Strong Scalability Studies for the 2-D and 3-D Poisson Equations on the Taki 2018 Cluster

Jinglong Sun, Ehsan Shakeri, and Matthias K. Gobbert (gobbert@umbc.edu)

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

Technical Report HPCF–2020–1, hpcf.umbc.edu > Publications

Abstract

The Poisson equation on both 2-D and 3-D spatial domains is a classical test problem for the performance of parallel computer code, since its memory-bound code provides a good test of the communication network. Different implementations of this classical elliptic test problem are tested on taki 2018 up to 32 nodes with two 18-core Intel Skylake Xeon CPUs, for a total of 36 cores per node, and 384 GB memory each, connected by a high-performance EDR InfiniBand network. Each case has three versions of the implementation: blocking MPI commands, non-blocking MPI commands, and non-blocking commands without conditional control flow. For both the 2-D and 3-D versions, as we double the number of nodes, the runtime is almost halved. This observation confirms the quality of the InfiniBand interconnect. By progressively doubling the number of processes per node, the runtime is also halved for the smaller numbers of 1, 2, 4, and 8 processes per node. However, for the largest numbers of 16 and 32 processes per node, the runtime for the code is not significantly reduced. In the case of the 3-D version, some entries with 32 processes per node are even slower than those with 16 processes per node.

1 Introduction

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

In 2017, the user community, represented by 51 researchers from 17 academic departments and research centers across UMBC, was successful for a third time to secure a grant from the National Science Foundation through its MRI program (grant no. OAC–1726023) for the extension and state-of-the-art update of HPCF. The 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 (12×32 GB DDR4 at 2666 MT/s) for a total of 10.6 GB per core, and a 120 GB SSD drive. Figure 1.1 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. 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 same test problem that has been used repeatedly to test cluster performance, see [1] for the 2-D spatial domain [2] for the 3-D spatial domain, and the references therein. The problem is the numerical solution of the Poisson equation with homogeneous Dirichlet boundary conditions on a unit square domain in two or three 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, 5, 6, 8] for the two-dimensional case. 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.

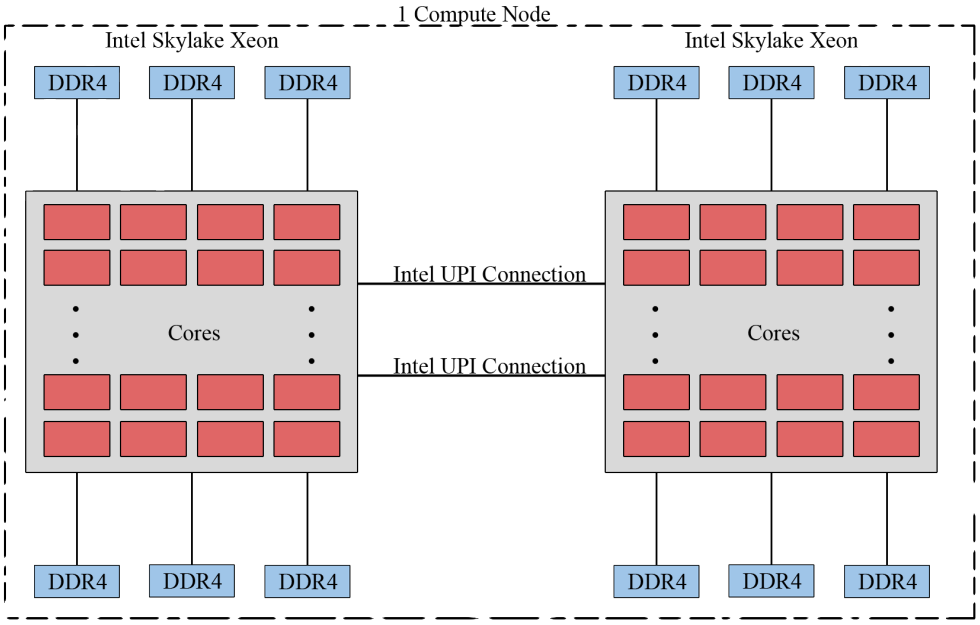


Figure 1.1: Schematic of a compute node with two Intel Skylake Xeon CPUs.

