# Strong and Weak Scalability Studies for the 3-D Poisson Equation on the Taki 2018 Cluster

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

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

Technical Report HPCF–2019–2, hpcf.umbc.edu > Publications

**Abstract**

The new 2018 nodes in the 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. Parallel performance studies for the memory-bound test problem of the Poisson equation in three spatial dimensions yield several conclusions for the operation of the CPU cluster in taki. Strong scalability studies demonstrate excellent performance 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. Weak scalability studies confirm that best throughput is achieved by using all cores on a shared-memory node. Both types of performance studies highlight that parallel code is needed to take full advantage of the large number of computational cores on the high-memory 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 HPCF Governance Committee ultimately decided in 2017–2018 to order a new cluster from Dell using the funds of this grant. 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 ($12 \times 32$ GB DDR4), and a 120 GB SSD drive. The 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 the system description at hpcf.umbc.edu for photos, schematics, and more detailed information, also on the other portions of the taki cluster.

This report uses the Poisson equation with homogeneous Dirichlet boundary conditions on a unit cube domain in three spatial dimensions to test the performance of the 2018 portion of the CPU cluster in taki with both strong and weak scalability studies. The recent report [6] already studied the strong scalability for this problem on taki 2018. Two earlier reports also considered the problem, namely [10] with the same linear solver, but implemented in Matlab, and [11] using a parallel C code using MPI, but with a different linear solver. Many other previous reports considered the two-dimensional analogue of this test problem (see [3] for complete references), most recently using compute nodes with Intel Skylake CPUs [2, 3] on taki 2018 and [1] on Stampede2. Discretizing the spatial derivatives in the Poisson equation by the finite difference method yields a system of linear equations with a large, sparse, highly structured, symmetric positive definite system matrix. The linear system resulting from the two-dimensional version of the test problem is a classical test problem for iterative solvers and contained in several textbooks including [5, 7, 8, 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 implementation uses the C programming language, with MPI (Message Passing Interface) for communications between distributed-memory cluster nodes and with OpenMP multi-threading on the shared-memory nodes.

The studies in this report allow for a number of conclusions:

(1) With the pooled memory of 32 high-memory nodes of the 2018 portion of taki, a convergence study of the finite difference method is now possible up to a mesh size of $N \times N \times N = 4096 \times 4096 \times 4096$ mesh points, requiring a system of linear equations with dimension $n = N^3 =$ 68,719,476,736 or nearly 69 billion equations to be solved. The runtimes observed for serial jobs in this study motivates the use of parallel computing to significantly speed up the studies.

(2) The strong scalability studies demonstrate excellent performance 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. For jobs using large numbers of processes, studies of all combinations of numbers of nodes and numbers of processes per node show that best throughout is typically achieved by using the fewest number of nodes that can accommodate the desired numbers of processes. Moreover, weak scalability studies show that there is no downside to using MPI-only code also within a shared-memory node, whether only one node is used or multiple nodes.

The remainder of this report is organized as follows: Section 2 details the test problem and discusses the parallel implementation in more detail, and Section 3 summarizes the solution and method convergence data. Section 4 contains the strong scalability studies using MPI-only code on taki 2018, while Section 5 contains a weak scalability study using hybrid MPI+OpenMP code on taki 2018.

## 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] for the two-dimensional analogue)

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

on the unit cube domain $\Omega = (0,1) \times (0,1) \times (0,1) \subset \mathbb{R}^3$ in three spatial dimensions. Here, $\partial\Omega$ denotes the boundary of the domain $\Omega$ and the Laplace operator in is defined as $\triangle u = \frac{\partial^2 u}{\partial x_1^2} + \frac{\partial^2 u}{\partial x_2^2} + \frac{\partial^2 u}{\partial x_3^2}$. 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 $(x_{k_1}, x_{k_2}, x_{k_3}) \in \overline{\Omega} \subset \mathbb{R}^3$ with $x_{k_i} = h\, k_i$, $k_i = 0, 1, \ldots, N, N+1$, in each dimension $i = 1,2,3$. Denote the approximations to the solution at the mesh points by $u_{k_1,k_2,k_3} \approx u(x_{k_1}, x_{k_2}, x_{k_3})$. Then approximate the second-order derivatives in the Laplace operator at the $N^2$ interior mesh points by

$$\frac{\partial^2 u(x_{k_1}, x_{k_2}, x_{k_3})}{\partial x_1^2} \approx \frac{u_{k_1-1,k_2,k_3} - 2u_{k_1,k_2,k_3} + u_{k_1+1,k_2,k_3}}{h^2}, \tag{2.2}$$

$$\frac{\partial^2 u(x_{k_1}, x_{k_2}, x_{k_3})}{\partial x_2^2} \approx \frac{u_{k_1,k_2-1,k_3} - 2u_{k_1,k_2,k_3} + u_{k_1,k_2+1,k_3}}{h^2}, \tag{2.3}$$

$$\frac{\partial^2 u(x_{k_1}, x_{k_2}, x_{k_3})}{\partial x_3^2} \approx \frac{u_{k_1,k_2,k_3-1} - 2u_{k_1,k_2,k_3} + u_{k_1,k_2,k_3+1}}{h^2} \tag{2.4}$$

for $k_i = 1, \ldots, N$, $i = 1,2,3$, for the approximations at the interior points. Using this approximation together with the homogeneous boundary conditions (2.1) gives a system of $N^3$ linear equations for the finite difference approximations at the $N^3$ interior mesh points.

