Parallel Performance Studies for an Elliptic Test Problem on the 2018 Portion of the Taki Cluster

Carlos Barajas and Matthias K. Gobbert (gobbert@umbc.edu)

Department of Mathematics and Statistics, University of Maryland, Baltimore County

Technical Report HPCF–2018–18, hpcf.umbc.edu > Publications

Abstract

The new 2018 nodes in the CPU cluster taki in the UMBC High Performance Computing Facility contain two 18-core Intel Skylake CPUs and 384 GB of memory per node, connected by an EDR (Enhanced Data Rate) InfiniBand interconnect. The performance studies use the classical test problem of the 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 results demonstrate excellent scalability when using multiple nodes due to the low latency of the high-performance interconnect and good speedup when using all cores of the multi-core CPUs. Comparisons to results on the 2009 and 2013 nodes bring out that core-per-core performance of serial code improvements have stalled, but that node-per-node performance of parallel code continues to improve due to the larger number of cores available on a node. These observations compel the recommendations that (i) serial code should use the 2009 and 2013 nodes of taki and (ii) parallel code is needed to take full advantage of the 2018 nodes.

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 reported here use the new 2018 portion of the CPU cluster in taki. This portion of the CPU cluster consists of 42 compute nodes with two 18-core Intel Xeon Gold 6140 Skylake CPUs (2.3 GHz clock speed, 24.75 MB L3 cache, 6 memory channels, 140 W power), for a total of 36 cores per node, 384 GB memory (12×32 GB DDR4), and a 120 GB SSD drive. These nodes are connected by a network of four 36-port EDR (Enhanced Data Rate) InfiniBand switches (100 Gb/s bandwidth, 90 ns latency) to a central storage of more than 750 TB. See hpcf.umbc.edu for photos, schematics, and more detailed information, also on the other portions of the taki cluster.

This report uses the same test problem that has been used repeatedly to test cluster performance, including in 2018 [2] on Stampede2 with CPUs of the same generation as in this report, in 2014 [8] and 2015 [9] on maya, in 2010 [11, 12] on tara, in 2008 [5] on hpc, and in 2003 [1] on kali. 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 [4, 6, 7, 13]. 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 multiple nodes due to the low latency of the high-performance interconnect and good speedup when using all cores of the multi-core CPUs. Comparisons to past results bring out that core-per-core performance of serial code improvements have stalled, but that node-per-node performance of parallel code continues to improve due to the larger number of cores available on a node. This justifies usage rules that direct serial runs with moderate memory needs to the 2009 and 2013 portions of the CPU cluster and parallel jobs or jobs with large memory needs to the 2018 portion of the CPU cluster.

2 The Elliptic Test Problem

We consider the classical elliptic test problem of the Poisson equation with homogeneous Dirichlet boundary conditions (see, e.g., [13, Chapter 8])

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

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 $k_i = 1, \ldots, N$, $i = 1, 2$, 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 $S = \text{tridiag}(-1, 4, -1) \in \mathbb{R}^{N \times N}$ 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.

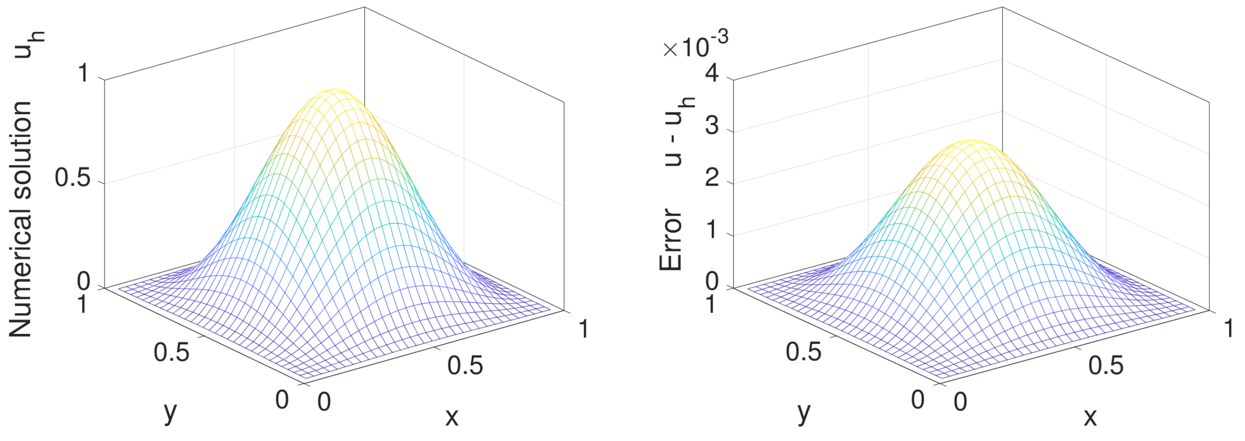
For fine meshes with large N, iterative methods such as the conjugate gradient method are appropriate for solving this linear system. The system matrix A is known to be symmetric positive definite and thus the method is guaranteed to converge for this problem. In a careful implementation, the conjugate 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 processes 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 processes 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` [10]. 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` [10]. For the hybrid MPI+OpenMP code, OpenMP threads are started on each MPI process using a `parallel for` pragma. This parallelizes each MPI process within the shared-memory of a node.