This report compares the performance of three versions of code for both cases of 2-D and 3-D spatial domains. The difference of the three code versions concerns the pairwise nearest process neighbor MPI communication that is needed in the algorithm. In the first version, blocking communication commands are used, employing an if-statement to differentiate between even and odd process numbers to prevent deadlock. In the second and third versions, non-blocking communication commands are utilized, with the third version being without conditional control flow by eliminating the if-statements within the for-loops.

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

(1) For both the 2-D and 3-D versions, as we double the number of nodes, the runtime is almost halved. This observation confirms the quality of the InfiniBand interconnect. By progressively doubling the number of processes per node, the runtime is also halved for the smaller numbers of 1, 2, 4, and 8 processes per node. However, for the largest numbers of 16 and 32 processes per node, the runtime for the code is not significantly reduced. In the case of the 3-D version, some entries with 32 processes per node are even slower than those with 16 processes per node. This behavior is a typical characteristic of memory-bound code such as this. The limiting factor in the performance of memory-bound code is memory access. Therefore, 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, as is indicated in the schematic of a compute node in Figure 1.1.

(2) In both the 2-D and 3-D cases, the arrangement of blocking code, with its deadlock avoidance, renders it efficient enough to compete with non-blocking code. There is no significant difference in performance between the code versions. There is also no noticeable improvement when employing non-blocking code without conditional control flow, which involves no if-statements in this case. In fact, some entries are even slightly slower.

The remainder of this report is organized as follows: Section 2 details the 2-D test problem and discusses the parallel implementation in more detail, while Section 3 summarizes the solution and method convergence data for the 2-D Poisson equation. Section 4 contains the strong scalability studies for blocking MPI commands, non-blocking MPI commands, and non-blocking commands without conditional control flow for the 2-D Poisson equation. Following that, Section 5 details the 3-D test problem and discusses the parallel implementation in more detail, with Section 6 summarizing the solution and method convergence data for the 3-D Poisson equation. Finally, Section 7 contains the strong scalability studies for blocking MPI commands, non-blocking MPI commands, and non-blocking commands without conditional control flow for the 3-D Poisson equation.

2 The Elliptic Test Problem with 2-D Poisson Equation

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

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

on the unit square domain $\Omega = (0,1) \times (0,1) \subset \mathbb{R}^2$. Here, $\partial\Omega$ denotes the boundary of the domain Ω and the Laplace operator in is defined as $\triangle u = \frac{\partial^2 u}{\partial x_1^2} + \frac{\partial^2 u}{\partial x_2^2}$. Using $N+2$ mesh points in each dimension, we construct a mesh with uniform mesh spacing $h = 1/(N+1)$. Specifically, define the mesh points $(x_{k_1}, x_{k_2}) \in \overline{\Omega} \subset \mathbb{R}^2$ with $x_{k_i} = h\, k_i$, $k_i = 0, 1, \ldots, N, N+1$, in each dimension $i = 1, 2$. Denote the approximations to the solution at the mesh points by $u_{k_1,k_2} \approx u(x_{k_1}, x_{k_2})$. Then approximate the second-order derivatives in the Laplace operator at the N^2 interior mesh points by

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

for $k_i = 1, \ldots, N$, $i = 1, 2$, 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

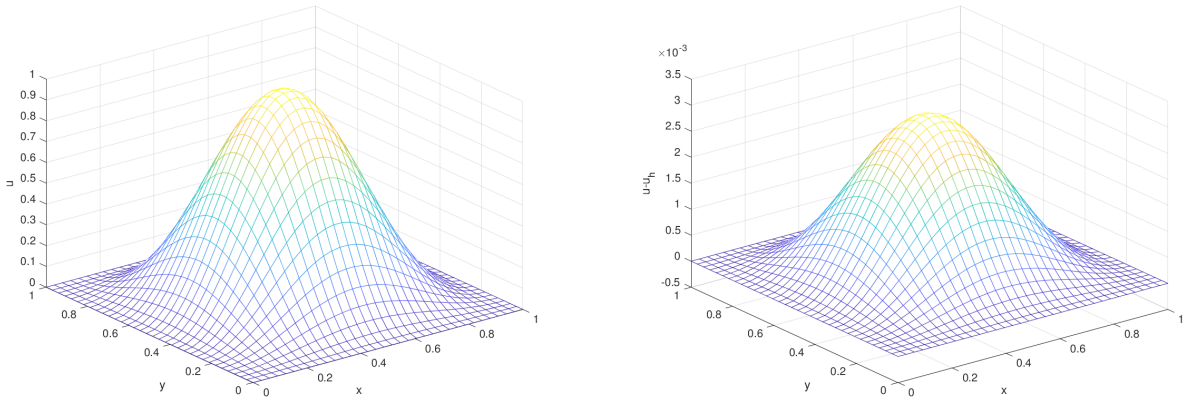
unavoidably dominate over the time used for the calculations that are done simultaneously in parallel — decreasing due to shorter local vectors for increasing number of processes. By contrast, the vector updates in each iteration can be executed simultaneously on all processes on their local blocks, 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` [7]. 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 communication commands `MPI_Isend` and `MPI_Irecv`.

