A Comparative Study of the Parallel Performance of the Blocking and Non-Blocking MPI Communication Commands on an Elliptic Test Problem on the Cluster tara

Hafez Tari[†] and Matthias K. Gobbert[‡]

[†]Department of Mechanical Engineering, email: hafez.tari@umbc.edu
[‡]Department of Mathematics and Statistics, email: gobbert@umbc.edu
University of Maryland, Baltimore County, Baltimore, MD 21250, USA
Technical Report HPCF–2010–6, www.umbc.edu/hpcf > Publications

Abstract

In this report we study the parallel solution of the elliptic test problem of a Poisson equation with homogenous Dirichlet boundary conditions in a two dimensional domain. We use the finite difference method to approximate the governing equations with a system of N^2 linear equations, with N the number of interior grid points in either spatial direction. To parallelize the computation, we distribute blocks of the rows of the interior mesh point values among the parallel processes. We then use the iterative conjugate gradient method featured with a so-called matrix-free implementation to solve the system of linear equations local to any of the processes. The conjugate gradient method initiates with local vectors of zero elements, as the start solution, and updates the successive solutions until the Euclidean norm of the global residual of the local iterative solutions relative to that of the global residual of the local start solutions vanishes based on a predefined tolerance. To achieve this and considering the fact that the conjugate gradient method forces some communication between the neighboring processes, i.e. the processes possessing data of the grid interfaces, two modes of MPI communications, namely blocking and non-blocking send and receive, are employed for the data exchange between the processes. The obtained results given accordingly show excellent performance on the cluster tara with up to 512 parallel processes when using 64 compute nodes, especially once non-blocking MPI commands are used. The cluster tara is an IBM Server x iDataPlex purchased in 2009 by the UMBC High Performance Computing Facility (www.umbc.edu/hpcf). It is an 86-node distributed-memory cluster comprised of 82 compute, 2 develop, 1 user, and 1 management nodes. Each node features two quad-core Intel Nehalem X5550 processors (2.66 GHz, 8 MB cache), 24 GB memory, and a 120 GB local hard drive. All nodes and the 160 TB central storage are connected by an InfiniBand (QDR) interconnect network.

1 Introduction

The interplay of the processors in clusters and supercomputers, the architecture of their compute nodes, their interconnect network, the numerical algorithm used and its implementation are the key to devise parallel computer codes of decent performances. The solution of large, sparse, highly structured systems of linear equations by an iterative linear solver that requires communication between the parallel processes at every iteration is an instructive test of this interplay.

The numerical approximation of the classical elliptic test problem given by the Poisson equation with homogeneous Dirichlet boundary conditions on a unit square domain in two spatial dimensions by the finite difference method results in a large, sparse, highly structured system of linear equations. The parallel implementation of the conjugate gradient method as appropriate iterative linear solver for this linear system involves necessarily communications both between all participating 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. 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 bounded algorithms. See [2] for the details of the conjugate gradient method. Section 2 elaborates the elliptic test problem under investigation.

Tara is the cluster that we will be using for our experimentation. It is an 86-node distributed-memory cluster purchased in 2009 and comprised of 1 user, 1 management, 2 develop and 82 compute nodes. Each node features two quad-core Intel Nehalem X5550 processors (2.66 GHz, 8 MB cache), 24 GB memory, and a 120 GB local hard drive, thus up to 8 parallel processes can be run simultaneously per node. All nodes and the 160 TB central storage are connected by an InfiniBand (QDR = quad-data rate) interconnect network. The cluster is an IBM System x iDataPlex.[1] An iDataPlex rack uses the same floor space as a conventional 42 U high rack but holds up to 84 nodes, which saves floor space. More importantly, two nodes share a power supply which reduces the power requirements of the rack and make it potentially more environmentally friendly than a solution based on standard racks.[2] For

[1]Vendor page www-03.ibm.com/systems/x/hardware/idataplex/
[2]Press coverage for instance www.theregister.co.uk/2008/04/23/ibm_idataplex/

tara, the iDataPlex rack houses the 84 compute and develop nodes and includes all ethernet switches and other components associated with the nodes in the rack such as power distributors and ethernet switches. The user and management nodes with their larger form factor are contained in a second, standard rack along with the InfiniBand switch. Moreover, for our coding the MVAPICH2 implementation of MPI and for the compilation the PGI 9.0 C compiler have been used to create the executables which were utilized in this report.

Past results using an implementation of this method [1, 2, 3] on the previous clusters show that the interconnect network between the compute nodes must be high-performance, that is, have low latency and wide bandwidth, for this numerical method to scale well to many parallel processes. This note like [4] is an update to the technical report [3] with considering the same problem for the new cluster tara. Moreover, this letter is in line with [4] which studies the same problem for the same cluster, tara, but with an independent improved coding effort considering both blocking and non-blocking MPI communication commands.

2 The Elliptic Test Problem

We consider the classical elliptic test problem of the Poisson equation with homogeneous Dirichlet boundary conditions (see, e.g., [5, Chapter 7])

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

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

$$\triangle u = \frac{\partial^2 u}{\partial x_1^2} + \frac{\partial^2 u}{\partial x_2^2}.$$

Using $N+2$ mesh points in each dimension, we construct a mesh with uniform mesh spacing $h = 1/(N+1)$. Specifically, define the mesh points $(x_{k_1}, x_{k_2}) \in \overline{\Omega} \subset \mathbb{R}^2$ with $x_{k_i} = h\, k_i$, $k_i = 0, 1, \ldots, N, N+1$, in each dimension $i = 1, 2$. Denote the approximations to the solution at the mesh points by $u_{k_1,k_2} \approx u(x_{k_1}, x_{k_2})$. Then approximate the second-order derivatives in the Laplace operator at the N^2 interior mesh points by

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

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

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

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

with the tri-diagonal matrix $S = \text{tridiag}(-1, 4, -1) \in \mathbb{R}^{N \times N}$ 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 and 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 product 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 processors the time needed for communication — increasing with the number of processes — will unavoidably dominate over the time used for the calculations that are done simultaneously in parallel — decreasing due to shorter local vectors for increasing number of processes. By contrast, the vector updates in each iteration can be executed simultaneously on all processes on their local blocks, because they do not require any parallel communications. However, this assumes tacitly 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 as `MPI_Allreduce`. Finally, the matrix-vector product $q = A\,p$ also computes only the block of the vector q that is local to each process. But since the matrix A has non-zero off-diagonal elements, each local block needs values of p that are local to the two processes that hold the neighboring blocks of p. The communications between parallel processes thus needed are so-called point-to-point communications, because not all processes participate in each of them, but rather only specific pairs of processes that exchange data needed for their local calculations. Observe now that it is only a few components of q that require data from p that is not local to the process. Therefore, it is possible and potentially very efficient to proceed to calculate those components that can be computed from local data only, while the communication with the other processes is 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` which are known to be 'safe' against deadlock. See [6] for a detailed explanation. A less effective approach is to use the blocking MPI communications commands, `MPI_Send` and `MPI_Recv` which will not return until the arguments to the functions can be safely modified by subsequent statements in the program. We will compare the results of both methods for our case of study.

3 Convergence Study for the Model Problem

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

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

for which the solution $u(x_1, x_2) = \sin^2(\pi x_1)\, \sin^2(\pi x_2)$ is known. On a mesh with 34×34 points ($N = 32$) and mesh spacing $h = 1/33 = 0.03030303$, the numerical solution $u_h(x_1, x_2)$ can be plotted vs. (x_1, x_2) as a mesh plot as in Figure 3.1 (a). The shape of the solution clearly agrees with the true solution of the problem. At each mesh point, an error is incurred compared to the true solution $u(x_1, x_2)$. A mesh plot of the error $u - u_h$ vs. (x_1, x_2) is plotted in Figure 3.1 (b). We see that the maximum error occurs at the center of the domain of size about 3.2e–3, which compares well to the order of magnitude $h^2 \approx 0.92$e–3 of the theoretically predicted error.