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 $f(x_1,x_2) = (-2\pi^2)\big(\cos(2\pi x_1)\sin^2(\pi x_2) + \sin^2(\pi x_1)\cos(2\pi x_2)\big)$, for which the true analytic solution in closed form $u(x_1,x_2) = \sin^2(\pi x_1)\sin^2(\pi x_2)$ is known. On a mesh with 32×32 interior points and mesh spacing $h = 1/33 \approx 0.030303$, 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 $u(x_1,x_2)$ 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 shown in Figure 3.1 (b). We see that the maximum error occurs at the center of the domain of size about 3×10^{-3} (note the scale on the vertical axis), which compares well to the order of magnitude $h^2 \approx 10^{-3}$ 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



(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 taki 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
32	1,024	3.0128e–03	—	48	00:00:00	0.00	< 1	< 1
64	4,096	7.7811e–04	3.87	96	00:00:00	0.00	< 1	< 1
128	16,384	1.9765e–04	3.94	192	00:00:00	0.01	< 1	< 1
256	65,536	4.9797e–05	3.97	387	00:00:00	0.08	< 1	< 1
512	262,144	1.2494e–05	3.99	783	00:00:01	0.76	< 1	< 1
1024	1,048,576	3.1266e–06	4.00	1,581	00:00:09	8.82	< 1	< 1
2048	4,194,304	7.8019e–07	4.01	3,192	00:01:34	93.54	< 1	< 1
4096	16,777,216	1.9366e–07	4.03	6,452	00:13:32	812.44	< 1	< 1
8192	67,108,864	4.7392e–08	4.09	13,033	01:52:32	6,752.15	2	2.02
16384	268,435,456	1.1647e–08	4.07	26,316	15:07:31	54,451.38	8	8.02
32768	1,073,741,824	2.6004e–09	4.48	53,141	129:18:17	465,496.67	32	32.02
*65536	4,294,967,296	8.9963e–10	2.89	107,261	*02:44:12	*9,852.10	128	*161.53
*131072	17,179,869,184	3.0435e–10	2.96	216,433	*23:01:28	*82,888.17	512	*547.54

*This case uses 32 cores on 32 nodes; the observed memory is the total over all processes.

and observed memory usage in GB for studies performed in serial. More precisely, the serial runs use the parallel code run on one process only, on a dedicated node (no other processes running on the node), and with all parallel 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 cases $N = 65536$ and 131072 require an excessive amount of time needed in serial. Therefore, 32 cores on 32 nodes are used for these cases, 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 [3, 7]. 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 the finite difference error is reduced, which points to the tolerance on the linear solver not being tight enough for 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, 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 $N \leq 32768$, the observed memory usage in units of GB rounds to within 0.02 GB of the predicted usage. This 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. For $N = 65536$ and 131072, the observed memory shows the memory usage totalled over all of the 1024 processes (32 cores on 32 nodes), which leads to a significant duplication of overhead, thus the observed memory usage is quite a bit larger than the predicted one. 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 note that the results in Table 3.1 agree with past results for this problem, where possible, see [2, 9] and the references therein; but they extend to $N = 131072$ for the first time, demonstrating the benefit of the larger memory available on the new cluster. 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 taki 2018 Using MPI-Only Code