Nominally, the interleaving achieved by using non-blocking communications should be the most efficient way to program the method. This report prefaces these results by a baseline study using the blocking MPI communication commands `MPI_Send` and `MPI_Recv`. Additionally, another experiment uses the non-blocking MPI communication command without conditional control flow by avoiding the if-statements inside the for-loops.

3 Convergence Study for the Model Problem with 2-D Poisson Equation

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



(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 of the finite difference method for the 2-D Poisson equation with serial code.

N	DOF	$\|u-u_h\|$	Ratio	`#iter`	wall clock time		memory usage (GB)	
					HH:MM:SS	seconds	predicted	observed
1024	1,048,576	3.1266e–06	—	1,581	00:00:09	8.79	< 1	< 1
2048	4,194,304	7.8019e–07	4.01	3,192	00:01:41	100.57	< 1	< 1
4096	16,777,216	1.9366e–07	4.03	6,452	00:15:04	904.21	< 1	< 1
8192	67,108,864	4.7376e–08	4.09	13,033	02:04:33	7,472.74	2	2.01
16384	268,435,456	1.1545e–08	4.10	26,316	17:02:09	61,329.24	8	8.02

$N \times N$ mesh, the number of degrees of freedom N^2 (DOF; i.e., the dimension of the linear system), the norm of the finite difference error $\|u-u_h\| \equiv \|u-u_h\|_{L^\infty(\Omega)}$, the ratio of consecutive errors $\|u-u_{2h}\|/\|u-u_h\|$, the number of conjugate gradient iterations `#iter`, the observed wall clock time in HH:MM:SS and in seconds, and the predicted and observed memory usage in GB for studies performed in serial. More precisely, the 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`.

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 [3, 6]. The fact that this convergence order is attained also confirms that the tolerance of the iterative linear solver is tight enough to ensure a sufficiently accurate solution of the linear system. 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. 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. 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, see [1] and the references therein. 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 2-D Performance Studies on taki 2018

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 -mkl` for the Intel Skylake CPUs.

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 `-print-rank-map` that is supposed to print the MPI's rank mapping. 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 conduct complete performance studies of the test problem for five progressively finer meshes of $N = 1024$, 2048, 4096, 8192, 16384. These studies result in progressively larger systems of linear equations with system dimensions ranging from about 1 million for $N = 1024$ to over 250 million for $N = 16384$.

Tables 4.1, 4.2, and 4.3 collect 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 $N = 1024$, 2048, 4096, 8192, 16384, 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.

2-D Performance Studies using Blocking MPI Commands

The blocking MPI commands used in this study study are `MPI_Send` and `MPI_Recv`. `MPI_Recv` will block the communication until it receives messages from another process. The implementation has all even-numbered processes `(id%2 == 0)` send first and then receive, and vice versa for all odd-numbered processes. This is to ensure that there is no "deadlock" and to improve efficiency.

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 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 indicated in Figure 1.1.

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 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 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 mesh subtable. 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.

Table 4.1: Wall clock time in HH:MM:SS using blocking MPI commands for the 2-D Poisson equation.