To check the convergence of the finite difference method as well as to analyze the performance of the conjugate gradient method, we solve the problem on a sequence of progressively finer meshes. The conjugate gradient method is started with a zero vector as initial guess and the solution is accepted as converged when the Euclidean vector norm of the residual is reduced to the fraction 10^{-6} of the initial residual. Table 3.1 lists the mesh resolution N of the $N \times N$ mesh, the number of degrees of freedom N^2 (DOF; i.e., the dimension of the linear system), the norm of the finite difference error $\|u - u_h\|_{L^\infty(\Omega)}$, 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 MB for studies performed in serial. More precisely, the runs used the parallel code run on one process only, on a dedicated node (no other

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

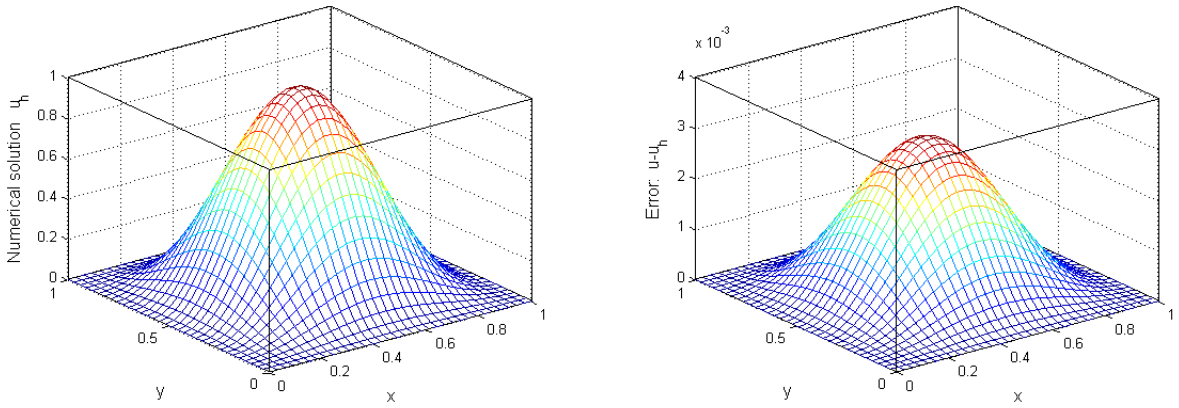


Figure 3.1: Mesh plots ($N = 32$) of (a) the numerical solution u_h vs. (x_1, x_2) and (b) the error $u - u_h$ vs. (x_1, x_2).

Table 3.1: Convergence study listing the mesh resolution N, the number of degrees of freedom (DOF), the norm of the finite difference error $\|u - u_h\|_{L^\infty(\Omega)}$, the number of conjugate gradient iterations to convergence, the observed wall clock time in HH:MM:SS and in seconds, and the predicted and observed memory usage in MB for a one-process run.

N	DOF	$\|u - u_h\|_{L^\infty(\Omega)}$	`#iter`	wall clock time		memory usage (MB)	
				HH:MM:SS	seconds	predicted	observed
32	1,024	3.0127e–03	48	<00:00:01	< 0.01	< 1	12
64	4,096	7.7810e–04	96	<00:00:01	< 0.01	< 1	12
128	16,384	1.9764e–04	192	<00:00:01	0.03	< 1	12
256	65,536	4.9797e–05	387	<00:00:01	0.16	2	13
512	262,144	1.2494e–05	783	00:00:01	1.39	8	19
1024	1,048,576	3.1266e–06	1,581	00:00:19	18.88	32	44
2048	4,194,304	7.8019e–07	3,192	00:02:21	140.75	128	143
4096	16,777,216	1.9365e–07	6,452	00:18:43	1,123.34	512	536
8192	67,108,864	4.7374e–08	13,033	02:18:51	8,331.01	2,048	2,109
16384	268,435,456	1.1541e–08	26,316	18:24:11	66,251.23	8,192	8,401

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 1024^2 converts its value to units of MB, 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`. For the one case where multiple processes were needed, this number is summed across all running processes to get the total usage. For the runs that take under one second, the observed memory appears to be dominated by some system overhead, rather than reflecting the problem size directly.

In nearly all cases, the norms of the finite difference errors in Table 3.1 decrease by a factor of about 4 each time that the mesh is refined by a factor 2. This confirms that the finite difference method is second-order convergent, as predicted by the numerical theory for the finite difference method [7, 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 too much, because otherwise its solution would not be accurate enough and the norm of the finite difference error would not

show a second-order convergence behavior, as required by the theory. The good agreement between predicted and observed memory usage in the last two columns of the table indicates that the implementation of the code does not have any unexpected memory usage. The wall clock times and the memory usages for these serial runs indicate for which mesh resolutions this elliptic test problem becomes computationally challenging. Notice that the very fine meshes show very significant run times and memory usage; parallel computing clearly offers opportunities to decrease run times as well as to decrease memory usage per process by spreading the problem over the parallel processes.

The results for the finite difference error and the conjugate gradient iterations in Table 3.1 of this report agree essentially exactly with the corresponding results in Table 3.1 of [4]. This confirms that the parallel performance studies in the next section are practically relevant in that a correct solution of the test problem is computed. But the computational speed of the new serial results given in Table 3.1 are almost two times faster than those in Table 3.1 of [4]. For example, [4] reports 16,596.76 and 122,261.31 seconds for the cases $N = 8{,}192$ and $N = 16{,}384$, respectively, which are 2.0 and 1.8 times slower, respectively, than the results in Table 3.1 of this report. The obtained improvement in speed is due to only an improvement in the coding part not the MPI since for the serial runs none of the MPI commands are incorporated.

4 Performance Studies on tara with Non-Blocking MPI Communication Commands

To study the performance of non-blocking MPI communication commands on tara, we have run the non-blocking MPI version of our code for the test problem on a series of progressively finer meshes, resulting in progressively larger systems of linear equations with system dimensions ranging from about 1 million up to hundreds of millions of equations, with different numbers of nodes from 1 to 64 with different numbers of processes per node. After several runs for each case, Table 4.1 summarizes the maximum recorded run time results of our runs for 5 meshes with different numbers and arrangement of processes. Specifically, the upper-left entry of each sub-table with 1 process per node on 1 node represents the serial run of the code, which takes 19 seconds for the $1{,}024 \times 1{,}024$ mesh that results in a system of about 1 million linear equations to be solved. The lower-right entry of each sub-table lists the run using all cores of both quad-core processors on 64 nodes for a total of 512 parallel processes working together to solve the problem, which takes less than one second for this mesh. Results shown as 00:00:00 indicate that the observed wall clock time was less than 1 second for that case.

The summary results in Table 4.1 are arranged to study two key questions: (i) whether the code scales linearly to 64 nodes, which ascertains the quality of the InfiniBand interconnect network, and (ii) whether it is worthwhile to use multiple processors and cores on each node, which analyzes the quality of the architecture of the nodes and in turn guides the scheduling policy (whether it should be default to use all cores on a node or not).

(i) Reading along each row of Table 4.1, speedup in proportion to the number of nodes used is observable. This is to be discussed in detail in the following in terms of the number and pattern of parallel processes. As inherent to real experimentations, the results show some experimental variability with better-than-optimal results in some entries. But more remarkably, there is nearly optimal halving of the execution time even from 16 to 32 and from 32 to 64 nodes in the final columns of the table for the $4{,}096 \times 4{,}096$, $8{,}192 \times 8{,}192$, and $16{,}384 \times 16{,}384$ meshes. These excellent results successfully demonstrate the scalability of the algorithm and its implementation up to very large numbers of nodes as well as highlight the quality of the new quad-data rate InfiniBand interconnect network.

(ii) To analyze the effect of running 1, 2, 4, or 8 parallel processes per node, we compare the results column-wise in each sub-table. It is apparent that the execution time of each problem is in fact roughly halved with doubling the numbers of processes per node. This is an excellent result, as a slow-down is more typical traditionally on multi-processor nodes. These results confirm that it is not just effective to use both processors on each node, but also to use all cores of each quad-core processor simultaneously. Roughly, this shows that the architecture of the IBM nodes purchased in 2009 has sufficient capacity in all vital components to avoid creating any bottlenecks in accessing the memory of the node that is shared by the processes. These results thus justify the purchase of compute nodes with two processors (as opposed to one processor) and of multi-core processors (as opposed to single-core processors). Moreover, these results guide the scheduling policy implemented on the cluster: On the one hand, it is not disadvantageous to run several of serial jobs simultaneously on one node. On the other hand, for jobs using several nodes, it is advantageous to make use of all cores on all nodes reserved by the scheduler.