This section describes the parallel performance studies for the solution of the test problem on the 2018 portion of the CPU cluster in taki. This portion of the CPU cluster consists of 42 compute nodes with two 18-core Intel Xeon Gold 6140 Skylake CPUs (2.3 GHz clock speed, 24.75 MB L3 cache, 6 memory channels, 140 W power), for a total of 36 cores per node, 384 GB memory (12×32 GB DDR4), and a 120 GB SSD drive. Figure 4.1 shows a schematic of one of the compute nodes, showing also the two Intel UPI connections between the CPUs and indicating that each CPU has 6 memory channels to a DDR4 memory of 32 GB. These compute nodes are connected by a network of four 36-port EDR (Enhanced Data Rate) InfiniBand switches (100 Gb/s bandwidth, 90 ns latency).

The results in this section use the default Intel compiler and Intel MPI, currently version 18.0.3. Together with `-O3 -std=c99 -Wall`, we use the compiler options `-xCORE-AVX2` and `-axCORE-AVX512` to generate a binary for the Intel Skylake CPUs.

The SLURM submission script uses the `srun` command to start the job. The number of nodes are controlled by the `--nodes` option and the number of MPI processes per node by the `--ntasks-per-node` option. For a performance study, each node that is used is dedicated to the job with the remaining cores idling, if not all of them are used, using the `--exclusive` flag. Correspondingly, we request all memory of the node for the job by `--mem=MaxMemPerNode`.

We include the `OMP_PLACES` and `OMP_PROC_BIND` environment variables[1] in the SLURM script. The environment variable `OMP_PLACES=cores` is used to list the cores of the CPUs on the node as the places that OpenMP threads are pinned on, while `OMP_PROC_BIND` chooses the order of places in the pinning. 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. Studies in [2] with `OMP_PLACES=cores` using the choices of `OMP_PROC_BIND=close` and `OMP_PROC_BIND=spread` resulted in run time that were almost the same for corresponding values of nodes and processes per node 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.

We conduct complete performance studies of the test problem for six progressively finer meshes of $N = 1024$, 2048, 4096, 8192, 16384, 32768; additionally, we demonstrate how 32 nodes can be leveraged to solve two yet finer meshes with $N = 65536$ and 131072. These studies result in progressively larger systems of linear equations with system dimensions ranging from about 1 million for $N = 1024$ to over 1 billion for $N = 32768$; the two final meshes with $N = 65536$ and 131072 have over 4 billion and over 17 billion unknowns, respectively. The results for the last two resolutions are contained in Table 3.1.

Table 4.1 collects the results of the performance studies on the 2018 portion of the CPU cluster in taki. For each mesh resolution of the six meshes with $N = 1024$, 2048, 4096, 8192, 16384, 32768, the parallel implementation of the

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

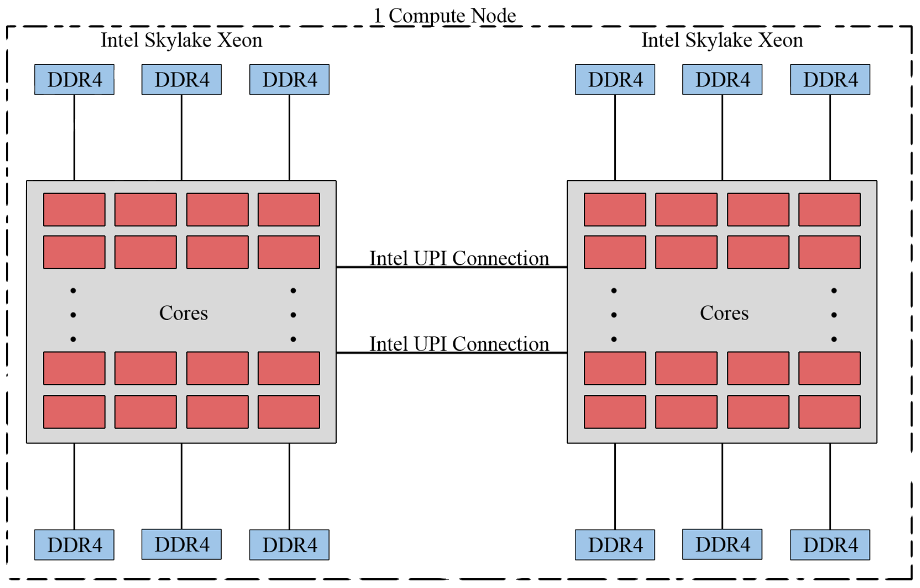


Figure 4.1: Schematic of a compute node with two Intel Skylake Xeon CPUs.

test problem is run on all possible combinations of nodes from 1 to 32 by powers of 2 and processes per node from 1 to 32 by powers of 2. The table summarizes 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 1-process run, i.e., 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 18-core processors, i.e., not using all available cores, on 32 nodes for a total of 1024 parallel processes working together to solve the problem.