Collecting the $N^3$ unknown approximations $u_{k_1,k_2,k_3}$ in a vector $u \in \mathbb{R}^{N^3}$ 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 $A \in \mathbb{R}^{N^3 \times N^3}$ and a right-hand side vector $b \in \mathbb{R}^{N^3}$. The components of the right-hand side vector $b$ are given by the product of $h^2$ multiplied by right-hand side function evaluations $f(x_{k_1}, x_{k_2}, x_{k_3})$ at the interior mesh points using the same ordering as the one used for $u_{k_1,k_2,k_3}$. The system matrix $A \in \mathbb{R}^{N^3 \times N^3}$ can be defined recursively as block tri-diagonal matrix with $N \times N$ blocks of size $N^2 \times N^2$ each, each of which in turn is a block tri-diagonal matrix with $N \times N$ blocks of size $N \times N$ each. Concretely, we have

$$A = \begin{bmatrix} S & -I_{N^2} & & & \\ -I_{N^2} & S & -I_{N^2} & & \\ & \ddots & \ddots & \ddots & \\ & & -I_{N^2} & S & -I_{N^2} \\ & & & -I_{N^2} & S \end{bmatrix} \in \mathbb{R}^{N^3 \times N^3}, \qquad S = \begin{bmatrix} T & -I_N & & & \\ -I_N & T & -I_N & & \\ & \ddots & \ddots & \ddots & \\ & & -I_N & T & -I_N \\ & & & -I_N & T \end{bmatrix} \in \mathbb{R}^{N^2 \times N^2}$$

with the tri-diagonal matrix $T = \text{tridiag}(-1, 6, -1) \in \mathbb{R}^{N \times N}$ for the diagonal blocks of $S$ as well as $A$ and identity matrices $I_{N^2} \in \mathbb{R}^{N^2 \times N^2}$ and $I_N \in \mathbb{R}^{N \times N}$ for the off-diagonal blocks of $A$ and $S$, respectively.

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` [9]. 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` [9]. 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 cube

$$\Omega = (0,1) \times (0,1) \times (0,1)$$

with right-hand side function

$$f(x_1,x_2,x_3) = (-2\pi^2)\Big(\cos(2\pi x_1)\sin^2(\pi x_2)\sin^2(\pi x_3)+\sin^2(\pi x_1)\cos(2\pi x_2)\sin^2(\pi x_3)+\sin^2(\pi x_1)\sin^2(\pi x_2)\cos(2\pi x_3)\Big),$$

for which the true analytic solution in closed form

$$u(x_1,x_2,x_3) = \sin^2(\pi x_1)\,\sin^2(\pi x_2)\,\sin^2(\pi x_3)$$

is known.

Table 3.1: Convergence study of the finite difference method 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 | 32,769 | 3.0060e–03 | — | 61 | < 00:00:01 | 0.01 | < 1 | < 1 |
| 64 | 262,144 | 7.7765e–04 | 3.86 | 120 | < 00:00:01 | 0.13 | < 1 | < 1 |
| 128 | 2,097,152 | 1.9763e–04 | 3.93 | 243 | 00:00:04 | 3.61 | < 1 | < 1 |
| 256 | 16,777,216 | 4.9807e–05 | 3.97 | 493 | 00:01:10 | 70.13 | < 1 | < 1 |
| 512 | 134,217,728 | 1.2503e–05 | 3.98 | 999 | 00:19:39 | 1,179.33 | 4 | 4.03 |
| 1024 | 1,073,741,824 | 3.1327e–06 | 3.99 | 2,023 | 05:40:09 | 20,408.66 | 32 | 32.04 |
| 2048 | 8,589,934,592 | 7.8346e–07 | 4.00 | 4,095 | 111:30:18 | 401,417.54 | 256 | 256.08 |
| *4096 | 68,719,476,736 | 1.9595e–07 | 4.00 | 8,289 | *04:08:07 | *14,887.35 | 2,048 | *2,327.87 |

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

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 $N \times N \times N$ mesh, the number of degrees of freedom $N^3$ (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 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 $4N^3$ double-precision numbers needed to store the four vectors of significant length $N^3$ 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 $N = 4096$ requires an excessive amount of runtime in serial as well as does not fit into the memory of one node. Therefore, 32 cores on 32 nodes are used for this case, with observed memory summed across all running processes to get the total usage.