Table 4.1: Wall clock time in HH:MM:SS for the code incorporating the non-blocking MPI communication commands running on tara for the solution of elliptic problems on $N \times N$ meshes using 1, 2, 4, 8, 16, 32, and 64 compute nodes with 1, 2, 4 and 8 processes per node.

(a) Mesh resolution $N \times N = 1024 \times 1024$, system dimension 1,048,576

	1 node	2 nodes	4 nodes	8 nodes	16 nodes	32 nodes	64 nodes
1 process per node	00:00:19	00:00:09	00:00:04	00:00:02	00:00:01	00:00:01	00:00:01
2 processes per node	00:00:09	00:00:04	00:00:02	00:00:01	00:00:01	00:00:01	00:00:01
4 processes per node	00:00:06	00:00:02	00:00:01	00:00:00	00:00:00	00:00:00	00:00:00
8 processes per node	00:00:04	00:00:01	00:00:01	00:00:00	00:00:00	00:00:00	00:00:00

(b) Mesh resolution $N \times N = 2048 \times 2048$, system dimension 4,194,304

	1 node	2 nodes	4 nodes	8 nodes	16 nodes	32 nodes	64 nodes
1 process per node	00:02:21	00:01:22	00:00:43	00:00:19	00:00:08	00:00:05	00:00:03
2 processes per node	00:01:24	00:00:42	00:00:19	00:00:08	00:00:04	00:00:03	00:00:02
4 processes per node	00:00:49	00:00:30	00:00:15	00:00:05	00:00:02	00:00:01	00:00:01
8 processes per node	00:00:41	00:00:21	00:00:09	00:00:03	00:00:01	00:00:01	00:00:01

(c) Mesh resolution $N \times N = 4096 \times 4096$, system dimension 16,777,216

	1 node	2 nodes	4 nodes	8 nodes	16 nodes	32 nodes	64 nodes
1 process per node	00:18:43	00:09:45	00:06:21	00:03:04	00:01:35	00:00:40	00:00:18
2 processes per node	00:10:28	00:05:40	00:02:51	00:01:26	00:00:39	00:00:17	00:00:10
4 processes per node	00:06:35	00:03:33	00:01:58	00:00:57	00:00:35	00:00:11	00:00:05
8 processes per node	00:05:30	00:02:46	00:01:24	00:00:42	00:00:19	00:00:07	00:00:03

(d) Mesh resolution $N \times N = 8192 \times 8192$, system dimension 67,108,864

	1 node	2 nodes	4 nodes	8 nodes	16 nodes	32 nodes	64 nodes
1 process per node	02:18:51	01:23:50	00:50:37	00:25:25	00:12:37	00:06:40	00:03:41
2 processes per node	01:22:42	00:43:16	00:22:28	00:11:10	00:05:47	00:02:56	00:01:22
4 processes per node	00:51:57	00:28:10	00:15:23	00:09:02	00:03:53	00:02:44	00:01:17
8 processes per node	00:44:46	00:22:33	00:11:27	00:05:44	00:02:56	00:01:29	00:00:40

(e) Mesh resolution $N \times N = 16384 \times 16384$, system dimension 268,435,456

	1 node	2 nodes	4 nodes	8 nodes	16 nodes	32 nodes	64 nodes
1 process per node	18:24:11	10:33:01	06:23:30	03:35:18	01:47:17	00:55:02	00:28:07
2 processes per node	10:32:19	05:16:17	02:48:07	01:32:45	00:46:40	00:23:26	00:11:50
4 processes per node	06:48:58	03:43:56	01:56:35	00:59:25	00:30:30	00:17:31	00:09:08
8 processes per node	06:06:07	03:03:01	01:32:33	00:46:00	00:23:17	00:11:45	00:06:07

In essence, the run times for the finer meshes observed for serial runs in Table 3.1 brought out one key motivation for parallel computing: The run times for a problem of a given, fixed size can be potentially dramatically reduced by spreading the work across a group of parallel processes. More precisely, the ideal behavior of code for a fixed problem size using p parallel processes is to do the job p times faster. 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 one to p processes, whose optimal value is bounded by p. The efficiency defined as $E_p = S_p/p$ 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 1.1 in the report [4] using a completely separate code shows the corresponding results as Table 4.1 in this report using a new implementation of the algorithm. The serial results (1 process on 1 node in Table 4.1) are about twice as fast with the new code, as already noted in the context of Table 3.1. For all other combinations of processes per node and nodes, it is also true that the new code is faster, in many cases significantly. However, the speedup decreases from its maximum of about 2.0 for the serial code, when the number of processes per node increase as well as when the number of nodes increase.

To further study the performance of the new code implementation, we reformat the obtained run times resulting in four groups of tables to study the speedup and efficiency of the code by the number of parallel processes. Accordingly, after several runs for each case and recording the maximum run time, Tables 4.2, 4.3, 4.4, and 4.5 tabulate the results for several runs, with 1, 2, 4, and 8 processes per node (whenever possible), respectively. The corresponding tables in [4] are Tables 4.1, 4.2, 4.3, and 4.5, respectively; notice the different numbering schemes for corresponding results. Comparing Tables 4.2 and 4.3 of this report to Tables 4.1 and 4.2 of [4], respectively, and also using the associated plots in corresponding figures, shows comparable observed speedup and efficiency.

However, comparing Tables 4.4 and 4.5 of this report to Tables 4.3 and 4.5 of [4], respectively, and also using the associated plots in corresponding figures, shows degraded speedup and efficiency for the new code; but recall again that their absolute run times are always better than those of the code using for [4]. These results demonstrate that subtle differences in ordering of loops with or without if statements inside of them can make significant difference in run times.

Table 4.2: Performance of non-blocking MPI communication on tara by number of processes used with 1 process per node, except for $p = 128$ which uses 2 processes per node, $p = 256$ which uses 4 processes per node, and $p = 512$ which uses 8 processes per node.

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

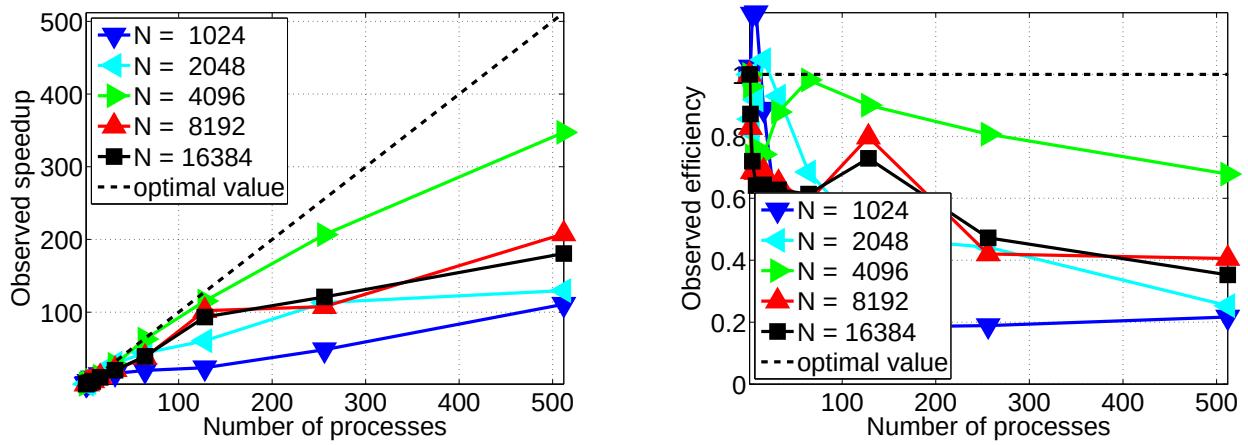
N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$	$p = 256$	$p = 512$
1024	00:00:19	00:00:09	00:00:04	00:00:02	00:00:01	00:00:01	00:00:01	00:00:01	00:00:00	00:00:00
2048	00:02:21	00:01:22	00:00:43	00:00:19	00:00:08	00:00:05	00:00:03	00:00:02	00:00:01	00:00:01
4096	00:18:43	00:09:45	00:06:21	00:03:04	00:01:35	00:00:40	00:00:18	00:00:10	00:00:05	00:00:03
8192	02:18:51	01:23:50	00:50:37	00:25:25	00:12:37	00:06:40	00:03:41	00:01:22	00:01:17	00:00:40
16384	18:24:11	10:33:01	06:23:30	03:35:18	01:47:17	00:55:02	00:28:07	00:11:50	00:09:08	00:06:07