We choose the mesh resolution of 16384×16384 in Table 4.1 to discuss in detail as example. Reading along the first column of this mesh subtable, we observe that by doubling the number of processes from 1 to 2 we approximately halve the runtime from each column to the next. We observe the same improvement from 2 to 4 processes as well as from 4 to 8processes. We also observe that by doubling the number of processes from 8 to 16 processes, there is still a significant improvement in runtime, although not the halving we observed previously. Finally, while the decrease in run time from 16 to 32 processes is small, the run times still do decrease, making the use of all available cores advisable. We observe that the behavior is analogous also in all other columns for this subtable. This behavior is a typical characteristic of memory-bound code such as this. The limiting factor in performance of memory-bound code is memory access, so we would expect a bottleneck when more processes on each CPU attempt to access the memory simultaneously than the available 6 memory channels per CPU; see Figure 4.1.

Reading along each row of the 16384×16384 mesh subtable, we observe that by doubling the number of nodes used, and thus also doubling the number of parallel processes, we approximately halve the runtime all the way up to 32 nodes. This behavior observed for increasing the number of nodes confirms the quality of the high-performance InfiniBand interconnect. Also, we can see that the timings for anti-diagonals in Table 4.1 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. Thus, it is advisable to use the smallest number of nodes with the largest number of processes per node.

When comparing now all subtables in Table 4.1, we observe that when we double the size of the mesh from one subtable to the next, the run times increase by a factor of about 8 to 10 for corresponding entries. The relative performance in each of the subtables in Table 4.1 exhibits largely analogous behavior to the 16384×16384 mesh, in particular the 8192×8192 and the 32768×32768 mesh subtables. For smaller meshes, some times for larger numbers of nodes are eventually so fast that improvement is small with more processes per node, but behavior is analogous for the more significant times.

Parallel scalability is often visually represented by plots of observed speedup and efficiency. The ideal behavior of code for a fixed problem size N using p parallel processes is that it be p times as fast as serial code. If $T_p(N)$ denotes the wall clock time for a problem of a fixed size parameterized by N using p processes, then the quantity $S_p = T_1(N)/T_p(N)$ measures the speedup of the code from 1 to p processes, whose optimal value is $S_p = p$. The efficiency $E_p = S_p/p$ characterizes in relative terms how close a run with p parallel processes is to this optimal value, for which $E_p = 1$. The behavior described here for speedup for a fixed problem size is known as strong scalability of parallel code.

Table 4.2 organizes the results of Table 4.1 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 4.2 (a) lists the raw timing data, like Table 4.1, but organized by numbers of parallel processes p. Tables 4.2 (b) and (c) show the numbers for speedup and efficiency, respectively, that will be visualized in Figures 4.2 (a) and (b), respectively. There are several choices for most values of p, such as for instance for $p = 4$, one could use 1 node with 4 processes per node, 2 nodes with 2 processes per node, or 4 nodes with 1 process per node. In all cases, we use the smallest number of nodes possible, with 32 processes per node for $p \geq 32$; for $p < 32$, only one node is used, with the remaining cores idle. Comparing adjacent columns in the raw timing data in Table 4.2 (a) confirms our previous observation that 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. The speedup numbers in Table 4.2 (b) help reach the same conclusions when speedup is near-optimal with $S_p \approx p$ for $p \leq 8$. For $p = 16$, sub-optimal speedup is clear. The speedup numbers also indicate sub-optimal speedup for $p > 16$, but recall that the runtimes clearly showed halving from each column to the next one; the speedup numbers can only give this indication qualitatively. The efficiency data in Table 4.2 (c) can bring out these effects more quantitatively, namely efficiency is near-optimal $E_p \approx 1$ for $p \leq 8$, then clearly identifies the efficiency drop taking place from $p = 8$ to $p = 16$. But for $p > 16$, the efficiency numbers stay essentially constant, which confirms quantitatively the aforementioned halving of runtimes from each column to the next one.