In 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 [4, 8]. 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. 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 $N \leq 2048$, the observed memory usage in units of GB rounds to within less than 1 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 $N = 4096$, 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, namely with Table 7.3 (first subtable) in [10], which used the software Matlab, but the same method as linear solver. This ensures that the parallel performance studies in the next section are practically relevant, since a correct solution of the test problem is computed.

## 4 Performance Studies on taki 2018 Using MPI-Only Code

This section presents the strong scalability studies using MPI-only code on taki 2018 using the default Intel compiler and Intel MPI. The Intel compiler `icc` and the Intel MPI implementation, currently version 18.0.3, are accessed on taki through the wrapper `mpiicc`. Since the compiler and MPI implementation are the defaults, they are available after the `module load default-environment` command in the `.bashrc` file in the user's home directory that is automatically executed upon login to taki. We use the compiler options `-O3 -std=c99 -Wall`. We actually compile hybrid MPI+OpenMP code by including OpenMP multi-threading with the compiler option `-qopenmp`, but set the environment variable `OMP_NUM_THREAD` to 1 at run time. This limits the MPI+OpenMP code to 1 thread per MPI process, which is equivalent to running MPI-only code compiled without OpenMP.

HPCF uses the slurm workload manager (`slurm.schedmd.com`) for job scheduling. The slurm submission script uses the `mpirun` command to start the job, with the option `-genv I_MPI_PIN_PROCESSOR_LIST allcores:map=scatter` that is supposed to optimize the placement of MPI processes for MPI-only / single-threaded jobs.[1] 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 by 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[2] 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. Alternatively, the setting `OMP_PROC_BIND=true` just prevents the operating system from moving threads around. Studies in [1] with `OMP_PLACES=cores` using the choices of `OMP_PROC_BIND=close` and `OMP_PROC_BIND=spread` resulted in runtimes that were almost the same for corresponding values of nodes and processes per node used, which should be expected, since the environment variables tested are supposed to influence the placement of OpenMP threads.

We conduct complete performance studies of the test problem for five progressively finer meshes of $N = 128$, 256, 512, 1024, 2048; additionally, we demonstrate how 32 nodes can be leveraged to solve the yet finer mesh with $N = 4096$. These studies result in progressively larger systems of linear equations with system dimensions ranging from about 2 million for $N = 128$ to over 8 billion for $N = 2048$ and nearly 69 billion for $N = 4096$. The results for the last resolution are contained in Table 3.1.

### All possible nodes using all possible processes per node

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 five meshes with $N = 128$, 256, 512, 1024, 2048, the parallel implementation of the 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 on 32 nodes for a total of 1024 parallel processes working together to solve the problem. Notice that each node has two 18-core CPUs for a total of 36 cores, so even with 32 processes per node, several cores are not used by our job and remain available for the operating system and other system tasks. Since the algorithm block-distributes the data by dividing the last dimension with $N$ mesh points to the $p$ processes (the product of the number of nodes and the processes per node) is limited to be $p \leq N$; the notation "N/A" in the table indicates where this is violated.

We choose the mesh resolution with $N = 1024$ 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 8 processes. 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 runtime from 16 to 32 processes is small, the runtimes 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.

[1] Personal communication from Dell.

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

Table 4.1: Wall clock time in HH:MM:SS using MPI-only code on taki 2018 using the Intel compiler and Intel MPI.

| (a) Mesh resolution $N \times N \times N = 128 \times 128 \times 128$, system dimension 2097152 | | | | | | |
|---|---|---|---|---|---|---|
| | 1 node | 2 nodes | 4 nodes | 8 nodes | 16 nodes | 32 nodes |
| 1 process per node | 00:00:04 | 00:00:01 | 00:00:01 | 00:00:00 | 00:00:00 | 00:00:00 |
| 2 processes per node | 00:00:01 | 00:00:01 | 00:00:00 | 00:00:00 | 00:00:00 | 00:00:00 |
| 4 processes per node | 00:00:01 | 00:00:00 | 00:00:00 | 00:00:00 | 00:00:00 | 00:00:00 |
| 8 processes per node | 00:00:00 | 00:00:00 | 00:00:00 | 00:00:00 | 00:00:00 | N/A |
| 16 processes per node | 00:00:00 | 00:00:00 | 00:00:00 | 00:00:00 | N/A | N/A |
| 32 processes per node | 00:00:00 | 00:00:00 | 00:00:00 | N/A | N/A | N/A |