(b) Observed speedup S_p

N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$	$p = 256$	$p = 512$
1024	1.0000	2.0478	4.7778	9.5945	14.2054	15.6180	19.8336	23.5534	48.3816	110.8834
2048	1.0000	1.7108	3.2457	7.3422	16.7593	29.7518	43.8392	60.6864	113.2068	129.6279
4096	1.0000	1.9213	2.9491	6.0913	11.8661	28.1120	62.8595	115.2729	206.6333	347.2978
8192	1.0000	1.6564	2.7433	5.4640	11.0072	20.8098	37.7515	102.0918	107.5759	207.6313
16384	1.0000	1.7443	2.8792	5.1286	10.2916	20.0657	39.2785	93.3231	120.9622	180.5992

(c) Observed efficiency E_p

N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$	$p = 256$	$p = 512$
1024	1.0000	1.0239	1.1945	1.1993	0.8878	0.4881	0.3099	0.1840	0.1890	0.2166
2048	1.0000	0.8554	0.8114	0.9178	1.0475	0.9297	0.6850	0.4741	0.4422	0.2532
4096	1.0000	0.9606	0.7373	0.7614	0.7416	0.8785	0.9822	0.9006	0.8072	0.6783
8192	1.0000	0.8282	0.6858	0.6830	0.6879	0.6503	0.5899	0.7976	0.4202	0.4055
16384	1.0000	0.8722	0.7198	0.6411	0.6432	0.6271	0.6137	0.7291	0.4725	0.3527



Observed speedup S_p Observed efficiency E_p

Figure 4.1: Performance of non-blocking MPI communication on tara by number of processes used with 1 process per node, except for $p = 128$ which uses 2 processes per node, $p = 256$ which uses 4 processes per node, and $p = 512$ which uses 8 processes per node.

Table 4.3: Performance of non-blocking MPI communication on tara by number of processes used with 2 processes per node, except for $p = 1$ which uses 1 process per node, $p = 256$ which uses 4 processes per node, and $p = 512$ which uses 8 processes per node.

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

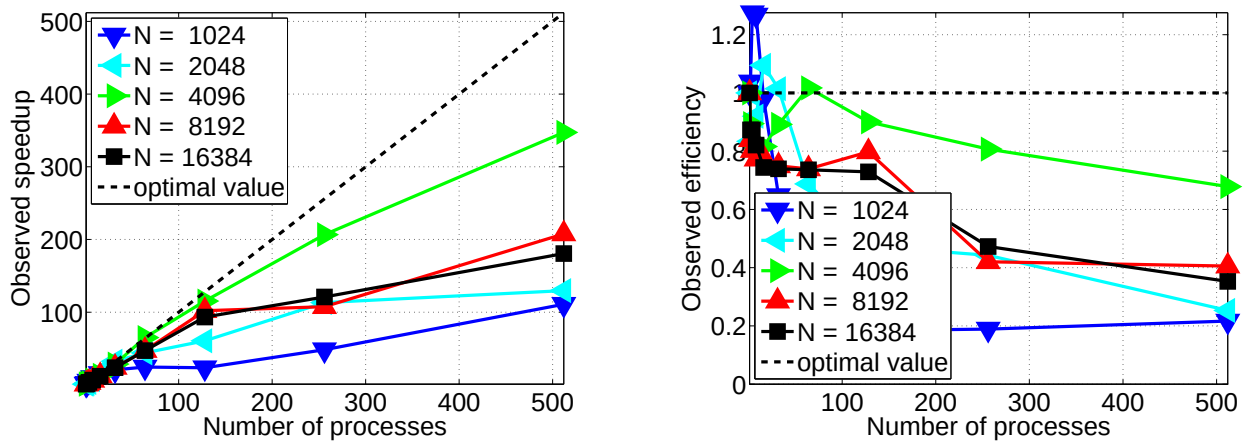
N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$	$p = 256$	$p = 512$
1024	00:00:19	00:00:09	00:00:04	00:00:02	00:00:01	00:00:01	00:00:01	00:00:01	00:00:00	00:00:00
2048	00:02:21	00:01:24	00:00:42	00:00:19	00:00:08	00:00:04	00:00:03	00:00:02	00:00:01	00:00:01
4096	00:18:43	00:10:28	00:05:40	00:02:51	00:01:26	00:00:39	00:00:17	00:00:10	00:00:05	00:00:03
8192	02:18:51	01:22:42	00:43:16	00:22:28	00:11:10	00:05:47	00:02:56	00:01:22	00:01:17	00:00:40
16384	18:24:11	10:32:19	05:16:17	02:48:07	01:32:45	00:46:40	00:23:26	00:11:50	00:09:08	00:06:07

(b) Observed speedup S_p

N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$	$p = 256$	$p = 512$
1024	1.0000	2.0756	5.1020	10.1391	15.6894	20.7131	24.4699	23.5534	48.3816	110.8834
2048	1.0000	1.6674	3.3299	7.3610	17.5143	32.4615	43.9981	60.6864	113.2068	129.6279
4096	1.0000	1.7890	3.3037	6.5779	13.0559	28.5507	65.0924	115.2729	206.6333	347.2978
8192	1.0000	1.6789	3.2090	6.1789	12.4275	23.9796	47.3083	102.0918	107.5759	207.6313
16384	1.0000	1.7463	3.4911	6.5680	11.9045	23.6653	47.1101	93.3231	120.9622	180.5992

(c) Observed efficiency E_p

N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$	$p = 256$	$p = 512$
1024	1.0000	1.0378	1.2755	1.2674	0.9806	0.6473	0.3823	0.1840	0.1890	0.2166
2048	1.0000	0.8337	0.8325	0.9201	1.0946	1.0144	0.6875	0.4741	0.4422	0.2532
4096	1.0000	0.8945	0.8259	0.8222	0.8160	0.8922	1.0171	0.9006	0.8072	0.6783
8192	1.0000	0.8394	0.8023	0.7724	0.7767	0.7494	0.7392	0.7976	0.4202	0.4055
16384	1.0000	0.8731	0.8728	0.8210	0.7440	0.7395	0.7361	0.7291	0.4725	0.3527



Observed speedup S_p Observed efficiency E_p

Figure 4.2: Performance of non-blocking MPI communication on tara by number of processes used with 2 processes per node, except for $p = 1$ which uses 1 process per node, $p = 256$ which uses 4 processes per node, and $p = 512$ which uses 8 processes per node.

Table 4.4: Performance of non-blocking MPI communication on tara by number of processes used with 4 processes per node, except for $p = 1$ which uses 1 process per node, $p = 2$ which uses 2 processes per node, and $p = 512$ which uses 8 processes per node.