(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	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00

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

	1 node	2 nodes	4 nodes	8 nodes	16 nodes	32 nodes
1 process per node	00:01:41	00:00:48	00:00:19	00:00:08	00:00:04	00:00:02
2 processes per node	00:00:47	00:00:19	00:00:07	00:00:04	00:00:02	00:00:01
4 processes per node	00:00:24	00:00:11	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	00:00:08	00:00:03	00:00:01	00:00:00	00:00:00	00:00:00
32 processes per node	00:00:06	00:00:01	00:00:01	00:00:00	00:00:00	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	00:15:00	00:07:21	00:03:31	00:01:39	00:00:42	00:00:15
2 processes per node	00:07:21	00:03:31	00:01:36	00:00:42	00:00:15	00:00:08
4 processes per node	00:03:43	00:01:46	00:00:50	00:00:21	00:00:08	00:00:04
8 processes per node	00:01:49	00:00:52	00:00:25	00:00:11	00:00:04	00:00:02
16 processes per node	00:01:14	00:00:36	00:00:17	00:00:05	00:00:02	00:00:01
32 processes per node	00:01:03	00:00:30	00:00:13	00:00:03	00:00:02	00:00:01

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

	1 node	2 nodes	4 nodes	8 nodes	16 nodes	32 nodes
1 process per node	02:04:50	01:02:04	00:30:32	00:15:04	00:07:13	00:03:20
2 processes per node	01:01:42	00:30:33	00:15:07	00:07:14	00:03:23	00:01:27
4 processes per node	00:31:09	00:15:20	00:07:36	00:03:40	00:01:44	00:00:45
8 processes per node	00:15:39	00:07:35	00:03:47	00:01:50	00:00:53	00:00:24
16 processes per node	00:10:31	00:05:16	00:02:34	00:01:14	00:00:36	00:00:13
32 processes per node	00:08:56	00:04:26	00:02:11	00:01:04	00:00:29	00:00:09

(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	17:00:09	08:33:59	04:14:08	02:07:06	01:02:54	00:30:45
2 processes per node	08:30:15	04:17:52	02:08:52	01:03:30	00:31:06	00:14:55
4 processes per node	04:16:11	02:09:40	01:04:05	00:31:30	00:15:30	00:07:32
8 processes per node	02:07:47	01:03:18	00:31:43	00:15:45	00:07:49	00:03:49
16 processes per node	01:27:40	00:43:55	00:21:32	00:10:46	00:05:18	00:02:36
32 processes per node	01:12:18	00:36:40	00:18:24	00:09:10	00:04:34	00:02:17

2-D Performance Studies using Non-Blocking MPI Commands

The technique of implementing non-blocking MPI commands is known as interleaving calculations and communications and can be implemented using commands `MPI_Isend` and `MPI_Irecv` [7].

We choose the mesh resolution of 16384×16384 in Table 4.2 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 indicated in Figure 1.1.

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 4.2 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 4.2, 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 8 to 10 for corresponding entries. The relative performance in each of the subtables in Table 4.2 exhibits largely analogous behavior to the 16384×16384 mesh, in particular the 8192×8192 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.

When comparing absolute runtime in Table 4.1 and Table 4.2, we observed that there's no significant difference between implementing blocking MPI commands or non-blocking MPI commands. For example, in the 16384×16384 mesh subtables, some entries in Table 4.1 are slightly smaller and other entries in Table 4.2 are slightly smaller. The differences are also quite random and negligible.

Table 4.2: Wall clock time in HH:MM:SS using non-blocking MPI commands for the 2-D Poisson equation.

(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	00:00:04	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	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00

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

	1 node	2 nodes	4 nodes	8 nodes	16 nodes	32 nodes
1 process per node	00:01:41	00:00:58	00:00:19	00:00:07	00:00:04	00:00:02
2 processes per node	00:00:46	00:00:19	00:00:07	00:00:04	00:00:02	00:00:01
4 processes per node	00:00:24	00:00:10	00:00:04	00:00:02	00:00:01	00:00:00
8 processes per node	00:00:12	00:00:05	00:00:03	00:00:01	00:00:00	00:00:00
16 processes per node	00:00:08	00:00:03	00:00:01	00:00:00	00:00:00	00:00:00
32 processes per node	00:00:06	00:00:01	00:00:01	00:00:00	00:00:00	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	00:15:04	00:07:20	00:03:29	00:01:38	00:00:40	00:00:16
2 processes per node	00:07:22	00:05:05	00:01:37	00:00:41	00:00:23	00:00:08
4 processes per node	00:03:39	00:01:47	00:00:50	00:00:22	00:00:12	00:00:04
8 processes per node	00:01:48	00:00:52	00:00:25	00:00:10	00:00:06	00:00:02
16 processes per node	00:01:15	00:00:36	00:00:16	00:00:05	00:00:03	00:00:01
32 processes per node	00:01:04	00:00:30	00:00:13	00:00:03	00:00:01	00:00:01