| (b) Mesh resolution $N \times N \times N = 256 \times 256 \times 256$, system dimension 16777216 | | | | | | |
|---|---|---|---|---|---|---|
| | 1 node | 2 nodes | 4 nodes | 8 nodes | 16 nodes | 32 nodes |
| 1 process per node | 00:01:10 | 00:00:35 | 00:00:17 | 00:00:08 | 00:00:03 | 00:00:01 |
| 2 processes per node | 00:00:35 | 00:00:17 | 00:00:08 | 00:00:03 | 00:00:01 | 00:00:01 |
| 4 processes per node | 00:00:18 | 00:00:09 | 00:00:04 | 00:00:02 | 00:00:01 | 00:00:01 |
| 8 processes per node | 00:00:09 | 00:00:05 | 00:00:02 | 00:00:01 | 00:00:01 | 00:00:00 |
| 16 processes per node | 00:00:06 | 00:00:03 | 00:00:02 | 00:00:01 | 00:00:00 | N/A |
| 32 processes per node | 00:00:06 | 00:00:03 | 00:00:02 | 00:00:01 | N/A | N/A |

| (c) Mesh resolution $N \times N \times N = 512 \times 512 \times 512$, system dimension 134217728 | | | | | | |
|---|---|---|---|---|---|---|
| | 1 node | 2 nodes | 4 nodes | 8 nodes | 16 nodes | 32 nodes |
| 1 process per node | 00:19:39 | 00:09:54 | 00:04:55 | 00:02:27 | 00:01:13 | 00:00:36 |
| 2 processes per node | 00:09:50 | 00:04:55 | 00:02:29 | 00:01:14 | 00:00:37 | 00:00:18 |
| 4 processes per node | 00:05:03 | 00:02:32 | 00:01:17 | 00:00:39 | 00:00:20 | 00:00:11 |
| 8 processes per node | 00:02:37 | 00:01:20 | 00:00:41 | 00:00:21 | 00:00:11 | 00:00:07 |
| 16 processes per node | 00:01:52 | 00:00:57 | 00:00:29 | 00:00:15 | 00:00:09 | 00:00:05 |
| 32 processes per node | 00:01:36 | 00:00:50 | 00:00:27 | 00:00:15 | 00:00:09 | N/A |

| (d) Mesh resolution $N \times N \times N = 1024 \times 1024 \times 1024$, system dimension 1073741824 | | | | | | |
|---|---|---|---|---|---|---|
| | 1 node | 2 nodes | 4 nodes | 8 nodes | 16 nodes | 32 nodes |
| 1 process per node | 05:40:09 | 02:50:29 | 01:24:45 | 00:42:36 | 00:21:29 | 00:10:43 |
| 2 processes per node | 02:49:20 | 01:25:16 | 00:43:51 | 00:21:30 | 00:10:53 | 00:05:33 |
| 4 processes per node | 01:31:52 | 00:44:05 | 00:21:59 | 00:11:05 | 00:05:40 | 00:02:56 |
| 8 processes per node | 00:43:52 | 00:22:10 | 00:11:14 | 00:05:42 | 00:03:01 | 00:01:37 |
| 16 processes per node | 00:30:04 | 00:15:09 | 00:07:44 | 00:04:02 | 00:02:10 | 00:01:15 |
| 32 processes per node | 00:26:32 | 00:12:59 | 00:06:48 | 00:03:40 | 00:02:06 | 00:01:14 |

| (e) Mesh resolution $N \times N \times N = 2048 \times 2048 \times 2048$, system dimension 8589934592 | | | | | | |
|---|---|---|---|---|---|---|
| | 1 node | 2 nodes | 4 nodes | 8 nodes | 16 nodes | 32 nodes |
| 1 process per node | 111:30:18 | 46:55:11 | 23:26:24 | 11:43:40 | 06:23:22 | 03:04:30 |
| 2 processes per node | 47:02:02 | 23:27:33 | 11:46:41 | 05:53:33 | 02:57:39 | 01:34:23 |
| 4 processes per node | 23:40:21 | 11:50:08 | 05:54:59 | 02:58:20 | 01:33:49 | 00:48:42 |
| 8 processes per node | 11:53:45 | 05:57:34 | 02:58:59 | 01:37:36 | 00:48:28 | 00:25:54 |
| 16 processes per node | 08:04:49 | 04:03:07 | 02:02:30 | 01:02:30 | 00:35:06 | 00:18:33 |
| 32 processes per node | 06:45:57 | 03:25:31 | 01:45:20 | 00:57:41 | 00:30:33 | 00:17:24 |

Reading along each row of the $N = 1024$ 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 for instance, the runtime for 2 nodes with 1 processes per node is almost same as for 1 node with 2 processes per node. 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 runtimes increase by a factor of about 16 to 20 for corresponding entries. The relative performance in each of the subtables in Table 4.1 exhibits largely analogous behavior to the $N = 1024$ mesh, in particular the $N = 512$ and the $N = 2048$ 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.

## All possible nodes using 32 processes per node

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 4.1 (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, from 2 to 4 processes, and from 4 to 8 processes, while not quite as good from 8 to 16 processes and from 16 to 32 processes, but near-perfect halving of runtimes again for $p \geq 32$. The speedup numbers in Table 4.2 (b) help reach the same conclusions when speedup is essentially optimal with $S_p \approx p$ for $p \leq 8$. For $p = 16$ and 32, sub-optimal speedup is visible. The speedup numbers also indicate sub-optimal speedup for $p \geq 32$, 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 for $p = 16$ and 32. But for $p \geq 32$, the efficiency numbers decrease only slowly, which confirms quantitatively the aforementioned halving of runtimes from each column to the next one for these columns.