The plots in Figures 4.2 (a) and (b) visualize the numbers in Tables 4.2 (b) and (c), respectively. These plots do not provide new data but simply provide a graphical representation of the results in Table 4.2. 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 over the calculation time as quickly as it does for small problems. This is born out generally by both plots in Figure 4.2. Specifically, the speedup in Figure 4.2 (a) appears near-optimal up to

Table 4.1: Wall clock time in HH:MM:SS on taki 2018 using the default Intel compiler and Intel MPI, version 18.0.3.

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

	1 node	2 nodes	4 nodes	8 nodes	16 nodes	32 nodes
1 process per node	00:00:09	00:00:03	00:00:02	00:00:01	00:00:00	00:00:00
2 processes per node	00:00:03	00:00:02	00:00:01	00:00:00	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	00:00:00
8 processes per node	00:00:01	00:00:00	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	00:00:00
32 processes per node	00:00:00	00:00:01	00:00:01	00:00:01	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	32 nodes
1 process per node	00:01:34	00:00:43	00:00:18	00:00:07	00:00:03	00:00:02
2 processes per node	00:00:43	00:00:16	00:00:07	00:00:03	00:00:02	00:00:01
4 processes per node	00:00:23	00:00:09	00:00:04	00:00:02	00:00:01	00:00:00
8 processes per node	00:00:12	00:00:05	00:00:02	00:00:01	00:00:00	00:00:00
16 processes per node	00:00:09	00:00:03	00:00:01	00:00:00	00:00:00	00:00:00
32 processes per node	00:00:06	00:00:02	00:00:01	00:00:01	00:00:01	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	32 nodes
1 process per node	00:13:32	00:06:36	00:03:12	00:01:36	00:00:38	00:00:14
2 processes per node	00:06:41	00:03:12	00:01:30	00:00:37	00:00:14	00:00:07
4 processes per node	00:03:23	00:01:38	00:00:46	00:00:20	00:00:07	00:00:04
8 processes per node	00:01:48	00:00:51	00:00:24	00:00:10	00:00:04	00:00:02
16 processes per node	00:01:15	00:00:38	00:00:19	00:00:06	00:00:02	00:00:01
32 processes per node	00:01:03	00:00:31	00:00:14	00:00:04	00:00:02	00:00:02

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

	1 node	2 nodes	4 nodes	8 nodes	16 nodes	32 nodes
1 process per node	01:52:32	00:56:47	00:28:14	00:14:47	00:07:20	00:03:40
2 processes per node	00:57:03	00:28:08	00:13:49	00:06:38	00:03:04	00:01:21
4 processes per node	00:29:10	00:14:27	00:07:10	00:03:27	00:01:36	00:00:42
8 processes per node	00:15:12	00:07:34	00:03:44	00:01:49	00:00:51	00:00:22
16 processes per node	00:10:09	00:05:12	00:02:39	00:01:17	00:00:39	00:00:15
32 processes per node	00:08:45	00:04:24	00:02:11	00:01:05	00:00:31	00:00:11

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

	1 node	2 nodes	4 nodes	8 nodes	16 nodes	32 nodes
1 process per node	15:07:31	07:34:50	03:50:43	01:55:51	00:58:18	00:29:45
2 processes per node	07:37:16	03:50:04	01:54:52	00:56:46	00:27:51	00:13:26
4 processes per node	03:56:04	01:57:46	00:58:49	00:29:18	00:14:26	00:06:57
8 processes per node	02:03:10	01:01:35	00:30:48	00:15:22	00:07:35	00:03:44
16 processes per node	01:22:35	00:41:06	00:20:37	00:10:35	00:05:16	00:02:37
32 processes per node	01:11:27	00:35:42	00:17:55	00:09:02	00:04:40	00:02:21

(f) Mesh resolution $N \times N = 32768 \times 32768$, system dimension 1073741824