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

	1 node	2 nodes	4 nodes	8 nodes	16 nodes	32 nodes
1 process per node	02:04:33	01:02:15	00:30:54	00:15:00	00:07:12	00:03:16
2 processes per node	01:01:53	00:30:56	00:15:07	00:07:11	00:03:17	00:01:23
4 processes per node	00:30:51	00:15:16	00:07:34	00:03:39	00:01:42	00:00:45
8 processes per node	00:20:39	00:07:44	00:03:49	00:01:50	00:00:53	00:00:24
16 processes per node	00:10:30	00:05:14	00:02:34	00:01:15	00:00:36	00:00:12
32 processes per node	00:08:56	00:04:25	00:02:11	00:01:04	00:00:27	00:00:13

(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	17:02:09	09:39:42	04:13:20	02:05:28	01:01:41	00:30:12
2 processes per node	08:30:55	04:13:17	02:05:38	01:02:10	00:30:36	00:14:43
4 processes per node	04:12:34	02:06:51	01:03:12	00:31:18	00:15:26	00:07:30
8 processes per node	02:06:54	01:03:56	00:44:51	00:15:44	00:07:47	00:03:48
16 processes per node	01:26:24	00:43:42	00:21:30	00:10:55	00:05:16	00:02:35
32 processes per node	01:12:43	00:36:26	00:18:28	00:09:13	00:04:31	00:02:18

2-D Performance Studies using Non-Blocking Code Without Conditional Control Flow

Both previous studies use if-statements as the conditional control flow. This study is focused on the non-blocking code without if-statements. The following snippets of pseudo-code explain the code with and without conditional control flow inside the for-loops:

2-D non-blocking code WITH if-statements:

```
for (i = 0; i < N; i++){
    if(i >  0 )    code for i > 0
    if(i < N-1) code for i < N-1
}
```

2-D non-blocking code WITHOUT if-statements:

```
code for i = 0

for (i = 1; i < N-1; i++){
    code for 0 < i < N-1
}

code for i = N-1
```

This study has the same combinations of nodes as the previous two studies. The Table 4.3 setup is also the same.

We choose the mesh resolution of 16384×16384 in Table 4.3 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 indicated in Figure 1.1.

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 4.3 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 4.3, 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 8 to 10 for corresponding entries. The relative performance in each of the subtables in Table 4.3 exhibits largely analogous behavior to the 16384×16384 mesh, in particular the 8192×8192 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.

When comparing absolute runtime in Table 4.2 and Table 4.3, we observed that there's no significant difference between the two cases. In fact, most of the entries in Table 4.3 are slightly larger than those in Table 4.2, although the differences are negligible. Therefore, eliminating conditional control flow won't significantly improve the performance in this case.

Table 4.3: Wall clock time in HH:MM:SS using non-blocking code without if-statements for the 2-D Poisson equation.

(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	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00
(b) Mesh resolution $N \times N = 2048 \times 2048$, system dimension 4194304						
	1 node	2 nodes	4 nodes	8 nodes	16 nodes	32 nodes
1 process per node	00:01:42	00:00:48	00:00:18	00:00:07	00:00:04	00:00:02
2 processes per node	00:00:48	00:00:19	00:00:07	00:00:04	00:00:02	00:00:01
4 processes per node	00:00:25	00:00:11	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	00:00:08	00:00:02	00:00:01	00:00:00	00:00:00	00:00:00
32 processes per node	00:00:06	00:00:01	00:00:01	00:00:00	00:00:00	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	00:14:52	00:07:22	00:03:31	00:01:37	00:00:39	00:00:15
2 processes per node	00:07:19	00:03:32	00:01:39	00:00:40	00:00:15	00:00:08
4 processes per node	00:03:40	00:01:46	00:00:50	00:00:20	00:00:08	00:00:04
8 processes per node	00:01:49	00:00:52	00:00:25	00:00:10	00:00:04	00:00:02
16 processes per node	00:01:15	00:00:36	00:00:17	00:00:05	00:00:02	00:00:01
32 processes per node	00:01:03	00:00:30	00:00:13	00:00:04	00:00:01	00:00:01
(d) Mesh resolution $N \times N = 8192 \times 8192$, system dimension 67108864						
	1 node	2 nodes	4 nodes	8 nodes	16 nodes	32 nodes
1 process per node	02:02:44	01:02:04	00:31:03	00:15:08	00:07:12	00:03:20
2 processes per node	01:01:48	00:30:53	00:15:08	00:07:12	00:03:22	00:01:30
4 processes per node	00:30:53	00:15:16	00:07:33	00:03:39	00:01:44	00:00:46
8 processes per node	00:15:21	00:07:41	00:03:47	00:01:49	00:00:53	00:00:23
16 processes per node	00:10:29	00:05:16	00:02:33	00:01:14	00:00:35	00:00:12
32 processes per node	00:08:56	00:04:27	00:02:10	00:01:03	00:00:28	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	17:17:38	08:48:29	04:18:49	02:05:53	01:02:22	00:30:43
2 processes per node	08:34:42	04:18:02	02:09:10	01:03:29	00:31:02	00:14:57
4 processes per node	04:17:04	02:09:17	01:03:48	00:31:52	00:15:31	00:07:33
8 processes per node	02:09:08	01:04:54	00:32:04	00:15:49	00:07:44	00:03:50
16 processes per node	01:26:58	00:44:01	00:21:43	00:10:42	00:05:17	00:02:36
32 processes per node	01:13:00	00:36:52	00:18:25	00:09:10	00:04:34	00:02:17