The plots in Figures 4.1 (a) and (b) visualize the numbers in Tables 4.2 (b) and (c), respectively. These plots do not provide new results, but give a graphical representation of the data in Table 4.2. It is customary in strong scalability studies 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 4.1. Specifically, the lines in the speedup plot in Figure 4.1 (a) continue to exhibit speedup all the way throughout the plot. Even though the data are directly derived from the speedup data, the efficiency plot in Figure 4.1 (b) can provide additional insight into the observed behavior, particularly for small $p$. Here, the better-than-optimal behavior for $p \leq 32$ for the smallest mesh is noticeable now. This excellent performance of runs on several processes can result from local problems that fit better into the cache of each processor, which leads to fewer cache misses and thus potentially dramatic improvement of the runtime, beyond merely distributing the calculations to more processes. For $p > 32$, we observe decreasing efficiency for the smallest mesh. For the larger meshes, the efficiency plot shows that the precipitous drop of efficiency takes place at $p = 16$ and 32, but the only falls of very slowly for $p > 32$. For the largest mesh, we also see better-than-optimal behavior for $p \leq 32$; this cannot be explained by a better fit into cache though, but rather probably reflects a comparably poor serial runtime, since that is a case, where one computational core in the node accesses data that are stored in memory connected the other CPU on the node.

Table 4.2: Strong scalability study using MPI-only code on taki 2018 using the Intel compiler and Intel MPI.

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

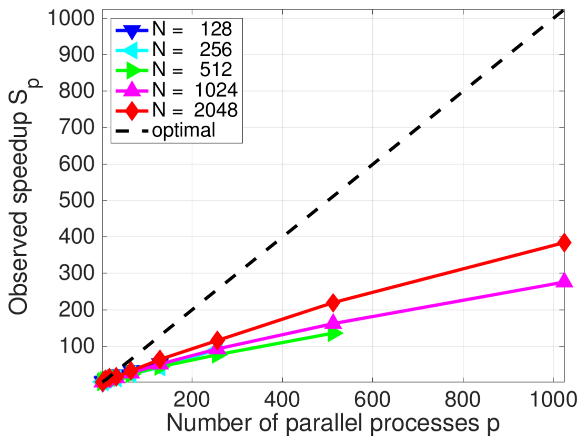
| $N$ | $p = 1$ | $p = 2$ | $p = 4$ | $p = 8$ | $p = 16$ | $p = 32$ | $p = 64$ | $p = 128$ | $p = 256$ | $p = 512$ | $p = 1024$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 128 | 00:00:04 | 00:00:01 | 00:00:01 | 00:00:00 | 00:00:00 | 00:00:00 | 00:00:00 | 00:00:00 | N/A | N/A | N/A |
| 256 | 00:01:10 | 00:00:35 | 00:00:18 | 00:00:09 | 00:00:06 | 00:00:06 | 00:00:03 | 00:00:02 | 00:00:01 | N/A | N/A |
| 512 | 00:19:39 | 00:09:50 | 00:05:03 | 00:02:37 | 00:01:52 | 00:01:36 | 00:00:50 | 00:00:27 | 00:00:15 | 00:00:09 | N/A |
| 1024 | 05:40:09 | 02:49:20 | 01:31:52 | 00:43:52 | 00:30:04 | 00:26:32 | 00:12:59 | 00:06:48 | 00:03:40 | 00:02:06 | 00:01:14 |
| 2048 | 111:30:18 | 47:02:02 | 23:40:21 | 11:53:45 | 08:04:49 | 06:45:57 | 03:25:31 | 01:45:20 | 00:57:41 | 00:30:33 | 00:17:24 |