	1 node	2 nodes	4 nodes	8 nodes	16 nodes	32 nodes
1 process per node	129:18:17	64:15:32	32:08:45	16:13:22	08:06:37	04:01:54
2 processes per node	64:44:32	32:21:54	16:12:47	09:35:55	04:21:39	02:39:27
4 processes per node	33:04:35	18:11:46	09:54:45	04:58:51	02:16:26	01:20:41
8 processes per node	17:13:52	08:44:30	04:36:02	02:14:35	01:17:27	00:40:58
16 processes per node	11:19:13	05:53:54	03:00:33	01:29:34	00:45:11	00:24:23
32 processes per node	09:40:51	04:52:29	02:27:33	01:14:26	00:38:57	00:20:38

$p = 512$ for all problem sizes $N \geq 4096$. From $p = 512$ to $p = 1024$, we see the expected dramatic decrease in speedup that the raw run times exhibit. One would expect that the efficiency plot in Figure 4.2 (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. For larger values of p, excluding the final value at $p = 1024$, the horizontal shape of the lines in the efficiency plot brings out that no further degradation of performance occurs as p increases for large N.

Table 4.2: Strong scalability study on taki 2018 using the default Intel compiler and Intel MPI, version 18.0.3.

(a) Wall clock time T_p in HH:MM:SS

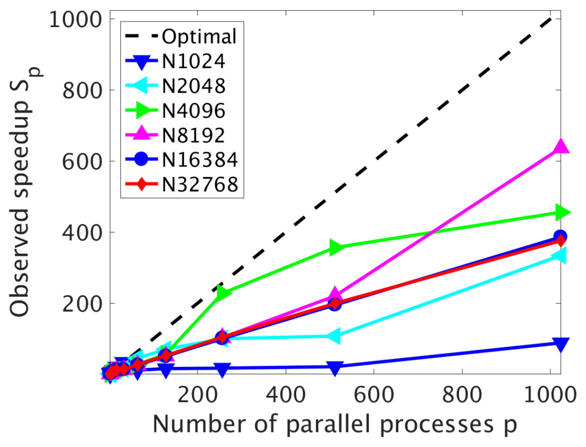
N	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$	$p=256$	$p=512$	$p=1024$
1024	00:00:09	00:00:03	00:00:02	00:00:01	00:00:00	00:00:00	00:00:01	00:00:01	00:00:01	00:00:00	00:00:00
2048	00:01:34	00:00:43	00:00:23	00:00:12	00:00:09	00:00:06	00:00:02	00:00:01	00:00:01	00:00:01	00:00:00
4096	00:13:32	00:06:41	00:03:23	00:01:48	00:01:15	00:01:03	00:00:31	00:00:14	00:00:04	00:00:02	00:00:02
8192	01:52:32	00:57:03	00:29:10	00:15:12	00:10:09	00:08:45	00:04:24	00:02:11	00:01:05	00:00:31	00:00:11
16384	15:07:31	07:37:16	03:56:04	02:03:10	01:22:35	01:11:27	00:35:42	00:17:55	00:09:02	00:04:40	00:02:21
32768	129:18:17	64:44:32	33:04:35	17:13:52	11:19:13	09:40:51	04:52:29	02:27:33	01:14:26	00:38:57	00:20:38

(b) Observed speedup $S_p = T_1/T_p$

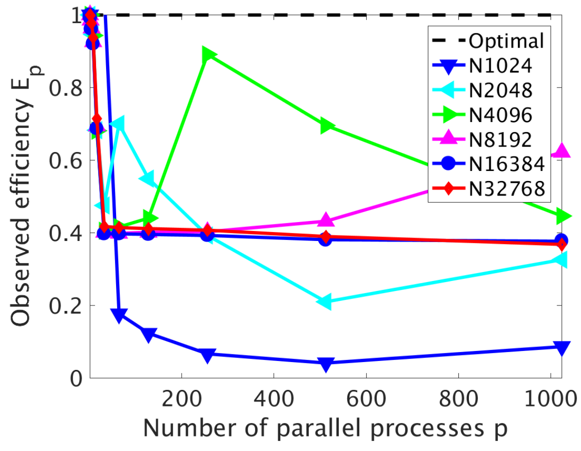
N	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$	$p=256$	$p=512$	$p=1024$
1024	1.00	2.81	5.25	10.14	19.17	33.92	11.31	15.75	16.96	21.00	88.20
2048	1.00	2.19	4.05	7.57	10.91	15.21	44.76	70.33	100.58	107.52	334.07
4096	1.00	2.03	4.01	7.55	10.90	12.97	26.57	56.38	228.21	356.33	456.43
8192	1.00	1.97	3.86	7.41	11.10	12.85	25.53	51.41	103.26	220.66	637.00
16384	1.00	1.98	3.84	7.37	10.99	12.70	25.43	50.65	100.51	194.73	385.80
32768	1.00	2.00	3.91	7.50	11.42	13.36	26.53	52.58	104.23	199.16	375.98