5 The Elliptic Test Problem with 3-D Poisson Equation

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

$$\begin{aligned} -\triangle u &= f \qquad \text{in } \Omega, \\ u &= 0 \qquad \text{on } \partial\Omega, \end{aligned} \tag{5.1}$$

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 Ω and the Laplace operator in is defined as $\triangle u = \frac{\partial^2 u}{\partial x_1^2} + \frac{\partial^2 u}{\partial x_2^2} + \frac{\partial^2 u}{\partial x_3^2}$. 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

$$\frac{\partial^2 u(x_{k_1}, x_{k_2}, x_{k_3})}{\partial x_1^2} \approx \frac{u_{k_1-1,k_2,k_3} - 2u_{k_1,k_2,k_3} + u_{k_1+1,k_2,k_3}}{h^2}, \tag{5.2}$$

$$\frac{\partial^2 u(x_{k_1}, x_{k_2}, x_{k_3})}{\partial x_2^2} \approx \frac{u_{k_1,k_2-1,k_3} - 2u_{k_1,k_2,k_3} + u_{k_1,k_2+1,k_3}}{h^2}, \tag{5.3}$$

$$\frac{\partial^2 u(x_{k_1}, x_{k_2}, x_{k_3})}{\partial x_3^2} \approx \frac{u_{k_1,k_2,k_3-1} - 2u_{k_1,k_2,k_3} + u_{k_1,k_2,k_3+1}}{h^2} \tag{5.4}$$

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 (5.1) 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` [7]. 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 communication commands `MPI_Isend` and `MPI_Irecv`.

Nominally, the interleaving achieved by using non-blocking communications should be the most efficient way to program the method. This report prefaces these results by a baseline study using the blocking MPI communication commands `MPI_Send` and `MPI_Recv`. Additionally, another experiment uses the non-blocking MPI communication command without conditional control flow by avoiding the if-statements inside the for-loops.

6 Convergence Study for the Model Problem with 3-D Poisson Equation

To test the numerical method and its implementation, we consider the elliptic problem (5.1) 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.

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 6.1 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`.

Table 6.1: Convergence study of the finite difference method for the 3-D Poisson equation with serial code.

N	DOF	$\|u - u_h\|$	Ratio	`#iter`	wall clock time		memory usage (GB)	
					HH:MM:SS	seconds	predicted	observed
128	2,097,152	1.9763e–04	—	244	00:00:04	3.68	< 1	< 1
256	16,777,216	4.9807e–05	3.97	493	00:01:11	70.83	< 1	< 1
512	134,217,728	1.2503e–05	3.98	999	00:20:18	1,217.70	4	4.02
1024	1,073,741,824	3.1327e–06	3.99	2,023	07:19:20	26,359.83	32	33.04

In all cases, the norms of the finite difference errors in Table 6.1 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, 6]. The fact that this convergence order is attained also confirms that the tolerance of the iterative linear solver is tight enough to ensure a sufficiently accurate solution of the linear system. 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 $N \leq 1024$, the observed memory usage in units of GB rounds to within less than 2 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. 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, see [2] and the references therein. This ensures that the parallel performance studies in the next section are practically relevant, since a correct solution of the test problem is computed.