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

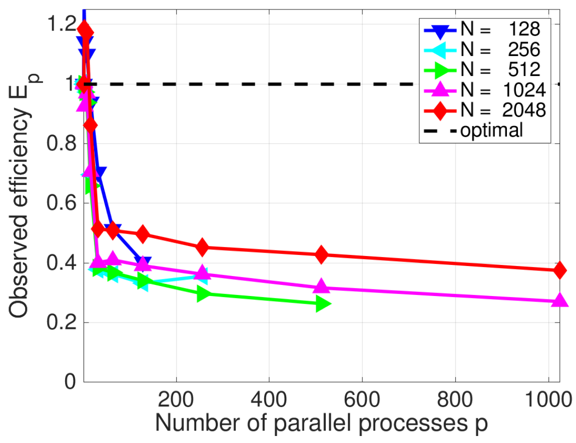
| $N$ | $p = 1$ | $p = 2$ | $p = 4$ | $p = 8$ | $p = 16$ | $p = 32$ | $p = 64$ | $p = 128$ | $p = 256$ | $p = 512$ | $p = 1024$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 128 | 1.00 | 2.51 | 4.57 | 8.80 | 15.04 | 22.56 | 32.82 | 51.57 | N/A | N/A | N/A |
| 256 | 1.00 | 2.02 | 3.94 | 7.78 | 11.13 | 12.13 | 23.22 | 42.50 | 91.08 | N/A | N/A |
| 512 | 1.00 | 2.00 | 3.89 | 7.50 | 10.56 | 12.27 | 23.59 | 43.70 | 76.28 | 135.40 | N/A |
| 1024 | 1.00 | 2.01 | 3.70 | 7.75 | 11.31 | 12.82 | 26.20 | 50.07 | 92.91 | 162.37 | 276.47 |
| 2048 | 1.00 | 2.37 | 4.71 | 9.37 | 13.80 | 16.48 | 32.55 | 63.52 | 115.98 | 219.01 | 384.60 |

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

| $N$ | $p = 1$ | $p = 2$ | $p = 4$ | $p = 8$ | $p = 16$ | $p = 32$ | $p = 64$ | $p = 128$ | $p = 256$ | $p = 512$ | $p = 1024$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 128 | 1.00 | 1.25 | 1.14 | 1.10 | 0.94 | 0.71 | 0.51 | 0.40 | N/A | N/A | N/A |
| 256 | 1.00 | 1.01 | 0.99 | 0.97 | 0.70 | 0.38 | 0.36 | 0.33 | 0.36 | N/A | N/A |
| 512 | 1.00 | 1.00 | 0.97 | 0.94 | 0.66 | 0.38 | 0.37 | 0.34 | 0.30 | 0.26 | N/A |
| 1024 | 1.00 | 1.00 | 0.93 | 0.97 | 0.71 | 0.40 | 0.41 | 0.39 | 0.36 | 0.32 | 0.27 |
| 2048 | 1.00 | 1.19 | 1.18 | 1.17 | 0.86 | 0.52 | 0.51 | 0.50 | 0.45 | 0.43 | 0.38 |



(a) Observed speedup $S_p$



(b) Observed efficiency $E_p$

Figure 4.1: Strong scalability study using MPI-only code on taki 2018 using the Intel compiler and Intel MPI.

## 5 Performance Studies on taki 2018 Using Hybrid MPI+OpenMP Code

This section presents a parallel performance study using hybrid MPI+OpenMP code on taki 2018. Parallel communication functions from MPI are needed to use multiple nodes in a distributed-memory cluster. But on each node with memory that is shared across all cores of the node, OpenMP multi-threading offers another way to parallelize code. More precisely, in hybrid MPI+OpenMP code, OpenMP multi-threading parallelizes each MPI process, thus we have the opportunity to access a chosen number of computational cores on a node by a combination of MPI processes and OpenMP threads per MPI process. Since the amount of computing power used on each node is fixed and only the way to access them is different from one run to another, one should expect comparable runtimes and the study is a weak scalability study with respect to each node. Since the amount of computing power used still increases with more nodes used, one should expect faster runtimes and the study is a strong scalability study with respect to all nodes used.

The Intel compiler `icc` and the Intel MPI implementation, currently version 18.0.3, are accessed on taki through the wrapper `mpiicc`. We use the compiler options `-O3 -std=c99 -Wall`. We compile hybrid MPI+OpenMP code by including OpenMP multi-threading with the compiler option `-qopenmp`. This OpenMP multi-threading is implemented using `#pragma` lines for performance-critical for-loops and other parallel portions of the code. The environment variable `OMP_NUM_THREAD` controls the number of OpenMP threads per MPI process at run time. This is set in the slurm submission script after the number of nodes are controlled by the `--nodes` option and the number of MPI processes per node by the `--ntasks-per-node`. Each node that is used is dedicated to the job with the remaining cores idling by using the `--exclusive` flag. Correspondingly, we request all memory of the node for the job by `--mem=MaxMemPerNode`. The slurm submission script uses the `mpirun` command to start the job. For runs with `OMP_NUM_THREAD` equal to 1, `mpirun` is called with the option `-genv I_MPI_PIN_PROCESSOR_LIST allcores:map=scatter` that is supposed to optimize the placement of MPI processes for MPI-only / single-threaded jobs, while for `OMP_NUM_THREAD` greater than 1, `mpirun` is called with the option `-genv I_MPI_PIN_DOMAIN omp` that is appropriate for running hybrid MPI+OpenMP codes.[3] We also use the environment variables `OMP_PLACES=cores` and `OMP_PROC_BIND=spread` in the slurm script.

