		1 node	2 nodes	4 nodes	8 nodes	16 nodes	32 nodes
1 MPI process,	32 OpenMP threads	00:13:23	00:06:41	00:03:26	00:01:47	00:00:41	00:00:09
2 MPI processes,	16 OpenMP threads	00:08:48	00:04:29	00:02:11	00:01:07	00:00:23	00:00:07
4 MPI processes,	8 OpenMP threads	00:08:56	00:04:19	00:02:10	00:01:04	00:00:22	00:00:06
8 MPI processes,	4 OpenMP threads	00:08:47	00:04:18	00:02:07	00:01:02	00:00:22	00:00:06
16 MPI processes,	2 OpenMP threads	00:08:48	00:04:48	00:02:14	00:01:02	00:00:21	00:00:06
32 MPI processes,	1 OpenMP thread	00:08:39	00:04:20	00:02:04	00:00:59	00:00:22	00:00:06

Table 7.2: Wall clock time in HH:MM:SS on Skylake nodes on the Stampede2 cluster using 2 threads per hardware core by hyperthreading with `OMP_PLACES=cores` and `OMP_PROC_BIND=close` using the Intel compiler 18.0.0 with Intel MPI 18.0.0.

(a) Mesh resolution $N \times N = 4096 \times 4096$, system dimension 16777216							
Processes and threads per node		1 node	2 nodes	4 nodes	8 nodes	16 nodes	32 nodes
1 MPI process,	64 OpenMP threads	00:01:05	00:00:29	00:00:08	00:00:03	00:00:01	00:00:01
2 MPI processes,	32 OpenMP threads	00:01:06	00:00:29	00:00:08	00:00:03	00:00:01	00:00:01
4 MPI processes,	16 OpenMP threads	00:01:08	00:00:30	00:00:08	00:00:02	00:00:01	00:00:01
8 MPI processes,	8 OpenMP threads	00:01:02	00:00:29	00:00:08	00:00:02	00:00:01	00:00:01
16 MPI processes,	4 OpenMP threads	00:01:03	00:00:28	00:00:08	00:00:02	00:00:01	00:00:01
32 MPI processes,	2 OpenMP threads	00:01:02	00:00:28	00:00:08	00:00:02	00:00:01	00:00:01
64 MPI processes,	1 OpenMP thread	00:01:05	00:00:30	00:00:09	00:00:02	00:00:02	00:00:01
(b) Mesh resolution $N \times N = 8192 \times 8192$, system dimension 67108864							
Processes and threads per node		1 node	2 nodes	4 nodes	8 nodes	16 nodes	32 nodes
1 MPI process,	64 OpenMP threads	00:09:22	00:04:39	00:02:21	00:01:05	00:00:19	00:00:06
2 MPI processes,	32 OpenMP threads	00:09:06	00:04:36	00:02:27	00:01:04	00:00:19	00:00:07
4 MPI processes,	16 OpenMP threads	00:09:03	00:05:02	00:02:24	00:01:01	00:00:18	00:00:07
8 MPI processes,	8 OpenMP threads	00:09:01	00:04:32	00:02:12	00:01:02	00:00:17	00:00:08
16 MPI processes,	4 OpenMP threads	00:09:16	00:04:41	00:02:08	00:01:02	00:00:17	00:00:06
32 MPI processes,	2 OpenMP threads	00:09:11	00:04:27	00:02:09	00:01:00	00:00:18	00:00:06
64 MPI processes,	1 OpenMP thread	00:09:22	00:04:41	00:02:18	00:01:05	00:00:20	00:00:08

Acknowledgments

This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI–1053575 [11]. We acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing the HPC resources Stampede2. The hardware in the UMBC High Performance Computing Facility (HPCF) is supported by the U.S. National Science Foundation through the MRI program (grant nos. CNS–0821258, CNS–1228778, and OAC–1726023) and the SCREMS program (grant no. DMS–0821311), with additional substantial support from the University of Maryland, Baltimore County (UMBC). See hpcf.umbc.edu for more information on HPCF and the projects using its resources. The first author was supported by UMBC as HPCF RA.

References

[1] Kevin P. Allen. Efficient parallel computing for solving linear systems of equations. *UMBC Review: Journal of Undergraduate Research and Creative Works*, vol. 5, pp. 8–17, 2004.

[2] Dietrich Braess. *Finite Elements*. Cambridge University Press, third edition, 2007.

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

[4] Matthias K. Gobbert. Parallel performance studies for an elliptic test problem. Technical Report HPCF–2008–1, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2008.

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

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

[7] Samuel Khuvis and Matthias K. Gobbert. Parallel performance studies for an elliptic test problem on maya 2013. Technical Report HPCF–2014–6, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2014.

[8] Samuel Khuvis and Matthias K. Gobbert. Parallel performance studies for an elliptic test problem on the cluster maya. Technical Report HPCF–2015–6, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2015.

[9] Andrew M. Raim and Matthias K. Gobbert. Parallel performance studies for an elliptic test problem on the cluster tara. Technical Report HPCF–2010–2, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2010.

[10] Hafez Tari and Matthias K. Gobbert. A comparative study of the parallel performance of the blocking and non-blocking MPI communication commands on an elliptic test problem on the cluster tara. Technical Report HPCF–2010–6, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2010.

[11] John Towns, Timothy Cockerill, Maytal Dahan, Ian Foster, Kelly Gaither, Andrew Grimshaw, Victor Hazlewood, Scott Lathrop, Dave Lifka, Gregory D. Peterson, Ralph Roskies, J. Ray Scott, and Nancy Wilkins-Diehr. XSEDE: Accelerating scientific discovery. *Comput. Sci. Eng.*, vol. 16, no. 5, pp. 62–74, 2014.

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