7 3-D Performance Studies on taki 2018

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 -mkl`.

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 `-print-rank-map` that is supposed to print the MPI's rank mapping. 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 conduct complete performance studies of the test problem for five progressively finer meshes of $N = 128$, 256, 512, 1024. These studies result in progressively larger systems of linear equations with system dimensions ranging from about 2 million for $N = 128$ to over 1 billion for $N = 1024$.

Tables 7.1, 7.2, and 7.3 collect 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, 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.

3-D Performance Studies using Blocking MPI Commands

The blocking MPI commands used in this study study are `MPI_Send` and `MPI_Recv`. `MPI_Recv` will block the communication until it receives messages from another process. The implementation has all even-numbered processes `(id%2 == 0)` send first and then receive, and vice versa for all odd-numbered processes. This is to ensure that there is no "deadlock" and to improve efficiency.

We choose the mesh resolution with $N = 1024$ in Table 7.1 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 isn't ideal, some of the runtimes even increase. 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.

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 7.1 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 7.1, 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 7.1 exhibits largely analogous behavior to the $N = 1024$ mesh, in particular the $N = 512$ mesh subtable. 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.

Table 7.1: Wall clock time in HH:MM:SS using blocking MPI commands for the 3-D Poisson equation.

(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	00:00:02	00:00:01	00:00:00	00:00:00	00:00:00
2 processes per node	00:00:02	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	00:01:11	00:00:36	00:00:17	00:00:08	00:00:04	00:00:02
2 processes per node	00:00:36	00:00:17	00:00:08	00:00:04	00:00:02	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	00:00:01	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	00:20:16	00:10:10	00:05:04	00:02:32	00:01:16	00:00:38
2 processes per node	00:10:13	00:05:05	00:02:36	00:01:17	00:00:39	00:00:19
4 processes per node	00:05:04	00:02:41	00:01:20	00:00:42	00:00:21	00:00:12
8 processes per node	00:02:41	00:01:22	00:00:44	00:00:22	00:00:13	00:00:08
16 processes per node	00:01:55	00:00:59	00:00:31	00:00:17	00:00:10	00:00:07
32 processes per node	00:01:38	00:00:51	00:00:29	00:00:18	00:00:11	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	07:34:10	03:08:07	01:32:35	00:46:20	00:22:45	00:11:27
2 processes per node	03:26:18	01:31:03	00:45:57	00:23:29	00:11:44	00:05:54
4 processes per node	01:38:18	00:46:58	00:24:11	00:12:10	00:06:17	00:03:15
8 processes per node	00:48:56	00:23:34	00:11:54	00:06:23	00:03:20	00:01:52
16 processes per node	00:31:53	00:16:06	00:08:28	00:04:20	00:02:25	00:01:30
32 processes per node	00:29:59	00:14:06	00:07:13	00:03:59	00:02:25	00:01:33

3-D Performance Studies using Non-Blocking MPI Commands

The technique of implementing non-blocking MPI commands is known as interleaving calculations and communications and can be implemented using commands `MPI_Isend` and `MPI_Irecv` [7].

We choose the mesh resolution with $N = 1024$ in Table 7.2 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 isn't ideal, some of the runtimes even increase. 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.

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 7.2 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 7.2, 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 7.2 exhibits largely analogous behavior to the $N = 1024$ mesh, in particular the $N = 512$ mesh subtable. 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.

When comparing absolute runtime in Table 7.1 and Table 7.2, we observed that there's no significant difference between implementing blocking MPI commands or non-blocking MPI commands for the 3-D Poisson equation. For example, in the $N = 1024$ mesh subtables, some entries in Table 7.1 are slightly smaller and other entries in Table 7.2 are slightly smaller. The differences are also quite random and negligible.

Table 7.2: Wall clock time in HH:MM:SS using non-blocking MPI commands for the 3-D Poisson equation.