(c) Observed efficiency $E_p = S_p/p$

N	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$	$p=256$	$p=512$	$p=1024$
1024	1.00	1.40	1.31	1.27	1.20	1.06	0.18	0.12	0.07	0.04	0.09
2048	1.00	1.10	1.01	0.95	0.68	0.48	0.70	0.55	0.39	0.21	0.33
4096	1.00	1.01	1.00	0.94	0.68	0.41	0.42	0.44	0.89	0.70	0.45
8192	1.00	0.99	0.96	0.93	0.69	0.40	0.40	0.40	0.40	0.43	0.62
16384	1.00	0.99	0.96	0.92	0.69	0.40	0.40	0.40	0.39	0.38	0.38
32768	1.00	1.00	0.98	0.94	0.71	0.42	0.41	0.41	0.41	0.39	0.37



(a) Observed speedup S_p



(b) Observed efficiency E_p

Figure 4.2: Strong scalability study on taki 2018 using the default Intel compiler and Intel MPI, version 18.0.3.

5 Comparison of Performance Studies on Different Clusters

This section contains a comparison of performance studies for the test problem obtained on different distributed-memory clusters that existed at UMBC over time. The oldest cluster kali from 2003 was the first cluster with a high-performance interconnect at UMBC, that is, it was the first to have a second network dedicated to MPI communications, in addition to the primary management network. These clusters, acquired over the years, give an impression what was the state-of-the-art CPU model and what constituted a high-performance interconnect at that time, on the concrete example of the test problem used in this report. Additionally, we include the cluster Stampede2 in the Texas Advanced Computing Center (TACC), a nationally funded supercomputer, which has the same generation CPU as taki 2018.

Table 5.1 reports results for the test problem on the historical mesh resolution of $N = 4096$, which was the largest resolution that could be solved on kali in 2003. The following list briefly describes each cluster in the table, see the cited references for detailed descriptions.

Cluster kali (2003) [1]: 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 processors (2.0 GHz clock speed) and 1 GB of memory per node, 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 cluster kali was funded by a SCREMS grant from the National Science Foundation and owned by the Department of Mathematics and Statistics at UMBC.

Cluster hpc (2008) [5]: The cluster hpc was a 35-node distributed-memory cluster with 33 compute nodes plus 1 development and 1 combined user/management node, each 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 cluster hpc was the first cluster of HPCF, when HPCF was formed by seed funding from UMBC and contributions of faculty in 2008.

Cluster tara (2009) [11]: The cluster tara was an 86-node distributed-memory cluster with two quad-core Intel Nehalem X5550 processors (2.6 GHz clock speed, 8 MB L3 cache, 3 memory channels) and 24 GB memory per node, a QDR InfiniBand interconnect, and 160 TB central storage. This cluster was the first one principally funded by an MRI grant from the National Science Foundation to the user community of HPCF.

Clusters maya 2009, 2010, 2013 [9]: The cluster tara became part of the cluster maya as maya 2009, and its QDR InfiniBand network was extended to the 2013 portion of maya. The results for maya 2009 used the identical hardware as the results for tara, but new middleware and new versions of the compiler and MPI implementation from 2013. The cluster maya 2010 was a 168-node distributed-memory cluster with two quad-core Intel Nehalem X5560 processors (2.8 GHz clock speed, 8 MB L3 cache, 3 memory channels) and 24 GB memory per node, connected by a DDR InfiniBand interconnect. The cluster maya 2013 was a 72-node cluster with two eight-core Intel E5-2650v2 Ivy Bridge CPUs (2.6 GHz clock speed, 20 MB L3 cache, 4 memory channels) and 64 GB memory per node, connected by same QDR InfiniBand as maya 2009. The cluster maya had a central storage of more than 750 TB.