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

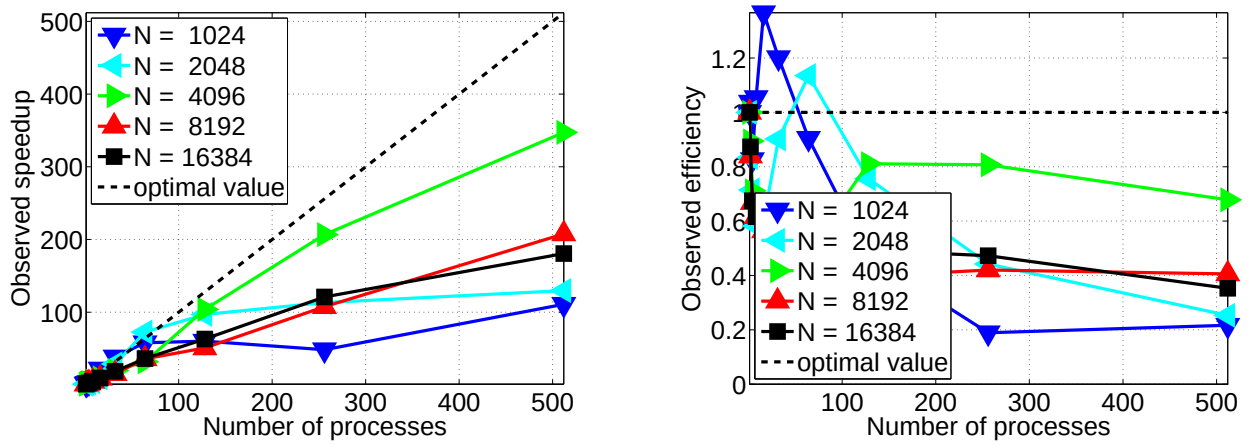
N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$	$p = 256$	$p = 512$
1024	00:00:19	00:00:09	00:00:06	00:00:02	00:00:01	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00
2048	00:02:21	00:01:24	00:00:49	00:00:30	00:00:15	00:00:05	00:00:02	00:00:01	00:00:01	00:00:01
4096	00:18:43	00:10:28	00:06:35	00:03:33	00:01:58	00:00:57	00:00:35	00:00:11	00:00:05	00:00:03
8192	02:18:51	01:22:42	00:51:57	00:28:10	00:15:23	00:09:02	00:03:53	00:02:44	00:01:17	00:00:40
16384	18:24:11	10:32:19	06:48:58	03:43:56	01:56:35	00:59:25	00:30:30	00:17:31	00:09:08	00:06:07

(b) Observed speedup S_p

N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$	$p = 256$	$p = 512$
1024	1.0000	2.0756	3.3051	8.4360	21.8697	38.4454	57.9093	60.1108	48.3816	110.8834
2048	1.0000	1.6674	2.8596	4.6569	9.6490	28.8742	72.6564	96.8552	113.2068	129.6279
4096	1.0000	1.7890	2.8459	5.2819	9.5147	19.7122	31.8089	103.8170	206.6333	347.2978
8192	1.0000	1.6789	2.6724	4.9290	9.0217	15.3799	35.7861	50.7585	107.5759	207.6313
16384	1.0000	1.7463	2.6999	4.9309	9.4717	18.5816	36.2067	63.0422	120.9622	180.5992

(c) Observed efficiency E_p

N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$	$p = 256$	$p = 512$
1024	1.0000	1.0378	0.8263	1.0545	1.3669	1.2014	0.9048	0.4696	0.1890	0.2166
2048	1.0000	0.8337	0.7149	0.5821	0.6031	0.9023	1.1353	0.7567	0.4422	0.2532
4096	1.0000	0.8945	0.7115	0.6602	0.5947	0.6160	0.4970	0.8111	0.8072	0.6783
8192	1.0000	0.8394	0.6681	0.6161	0.5639	0.4806	0.5592	0.3966	0.4202	0.4055
16384	1.0000	0.8731	0.6750	0.6164	0.5920	0.5807	0.5657	0.4925	0.4725	0.3527



Observed speedup S_p Observed efficiency E_p

Figure 4.3: Performance of non-blocking MPI communication on tara by number of processes used with 4 processes per node, except for $p = 1$ which uses 1 process per node, $p = 2$ which uses 2 processes per node, and $p = 512$ which uses 8 processes per node.

Table 4.5: Performance of non-blocking MPI communication on tara by number of processes used with 8 processes per node, except for $p = 1$ which uses 1 process per node, $p = 2$ which uses 2 processes per node, and $p = 4$ which uses 4 processes per node.

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

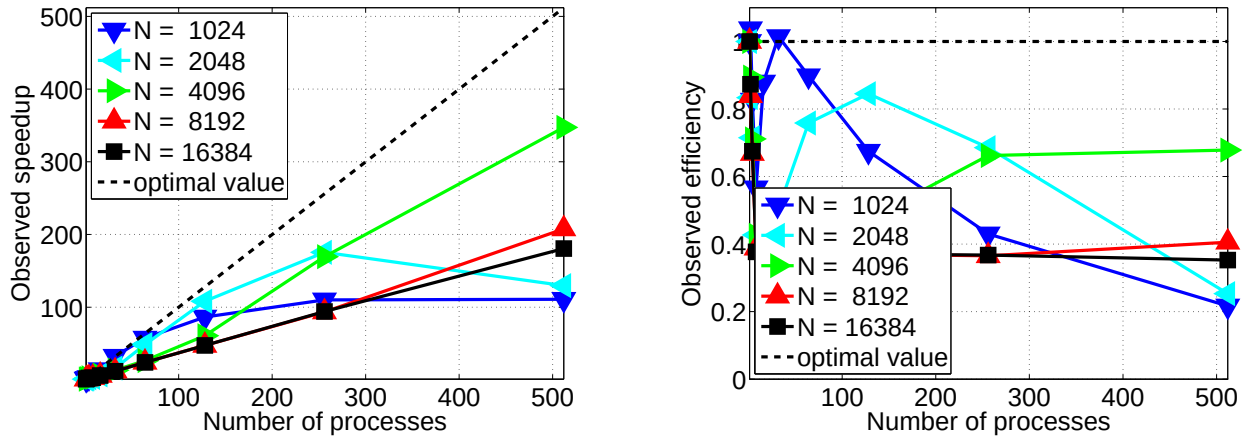
N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$	$p = 256$	$p = 512$
1024	00:00:19	00:00:09	00:00:06	00:00:04	00:00:01	00:00:01	00:00:00	00:00:00	00:00:00	00:00:00
2048	00:02:21	00:01:24	00:00:49	00:00:41	00:00:21	00:00:09	00:00:03	00:00:01	00:00:01	00:00:01
4096	00:18:43	00:10:28	00:06:35	00:05:30	00:02:46	00:01:24	00:00:42	00:00:19	00:00:07	00:00:03
8192	02:18:51	01:22:42	00:51:57	00:44:46	00:22:33	00:11:27	00:05:44	00:02:56	00:01:29	00:00:40
16384	18:24:11	10:32:19	06:48:58	06:06:07	03:03:01	01:32:33	00:46:00	00:23:17	00:11:45	00:06:07

(b) Observed speedup S_p

N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$	$p = 256$	$p = 512$
1024	1.0000	2.0756	3.3051	4.5320	14.0793	32.4655	57.5018	86.4067	110.0752	110.8834
2048	1.0000	1.6674	2.8596	3.4148	6.8216	16.1720	48.5512	108.1860	175.5163	129.6279
4096	1.0000	1.7890	2.8459	3.4069	6.7591	13.3113	26.4742	60.6992	169.5317	347.2978
8192	1.0000	1.6789	2.6724	3.1021	6.1556	12.1351	24.1976	47.3406	93.4859	207.6313
16384	1.0000	1.7463	2.6999	3.0159	6.0332	11.9313	24.0014	47.4136	94.0371	180.5992

(c) Observed efficiency E_p

N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$	$p = 256$	$p = 512$
1024	1.0000	1.0378	0.8263	0.5665	0.8800	1.0145	0.8985	0.6751	0.4300	0.2166
2048	1.0000	0.8337	0.7149	0.4268	0.4263	0.5054	0.7586	0.8452	0.6856	0.2532
4096	1.0000	0.8945	0.7115	0.4259	0.4224	0.4160	0.4137	0.4742	0.6622	0.6783
8192	1.0000	0.8394	0.6681	0.3878	0.3847	0.3792	0.3781	0.3698	0.3652	0.4055
16384	1.0000	0.8731	0.6750	0.3770	0.3771	0.3729	0.3750	0.3704	0.3673	0.3527



Observed speedup S_p Observed efficiency E_p

Figure 4.4: Performance of non-blocking MPI communication on tara by number of processes used with 8 processes per node, except for $p = 1$ which uses 1 process per node, $p = 2$ which uses 2 processes per node, and $p = 4$ which uses 4 processes per node.

5 Performance Studies on tara with Blocking MPI Communication Commands

The goal of this section is to compare the performance of blocking and non-blocking MPI communication commands on the cluster tara. This study is analogous to the previous section, but this time the blocking MPI communication commands are used instead of the non-blocking ones. The results reported in Table 3.1 will be unchanged since the serial runs in effect would not need any communication between processes as only a single computational process will be available.