(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	00:01:11	00:00:35	00:00:17	00:00:08	00:00:04	00:00:01
2 processes per node	00:00:35	00:00:17	00:00:08	00:00:03	00:00:02	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	00:20:18	00:10:10	00:05:00	00:02:31	00:01:15	00:00:37
2 processes per node	00:09:57	00:05:00	00:02:32	00:01:16	00:00:38	00:00:18
4 processes per node	00:05:03	00:02:32	00:01:18	00:00:40	00:00:20	00:00:11
8 processes per node	00:02:42	00:01:22	00:00:42	00:00:21	00:00:11	00:00:07
16 processes per node	00:01:55	00:00:56	00:00:30	00:00:15	00:00:09	00:00:05
32 processes per node	00:01:43	00:00:50	00:00:27	00:00:16	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	07:19:20	03:34:28	01:29:36	00:45:18	00:22:41	00:11:17
2 processes per node	03:07:46	01:30:13	00:45:25	00:22:23	00:11:29	00:05:54
4 processes per node	01:28:08	00:46:40	00:23:35	00:11:38	00:05:59	00:03:08
8 processes per node	00:48:04	00:23:46	00:11:53	00:06:00	00:03:07	00:01:40
16 processes per node	00:31:41	00:17:33	00:08:14	00:04:12	00:02:13	00:01:21
32 processes per node	00:33:17	00:13:37	00:07:10	00:03:46	00:02:12	00:01:17

3-D Performance Studies using Non-Blocking Code Without Conditional Control Flow

Both previous studies use if-statements as the conditional control flow. This study is focused on the non-blocking code without if-statements. The following snippets of pseudo-code explain the code with and without conditional control flow inside the for-loops:

3-D non-blocking code WITH if-statements:

```
for (j = 0; j < N; j++){
    for (i = 0; i < N; i++){
        if(j > 0 )    code for j > 0
        if(i > 0 )    code for i > 0
        if(i < N-1)   code for i < N-1
        if(j < N-1)   code for j < N-1
    }
}
```

3-D non-blocking code WITHOUT if-statements:

```
code for j = 0

for (j = 1; j < N-1; j++){
    code for 0 < j < N-1
}

code for j = N-1
```

Notice that each piece of code here also doesn't have if-statements about i, which is similar to the 2-D version.

We choose the mesh resolution with $N = 1024$ in Table 7.3 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 isn't ideal, some of the runtimes even increase. 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.

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 7.3 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 7.3, 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 7.3 exhibits largely analogous behavior to the $N = 1024$ mesh, in particular the $N = 512$ mesh subtable. 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.

When comparing absolute runtime in Table 7.2 and Table 7.3, we observed that although most of the entries in Table 7.3 are slightly smaller than those in Table 7.2, there's no significant difference between the two cases. For example, in $N = 1024$ subtables, some timings are still smaller in Table 7.2, like the entry with 32 processes per node and 2 nodes. Therefore, eliminating conditional control flow won't significantly improve the performance in this case.

Table 7.3: Wall clock time in HH:MM:SS using non-blocking code without if-statements for the 3-D Poisson equation.

(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	00:00:02	00:00:01	00:00:00	00:00:00	00:00:00
2 processes per node	00:00:02	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	00:01:11	00:00:35	00:00:17	00:00:08	00:00:03	00:00:01
2 processes per node	00:00:37	00:00:17	00:00:08	00:00:04	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	00:20:12	00:10:31	00:05:13	00:02:30	00:01:14	00:00:36
2 processes per node	00:10:01	00:05:00	00:02:34	00:01:18	00:00:38	00:00:18
4 processes per node	00:05:06	00:02:34	00:01:18	00:00:41	00:00:20	00:00:10
8 processes per node	00:02:41	00:01:22	00:00:42	00:00:22	00:00:11	00:00:07
16 processes per node	00:01:52	00:00:58	00:00:30	00:00:16	00:00:09	00:00:05
32 processes per node	00:01:37	00:00:51	00:00:28	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	07:10:07	03:23:24	01:31:37	00:45:23	00:22:29	00:11:07
2 processes per node	03:21:59	01:30:11	00:46:20	00:23:18	00:11:41	00:05:44
4 processes per node	01:39:44	00:46:14	00:23:44	00:12:05	00:06:00	00:03:06
8 processes per node	00:47:46	00:23:44	00:12:10	00:06:06	00:03:11	00:01:45
16 processes per node	00:31:33	00:16:08	00:08:14	00:04:17	00:02:14	00:01:21
32 processes per node	00:28:55	00:13:40	00:07:06	00:03:53	00:02:11	00:01:17

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 Ehsan Shakeri was supported by UMBC as HPCF RA.

References

[1] 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.

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

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

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

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

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

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

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