Table 5.1 collects the results of the performance studies on the 2018 portion of the CPU cluster in taki. For each mesh resolution of the five meshes with $N = 128$, 256, 512, 1024, 2048, the parallel implementation of the test problem is run on $N_n = 1, 2, 4, \ldots, 32$ nodes. On each node, 32 computational cores are used, accessed by a combination of $p_n$ MPI processes per node and $t_p$ OpenMP threads per MPI process such that their product $p_n\, t_p = 32$ total threads per node. Thus, 32 hardware cores per node are in use in each entry of the table, and with respect to the processes and threads per node, this study constitutes a weak scalability study, since the runtimes 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 runtimes should nominally halve from column to column with the doubling of nodes used. The table summarizes the observed wall clock time (total time to execute the code) in HH:MM:SS (hours:minutes:seconds) format. Since the algorithm block-distributes the data by dividing the last dimension with $N$ mesh points to the $p$ MPI processes, $p = N_n\, p_n$ is limited to be $p \leq N$; the notation "N/A" in the table indicates where this is violated.

Each subtable in Table 5.1 shows results for one mesh with resolution $N \times N \times N$. Reading along each column in each subtable shows approximately the same runtime for each entry, confirming the weak scalability of the hybrid MPI+OpenMP code with respect to the fixed 32 computational cores per node. Reading along each row in each subtable shows an approximate halving of the runtime as the number of nodes doubles from column to column, confirming the strong scalability of the code with respect to increasing number of nodes used. More subtly, the last row of each subtable exhibits a slightly better performance than the other rows in nearly all cases. This is the row of single-threaded runs in each subtable that agrees up to experimental variability with the corresponding last row of each subtable in Table 4.1. This indicates how efficient a pure MPI-only code accesses the cores of each node.

[3]Personal communication from Dell.

Table 5.1: Wall clock time in HH:MM:SS using hybrid MPI+OpenMP code on taki 2018 using the Intel compiler and Intel MPI. Each run uses $N_n$ nodes, $p_n$ MPI processes per node, and $t_p = 32/p_n$ OpenMP threads per MPI process.

(a) Mesh resolution $N \times N \times N = 128 \times 128 \times 128$, system dimension 2097152

| | | $N_n = 1$ | $N_n = 2$ | $N_n = 4$ | $N_n = 8$ | $N_n = 16$ | $N_n = 32$ |
|---|---|---|---|---|---|---|---|
| $p_n = 1$ | $t_p = 32$ | 00:00:00 | 00:00:00 | 00:00:00 | 00:00:00 | 00:00:00 | 00:00:00 |
| $p_n = 2$ | $t_p = 16$ | 00:00:00 | 00:00:00 | 00:00:00 | 00:00:00 | 00:00:00 | 00:00:00 |
| $p_n = 4$ | $t_p = 8$ | 00:00:00 | 00:00:00 | 00:00:00 | 00:00:00 | 00:00:00 | 00:00:00 |
| $p_n = 8$ | $t_p = 4$ | 00:00:00 | 00:00:00 | 00:00:00 | 00:00:00 | 00:00:00 | N/A |
| $p_n = 16$ | $t_p = 2$ | 00:00:00 | 00:00:00 | 00:00:00 | 00:00:00 | N/A | N/A |
| $p_n = 32$ | $t_p = 1$ | 00:00:00 | 00:00:00 | 00:00:00 | N/A | N/A | N/A |

(b) Mesh resolution $N \times N \times N = 256 \times 256 \times 256$, system dimension 16777216

| | | $N_n = 1$ | $N_n = 2$ | $N_n = 4$ | $N_n = 8$ | $N_n = 16$ | $N_n = 32$ |
|---|---|---|---|---|---|---|---|
| $p_n = 1$ | $t_p = 32$ | 00:00:06 | 00:00:03 | 00:00:01 | 00:00:01 | 00:00:01 | 00:00:00 |
| $p_n = 2$ | $t_p = 16$ | 00:00:06 | 00:00:03 | 00:00:01 | 00:00:01 | 00:00:00 | 00:00:00 |
| $p_n = 4$ | $t_p = 8$ | 00:00:06 | 00:00:03 | 00:00:01 | 00:00:01 | 00:00:00 | 00:00:00 |
| $p_n = 8$ | $t_p = 4$ | 00:00:06 | 00:00:03 | 00:00:01 | 00:00:01 | 00:00:00 | 00:00:00 |
| $p_n = 16$ | $t_p = 2$ | 00:00:06 | 00:00:03 | 00:00:01 | 00:00:01 | 00:00:00 | N/A |
| $p_n = 32$ | $t_p = 1$ | 00:00:06 | 00:00:03 | 00:00:02 | 00:00:01 | N/A | N/A |