Table 5.1 in the following summarizes the exhaustive timing results for our new studies, analogous to Table 4.1 in the previous section. Reading the data row-wise (varying number of nodes) or column-wise (varying processes per node), we again observe excellent scalability for small number of processes, roughly 64 processes.

Tables 5.2 through 5.5 and their corresponding figures show detailed performance results by number of parallel process using blocking MPI commands; they contrast to Tables 4.2 through 4.5 and their corresponding figures in the previous section using non-blocking MPI commands. For small number of processes acting on any arrangement of 1, 2, 4 and 8 processes per node, the results are quite similar. However, starting from using 128 processes, a close inspection reveals a considerably slower performance of the code when using blocking MPI commands. Comparison of all corresponding figures, both the speedup and the efficiency plots, brings out this difference and highlight that the result depends on the number of processes, for any arrangement of nodes and processes per node. As the raw timing results show, the speed advantage is sometimes a factor 2.0 or similar, but ranges up to an extreme case of a factor 5.0 or more for 512 processes and the finer meshes. In summary, we see that the code using non-blocking MPI commands is faster than using blocking MPI commands for any number of processes, but that it may be significantly faster when using large numbers or processes.

Table 5.1: Wall clock time in HH:MM:SS for the code incorporating the blocking MPI communication commands running on tara for the solution of elliptic problems on $N \times N$ meshes using 1, 2, 4, 8, 16, 32, and 64 compute nodes with 1, 2, 4 and 8 processes per node.

(a) Mesh resolution $N \times N = 1024 \times 1024$, system dimension 1,048,576

	1 node	2 nodes	4 nodes	8 nodes	16 nodes	32 nodes	64 nodes
1 process per node	00:00:19	00:00:09	00:00:04	00:00:02	00:00:02	00:00:02	00:00:02
2 processes per node	00:00:09	00:00:04	00:00:02	00:00:02	00:00:02	00:00:02	00:00:05
4 processes per node	00:00:07	00:00:02	00:00:01	00:00:01	00:00:01	00:00:02	00:00:04
8 processes per node	00:00:04	00:00:01	00:00:01	00:00:01	00:00:02	00:00:03	00:00:07

(b) Mesh resolution $N \times N = 2048 \times 2048$, system dimension 4,194,304

	1 node	2 nodes	4 nodes	8 nodes	16 nodes	32 nodes	64 nodes
1 process per node	00:02:21	00:01:21	00:00:42	00:00:22	00:00:10	00:00:08	00:00:11
2 processes per node	00:01:24	00:00:43	00:00:19	00:00:09	00:00:06	00:00:08	00:00:11
4 processes per node	00:00:49	00:00:27	00:00:18	00:00:06	00:00:05	00:00:07	00:00:12
8 processes per node	00:00:41	00:00:22	00:00:10	00:00:06	00:00:06	00:00:10	00:00:20

(c) Mesh resolution $N \times N = 4096 \times 4096$, system dimension 16,777,216

	1 node	2 nodes	4 nodes	8 nodes	16 nodes	32 nodes	64 nodes
1 process per node	00:18:43	00:11:15	00:06:32	00:02:51	00:01:37	00:00:54	00:00:39
2 processes per node	00:10:18	00:05:41	00:02:54	00:01:30	00:00:46	00:00:32	00:00:36
4 processes per node	00:06:41	00:03:14	00:02:03	00:01:01	00:00:42	00:00:31	00:00:41
8 processes per node	00:05:33	00:02:49	00:01:31	00:00:54	00:00:40	00:00:43	00:01:13

(d) Mesh resolution $N \times N = 8192 \times 8192$, system dimension 67,108,864

	1 node	2 nodes	4 nodes	8 nodes	16 nodes	32 nodes	64 nodes
1 process per node	02:18:51	01:21:34	00:43:17	00:22:28	00:13:02	00:06:53	00:04:23
2 processes per node	01:19:47	00:44:48	00:22:01	00:11:42	00:06:10	00:03:46	00:03:08
4 processes per node	00:55:47	00:27:40	00:14:48	00:08:01	00:04:36	00:04:14	00:04:03
8 processes per node	00:45:05	00:22:42	00:11:58	00:06:54	00:05:00	00:05:42	00:07:38

(e) Mesh resolution $N \times N = 16384 \times 16384$, system dimension 268,435,456

	1 node	2 nodes	4 nodes	8 nodes	16 nodes	32 nodes	64 nodes
1 process per node	18:24:11	09:50:28	06:35:50	02:56:17	01:44:01	00:54:44	00:29:60
2 processes per node	10:12:46	05:18:37	03:04:40	01:33:00	00:46:45	00:25:04	00:17:07
4 processes per node	06:49:13	03:50:17	01:57:49	01:02:01	00:32:26	00:20:39	00:18:53
8 processes per node	06:20:18	03:07:54	01:35:54	00:51:06	00:30:40	00:25:43	00:34:37

Table 5.2: Performance of blocking MPI communication on tara by number of processes used with 1 process per node, except for $p = 128$ which uses 2 processes per node, $p = 256$ which uses 4 processes per node, and $p = 512$ which uses 8 processes per node.

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

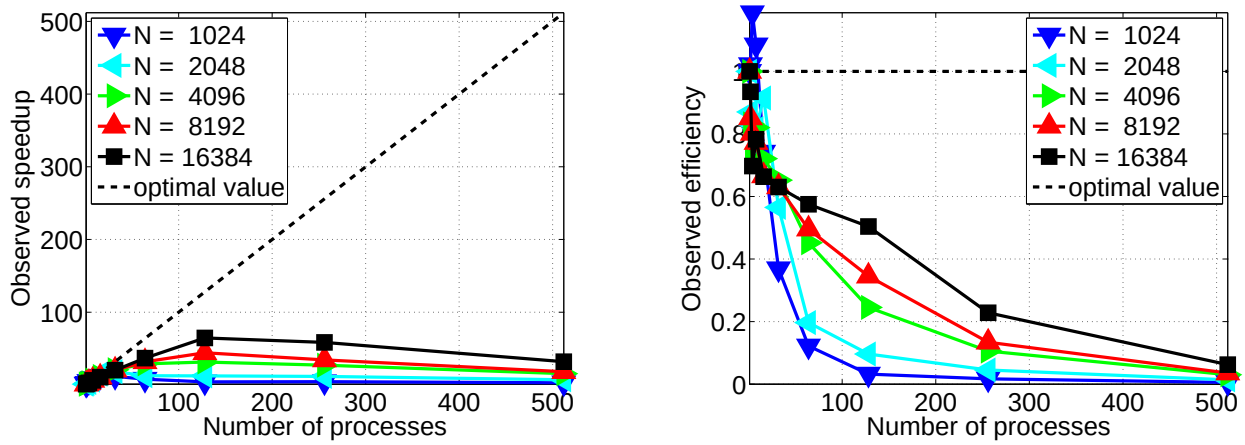
N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$	$p = 256$	$p = 512$
1024	00:00:19	00:00:09	00:00:04	00:00:02	00:00:02	00:00:02	00:00:02	00:00:05	00:00:04	00:00:07
2048	00:02:21	00:01:21	00:00:42	00:00:22	00:00:10	00:00:08	00:00:11	00:00:11	00:00:12	00:00:20
4096	00:18:43	00:11:15	00:06:32	00:02:51	00:01:37	00:00:54	00:00:39	00:00:36	00:00:41	00:01:13
8192	02:18:51	01:21:34	00:43:17	00:22:28	00:13:02	00:06:53	00:04:23	00:03:08	00:04:03	00:07:38
16384	18:24:11	09:50:28	06:35:50	02:56:17	01:44:01	00:54:44	00:29:60	00:17:07	00:18:53	00:34:37