Cluster taki 2018: The 2018 portion of the CPU cluster in taki consists of 42 compute nodes with two 18-core Intel Xeon Gold 6140 Skylake CPUs (2.3 GHz clock speed, 24.75 MB L3 cache, 6 memory channels) and 384 GB memory per node, connected by an EDR InfiniBand interconnect.

Cluster Stampede2 (2018) [2]: The last row in the table contains results for the Skylake portion of Stampede2 with each compute node having two 24-core Intel Xeon Platinum 8160 Skylake CPUs (2.1 GHz clock speed, 33 MB L3 cache, 6 memory channels) and 192 GB memory per node, connected by an Omni-Path network.

Results on kali: 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].

Results on hpc: 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

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

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) [5]	01:51:29	00:32:37 (3.42)	00:03:23 (32.95)	00:01:28 (76.01)
tara (2009) [11]	00:31:16	00:06:39 (4.70)	00:01:05 (28.86)	00:00:09 (208.44)
maya 2009 [9]	00:17:05	00:05:55 (2.89)	00:00:35 (29.29)	00:00:08 (128.13)
maya 2010 [9]	00:17:00	00:05:48 (2.93)	00:00:34 (30.00)	00:00:06 (170.00)
maya 2013 [9]	00:12:26	00:02:44 (4.55)	00:00:15 (49.07)	00:00:12 (62.17)
taki 2018	00:13:32	00:01:03 (12.97)	00:00:14 (56.11)	00:00:02 (456.43)
Stampede2 (2018) [2]	00:13:04	00:01:00 (13.07)	00:00:16 (49.00)	00:00:01 (784.00)

we observe a speedup of 76.01, less than the optimal speedup of 128, for a still respectable efficiency of 59.38% [5].

Results on tara: 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, but an excellent efficiency of 81.25% [11].

Results on maya: 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 optimal speedup of 256, or an efficiency of 50.00% [9]. 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, or an efficiency of 66.41% [9]. 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. [9].

Results on taki: Recall that each node in taki 2018 has two 18-core Intel Skylake CPUs, for a total of 36 cores available on each node. The data for "all cores" on 1 node and on 32 nodes uses actually 32 cores per node. We can observe by reading down the first column of Table 5.1 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; compare the clock speeds of each CPU specified above. 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 is an efficiency of only 27.10%, but still yields a time that is twice as fast than all cores of maya 2013. In fact, when comparing the "all cores" results of taki 2018 to the previous rows, we see that they are the fastest in each column and an improvement over all past results. We also see that the EDR InfiniBand on taki 2018 provides for excellent speedup, as the 32-node results have better than linear speedup, both comparing the 1-core and the 32-core cases, namely 56.11 instead of nominal 32 for the 1-core comparison, for instance. Then, for the all core comparison, the speedup of 456.43 of the nominal 1024 for 32 nodes with 32 cores gives an efficiency of only 44.57% compared to the serial run, but when compared to all cores on 1 node it is a speedup of 35.19, which is actually better than the nominal 32 improvement to be expected. We note that these better than optimal results typically result from the problem being divided into such small chunks on each core that they fit in cache there, but the multi-node performance is still a testament to the low latency of the network.

Results on Stampede2: The last row of the table shows the results for the Stampede2 cluster, which has the same generation Intel Skylake CPUs as taki 2018, but with 24 cores each instead of 18, for a total of 48 cores per node. Like for taki 2018, the data for "all cores" on 1 node and on 32 nodes uses actually 32 cores per node of the available 48 cores. The observations for taki 2018 also apply here in essentially the same way. It is notable that the serial run is actually slightly faster than on taki 2018, despite the nominally lower clock rate on Stampede2; see above. But as the remaining results show, the single-core results are not necessarily faster, and none of the differences between taki 2018 and Stampede2 are significant.

Acknowledgments

The hardware used in the computational studies is part of the UMBC High Performance Computing Facility (HPCF). The facility 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] Kritesh Arora, Carlos Barajas, and Matthias K. Gobbert. Parallel performance studies for an elliptic test problem on the Stampede2 cluster and comparison of networks. Technical Report HPCF–2018–10, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2018.

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

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

[5] 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.

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

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

[8] 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.

[9] 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.

[10] Peter S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann, 1997.

[11] 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.

[12] 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.

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