(c) Mesh resolution $N \times N \times N = 512 \times 512 \times 512$, system dimension 134217728

| | | $N_n = 1$ | $N_n = 2$ | $N_n = 4$ | $N_n = 8$ | $N_n = 16$ | $N_n = 32$ |
|---|---|---|---|---|---|---|---|
| $p_n = 1$ | $t_p = 32$ | 00:01:45 | 00:00:53 | 00:00:28 | 00:00:14 | 00:00:08 | 00:00:06 |
| $p_n = 2$ | $t_p = 16$ | 00:01:44 | 00:00:53 | 00:00:28 | 00:00:14 | 00:00:08 | 00:00:04 |
| $p_n = 4$ | $t_p = 8$ | 00:01:45 | 00:00:53 | 00:00:27 | 00:00:14 | 00:00:07 | 00:00:04 |
| $p_n = 8$ | $t_p = 4$ | 00:01:45 | 00:00:53 | 00:00:28 | 00:00:14 | 00:00:08 | 00:00:04 |
| $p_n = 16$ | $t_p = 2$ | 00:01:44 | 00:00:53 | 00:00:27 | 00:00:14 | 00:00:07 | 00:00:04 |
| $p_n = 32$ | $t_p = 1$ | 00:01:36 | 00:00:50 | 00:00:27 | 00:00:15 | 00:00:09 | N/A |

(d) Mesh resolution $N \times N \times N = 1024 \times 1024 \times 1024$, system dimension 1073741824

| | | $N_n = 1$ | $N_n = 2$ | $N_n = 4$ | $N_n = 8$ | $N_n = 16$ | $N_n = 32$ |
|---|---|---|---|---|---|---|---|
| $p_n = 1$ | $t_p = 32$ | 00:27:57 | 00:14:07 | 00:07:09 | 00:03:40 | 00:01:54 | 00:01:01 |
| $p_n = 2$ | $t_p = 16$ | 00:28:04 | 00:14:09 | 00:07:12 | 00:03:43 | 00:01:55 | 00:01:02 |
| $p_n = 4$ | $t_p = 8$ | 00:28:04 | 00:14:10 | 00:07:13 | 00:03:45 | 00:01:56 | 00:01:02 |
| $p_n = 8$ | $t_p = 4$ | 00:28:06 | 00:14:24 | 00:07:17 | 00:03:44 | 00:01:58 | 00:01:18 |
| $p_n = 16$ | $t_p = 2$ | 00:28:04 | 00:14:16 | 00:07:11 | 00:03:44 | 00:01:55 | 00:01:04 |
| $p_n = 32$ | $t_p = 1$ | 00:25:19 | 00:12:58 | 00:06:55 | 00:03:40 | 00:02:07 | 00:01:14 |

(e) Mesh resolution $N \times N \times N = 2048 \times 2048 \times 2048$, system dimension 8589934592

| | | $N_n = 1$ | $N_n = 2$ | $N_n = 4$ | $N_n = 8$ | $N_n = 16$ | $N_n = 32$ |
|---|---|---|---|---|---|---|---|
| $p_n = 1$ | $t_p = 32$ | 07:32:38 | 03:47:18 | 01:54:26 | 00:58:03 | 00:30:06 | 00:15:35 |
| $p_n = 2$ | $t_p = 16$ | 07:33:34 | 03:47:30 | 01:55:14 | 00:58:18 | 00:30:32 | 00:15:48 |
| $p_n = 4$ | $t_p = 8$ | 07:33:20 | 03:47:43 | 01:54:53 | 00:58:25 | 00:30:22 | 00:15:53 |
| $p_n = 8$ | $t_p = 4$ | 07:35:10 | 03:48:40 | 01:55:20 | 00:58:37 | 00:30:34 | 00:16:05 |
| $p_n = 16$ | $t_p = 2$ | 07:33:24 | 03:49:18 | 01:54:53 | 00:58:19 | 00:30:18 | 00:15:46 |
| $p_n = 32$ | $t_p = 1$ | 06:45:44 | 03:25:31 | 01:45:18 | 00:54:39 | 00:30:29 | 00:17:28 |

## 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. Co-author Carlos Barajas was supported by UMBC as HPCF RA.

## References

[1] 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.

[2] Carlos Barajas and Matthias K. Gobbert. Parallel performance studies for an elliptic test problem on the 2018 portion of the Taki cluster. Technical Report HPCF–2018–18, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2018.

[3] Carlos Barajas and Matthias K. Gobbert. Strong and weak scalability studies for the 2-D Poisson equation on the Taki 2018 cluster. Technical Report HPCF–2019–1, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2019.

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

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

[6] Richard Ebadi, Carlos Barajas, and Matthias K. Gobbert. Parallel performance studies for a 3-D elliptic test problem on the 2018 portion of the Taki cluster. Technical Report HPCF–2018–19, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2018.

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

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

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

[10] David Stonko, Samuel Khuvis, and Matthias K. Gobbert. Numerical methods to solve 2-D and 3-D elliptic partial differential equations using Matlab on the cluster maya. Technical Report HPCF–2014–9, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2014.

[11] Guan Wang and Matthias K. Gobbert. Performance comparison between blocking and non-blocking communications for a three-dimensional Poisson problem. Technical Report HPCF–2009–5, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2009.

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