(b) Observed speedup S_p

N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$	$p = 256$	$p = 512$
1024	1.0000	2.0436	4.7509	8.6780	11.8713	11.8082	7.8064	4.1192	4.3181	2.8875
2048	1.0000	1.7407	3.3145	6.5368	14.6378	18.0810	12.6030	12.3292	11.5691	7.1043
4096	1.0000	1.6650	2.8664	6.5636	11.5364	20.8757	28.9331	31.4201	27.0805	15.4527
8192	1.0000	1.7024	3.2077	6.1821	10.6569	20.1866	31.7347	44.2010	34.3433	18.1721
16384	1.0000	1.8700	2.7895	6.2637	10.6154	20.1726	36.8082	64.5344	58.4585	31.8928

(c) Observed efficiency E_p

N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$	$p = 256$	$p = 512$
1024	1.0000	1.0218	1.1877	1.0848	0.7420	0.3690	0.1220	0.0322	0.0169	0.0056
2048	1.0000	0.8704	0.8286	0.8171	0.9149	0.5650	0.1969	0.0963	0.0452	0.0139
4096	1.0000	0.8325	0.7166	0.8205	0.7210	0.6524	0.4521	0.2455	0.1058	0.0302
8192	1.0000	0.8512	0.8019	0.7728	0.6661	0.6308	0.4959	0.3453	0.1342	0.0355
16384	1.0000	0.9350	0.6974	0.7830	0.6635	0.6304	0.5751	0.5042	0.2284	0.0623



Observed speedup S_p Observed efficiency E_p

Figure 5.1: Performance of blocking MPI communication on tara by number of processes used with 1 process per node, except for $p = 128$ which uses 2 processes per node, $p = 256$ which uses 4 processes per node, and $p = 512$ which uses 8 processes per node.

Table 5.3: Performance of blocking MPI communication on tara by number of processes used with 2 processes per node, except for $p = 1$ which uses 1 process per node, $p = 256$ which uses 4 processes per node, and $p = 512$ which uses 8 processes per node.

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

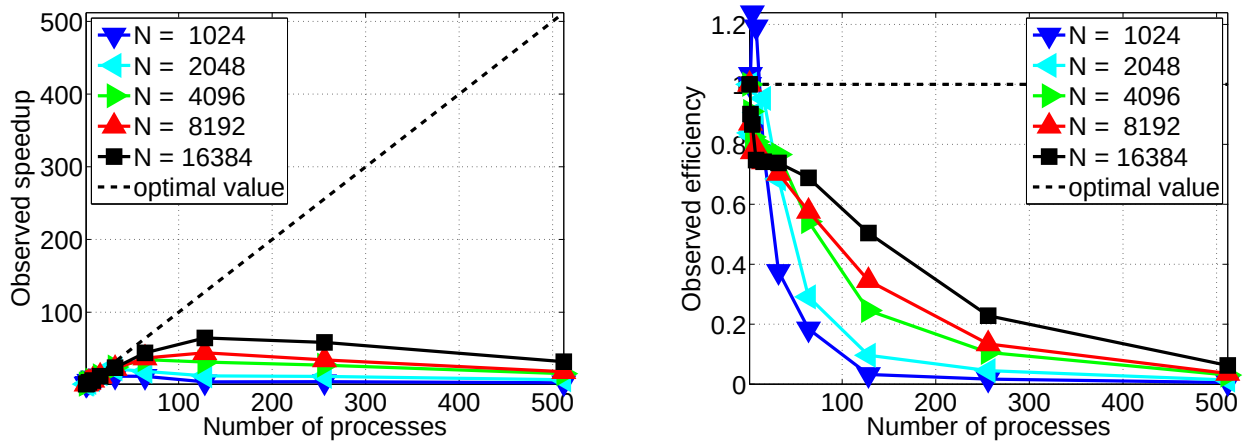
N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$	$p = 256$	$p = 512$
1024	00:00:19	00:00:09	00:00:04	00:00:02	00:00:02	00:00:02	00:00:02	00:00:05	00:00:04	00:00:07
2048	00:02:21	00:01:24	00:00:43	00:00:19	00:00:09	00:00:06	00:00:08	00:00:11	00:00:12	00:00:20
4096	00:18:43	00:10:18	00:05:41	00:02:54	00:01:30	00:00:46	00:00:32	00:00:36	00:00:41	00:01:13
8192	02:18:51	01:19:47	00:44:48	00:22:01	00:11:42	00:06:10	00:03:46	00:03:08	00:04:03	00:07:38
16384	18:24:11	10:12:46	05:18:37	03:04:40	01:33:00	00:46:45	00:25:04	00:17:07	00:18:53	00:34:37

(b) Observed speedup S_p

N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$	$p = 256$	$p = 512$
1024	1.0000	2.0673	4.9567	9.5368	12.4073	12.0363	11.7986	4.1192	4.3181	2.8875
2048	1.0000	1.6760	3.3056	7.2488	15.2532	21.9114	18.6431	12.3292	11.5691	7.1043
4096	1.0000	1.8190	3.2987	6.4650	12.5138	24.5101	34.7019	31.4201	27.0805	15.4527
8192	1.0000	1.7403	3.0993	6.3075	11.8609	22.4949	36.8857	44.2010	34.3433	18.1721
16384	1.0000	1.8020	3.4656	5.9793	11.8721	23.6155	44.0411	64.5344	58.4585	31.8928

(c) Observed efficiency E_p

N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$	$p = 256$	$p = 512$
1024	1.0000	1.0336	1.2392	1.1921	0.7755	0.3761	0.1844	0.0322	0.0169	0.0056
2048	1.0000	0.8380	0.8264	0.9061	0.9533	0.6847	0.2913	0.0963	0.0452	0.0139
4096	1.0000	0.9095	0.8247	0.8081	0.7821	0.7659	0.5422	0.2455	0.1058	0.0302
8192	1.0000	0.8702	0.7748	0.7884	0.7413	0.7030	0.5763	0.3453	0.1342	0.0355
16384	1.0000	0.9010	0.8664	0.7474	0.7420	0.7380	0.6881	0.5042	0.2284	0.0623



Observed speedup S_p Observed efficiency E_p

Figure 5.2: Performance of blocking MPI communication on tara by number of processes used with 2 processes per node, except for $p = 1$ which uses 1 process per node, $p = 256$ which uses 4 processes per node, and $p = 512$ which uses 8 processes per node.

Table 5.4: Performance of blocking MPI communication on tara by number of processes used with 4 processes per node, except for $p = 1$ which uses 1 process per node, $p = 2$ which uses 2 processes per node, and $p = 512$ which uses 8 processes per node.

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

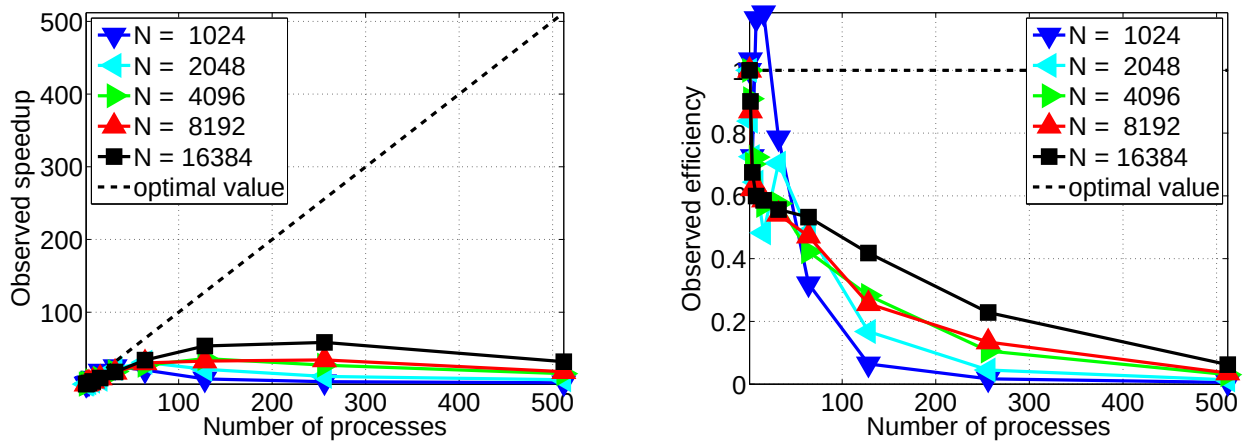
N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$	$p = 256$	$p = 512$
1024	00:00:19	00:00:09	00:00:07	00:00:02	00:00:01	00:00:01	00:00:01	00:00:02	00:00:04	00:00:07
2048	00:02:21	00:01:24	00:00:49	00:00:27	00:00:18	00:00:06	00:00:05	00:00:07	00:00:12	00:00:20
4096	00:18:43	00:10:18	00:06:41	00:03:14	00:02:03	00:01:01	00:00:42	00:00:31	00:00:41	00:01:13
8192	02:18:51	01:19:47	00:55:47	00:27:40	00:14:48	00:08:01	00:04:36	00:04:14	00:04:03	00:07:38
16384	18:24:11	10:12:46	06:49:13	03:50:17	01:57:49	01:02:01	00:32:26	00:20:39	00:18:53	00:34:37

(b) Observed speedup S_p

N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$	$p = 256$	$p = 512$
1024	1.0000	2.0673	2.9012	9.3184	18.9362	25.1171	20.4717	8.2387	4.3181	2.8875
2048	1.0000	1.6760	2.8988	5.1483	7.7144	22.5308	31.0336	21.4843	11.5691	7.1043
4096	1.0000	1.8190	2.8044	5.7860	9.1125	18.4090	27.0180	36.1795	27.0805	15.4527
8192	1.0000	1.7403	2.4888	5.0196	9.3855	17.3115	30.1706	32.7799	34.3433	18.1721
16384	1.0000	1.8020	2.6983	4.7949	9.3716	17.8032	34.0535	53.4584	58.4585	31.8928

(c) Observed efficiency E_p

N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$	$p = 256$	$p = 512$
1024	1.0000	1.0336	0.7253	1.1648	1.1835	0.7849	0.3199	0.0644	0.0169	0.0056
2048	1.0000	0.8380	0.7247	0.6435	0.4822	0.7041	0.4849	0.1678	0.0452	0.0139
4096	1.0000	0.9095	0.7011	0.7233	0.5695	0.5753	0.4222	0.2827	0.1058	0.0302
8192	1.0000	0.8702	0.6222	0.6274	0.5866	0.5410	0.4714	0.2561	0.1342	0.0355
16384	1.0000	0.9010	0.6746	0.5994	0.5857	0.5563	0.5321	0.4176	0.2284	0.0623



Observed speedup S_p Observed efficiency E_p

Figure 5.3: Performance of blocking MPI communication on tara by number of processes used with 4 processes per node, except for $p = 1$ which uses 1 process per node, $p = 2$ which uses 2 processes per node, and $p = 512$ which uses 8 processes per node.

Table 5.5: Performance of blocking MPI communication on tara by number of processes used with 8 processes per node, except for $p = 1$ which uses 1 process per node, $p = 2$ which uses 2 processes per node, and $p = 4$ which uses 4 processes per node.

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

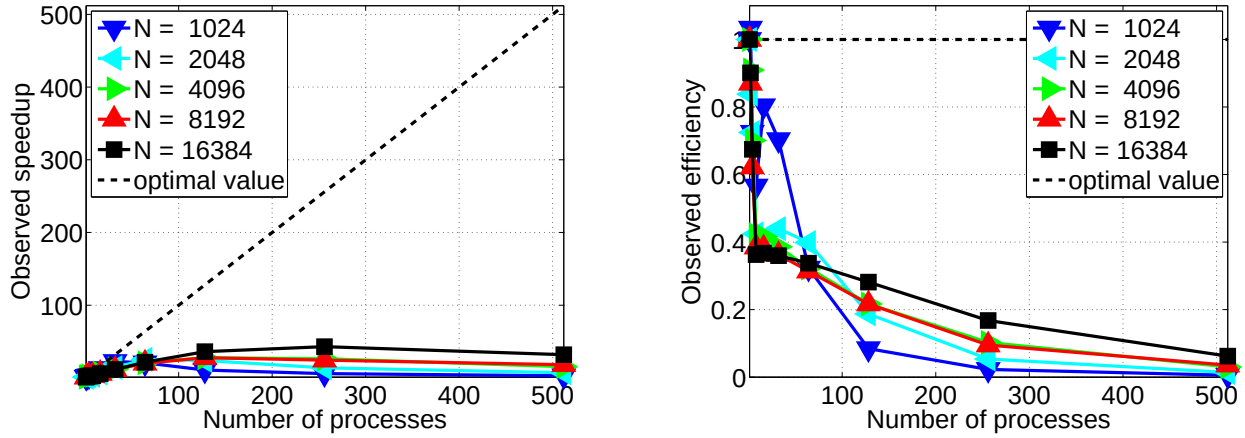
N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$	$p = 256$	$p = 512$
1024	00:00:19	00:00:09	00:00:07	00:00:04	00:00:01	00:00:01	00:00:01	00:00:02	00:00:03	00:00:07
2048	00:02:21	00:01:24	00:00:49	00:00:41	00:00:22	00:00:10	00:00:06	00:00:06	00:00:10	00:00:20
4096	00:18:43	00:10:18	00:06:41	00:05:33	00:02:49	00:01:31	00:00:54	00:00:40	00:00:43	00:01:13
8192	02:18:51	01:19:47	00:55:47	00:45:05	00:22:42	00:11:58	00:06:54	00:05:00	00:05:42	00:07:38
16384	18:24:11	10:12:46	06:49:13	06:20:18	03:07:54	01:35:54	00:51:06	00:30:40	00:25:43	00:34:37

(b) Observed speedup S_p

N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$	$p = 256$	$p = 512$
1024	1.0000	2.0673	2.9012	4.5282	12.8586	22.5093	20.6651	10.8090	5.8529	2.8875
2048	1.0000	1.6760	2.8988	3.3957	6.5054	14.0736	25.5167	23.9379	13.7896	7.1043
4096	1.0000	1.8190	2.8044	3.3745	6.6621	12.3505	20.6816	27.7392	26.3148	15.4527
8192	1.0000	1.7403	2.4888	3.0803	6.1185	11.6024	20.0994	27.7469	24.3668	18.1721
16384	1.0000	1.8020	2.6983	2.9035	5.8764	11.5145	21.6090	36.0138	42.9504	31.8928

(c) Observed efficiency E_p

N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$	$p = 256$	$p = 512$
1024	1.0000	1.0336	0.7253	0.5660	0.8037	0.7034	0.3229	0.0844	0.0229	0.0056
2048	1.0000	0.8380	0.7247	0.4245	0.4066	0.4398	0.3987	0.1870	0.0539	0.0139
4096	1.0000	0.9095	0.7011	0.4218	0.4164	0.3860	0.3231	0.2167	0.1028	0.0302
8192	1.0000	0.8702	0.6222	0.3850	0.3824	0.3626	0.3141	0.2168	0.0952	0.0355
16384	1.0000	0.9010	0.6746	0.3629	0.3673	0.3598	0.3376	0.2814	0.1678	0.0623



Observed speedup S_p Observed efficiency E_p

Figure 5.4: Performance of blocking MPI communication on tara by number of processes used with 8 processes per node, except for $p = 1$ which uses 1 process per node, $p = 2$ which uses 2 processes per node, and $p = 4$ which uses 4 processes per node.

6 Conclusions

Both the old code used for [4] and the new code used for this report show very good performance on tara. This observation characterizes the efficiency and appropriateness of the architecture of the cluster tara and the usefulness and capability of the InfiniBand interconnect network. The new coding effort clearly shows an improvement of speed for any pattern and arrangement of processes per node, with the extreme case of speedup by a factor of about 2.0 for the serial code. According to our experimentation with the two versions of the new code, i.e., using blocking or non-blocking MPI communication commands for the inevitable data exchange between the processes, it appears that non-blocking commands eventually significantly outperform blocking ones for large numbers of processes, for any pattern and arrangement of processes between 1 to 8 processes per node. It was also justified that the non-blocking version of the code scales well even to 512 processes for 1, 2, 4, and 8 processes per node.

References

[1] Kevin P. Allen. A parallel matrix-free implementation of the conjugate gradient method for the Poisson equation. Senior thesis, University of Maryland, Baltimore County, 2003.

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

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

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

[5] David S. Watkins. *Fundamentals of Matrix Computations*. Wiley, second edition, 2002.

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

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

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