Parallel Performance Studies for an Elliptic Test Problem on the Cluster tara

Andrew M. Raim and Matthias K. Gobbert (gobbert@umbc.edu)

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

Technical Report HPCF–2010–2, www.umbc.edu/hpcf > Publications

Abstract

The performance of parallel computer code depends on an intricate interplay of the processors, the architecture of the compute nodes, their interconnect network, the numerical algorithm, and its implementation. 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. This note considers the classical elliptic test problem of a Poisson equation with Dirichlet boundary conditions in two spatial dimensions, whose approximation by the finite difference method results in a linear system of this type. Our existing implementation of the conjugate gradient method for the iterative solution of this system is known to have the potential to perform well up to many parallel processes, provided the interconnect network has low latency. Since the algorithm is known to be memory bound, it is also vital for good performance that the architecture of the nodes in conjunction with the scheduling policy does not create a bottleneck. The results presented here show excellent performance on the cluster tara with up to 512 parallel processes when using 64 compute nodes. The results support the scheduling policy implemented, since they confirm that it is beneficial to use all eight cores of the two quad-core processors on each node simultaneously, giving us in effect a computer that can run jobs efficiently with up to 656 parallel processes when using all 82 compute nodes. 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 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, matrix-free 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 bound algorithms. Section 2 details the test problem and discusses the parallel implementation in more detail, and Section 3 summarizes the solution and method convergence data.

Past results using an implementation of this method [1, 2, 3, 4, 7] 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 is an update to the technical report [7] and considers the same problem for the new cluster tara. This 86-node distributed-memory cluster was purchased in 2009 by the UMBC High Performance Computing Facility (www.umbc.edu/hpcf) 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, 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 tara, the iDataPlex rack houses the 84 compute and develop nodes and includes all 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. The PGI 9.0 C compiler has been used to create the executables which were utilized in this report.

[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/

Section 4 describes the parallel performance studies in detail and provides the underlying data for the following summary results. Table 1.1 summarizes the key results of the present study by giving the observed wall clock time (total time to execute the code) in HH:MM:SS (hours:minutes:seconds) format. We consider the test problem on six progressively finer meshes, resulting in progressively larger systems of linear equations with system dimensions ranging from about 1 million to over 1 billion equations. The parallel implementation of the conjugate gradient method is run on increasing numbers of nodes from 1 to 64 while varying the number of processes per node from 1 to 8. 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 29 seconds for the 1024×1024 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. More strikingly, one realizes the advantage of parallel computing for the large 16384×16384 mesh with over 268 million equations: The serial run of about 34 hours can be reduced to about 7 minutes using 512 parallel processes. Furthermore, refining the mesh to 32768×32768 yields a problem with more than 1 billion equations, which is not feasible to solve on a single compute node due to memory limitations. However, this large problem can be solved using 512 parallel processes in under an hour, in fact. 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 1.1 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 1.1, speedup in proportion to the number of nodes used is observable. This is discussed in detail in Section 4 in terms of the number of parallel processes. 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 4096×4096, 8192×8192, and 16384×16384 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.

(ii) To analyze the effect of running 1, 2, 4, 6, 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 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.

The results shown in Table 1.1 used the MVAPICH2 implementation of MPI. They may be compared with the results in Table 1.2, where OpenMPI has been used. From the raw timings it is difficult to tell a clear winner between the two implementations, but a careful comparison highlights a few patterns. Fixing our attention to the largest problem with complete data ($N = 16384$), we notice that when more nodes are in use (16, 32, and 64) with more processes per node, the MVAPICH2 timings seems to reduce more quickly than OpenMPI, as processes-per-node increases. When the number of nodes in use is smaller (1, 2, 4, or 8), the pattern does not seem to be as clear. Sometimes OpenMPI achieves a better time, and sometimes it does not. However, we notice that in a few cases the difference is large. For example consider the case of 2 nodes and 4 processes per node; in this case MVAPICH2 achieves a time of 05:01:41 but OpenMPI trails behind at 06:21:11. But for instance for $N = 8192$ with one node and 1 process per node, the MVAPICH2 time of 04:36:37 is far worse than the OpenMPI time of 03:47:52. For smaller cases of N, the differences in absolute times are usually small. It can be seen in Section 5 that the efficiency for OpenMPI tends to be worse than for MVAPICH2, when mesh resolutions are larger, as the number of processes scales up. Some of the differences observed here may be due to experimental variability, and more studies (also with other codes) will be useful. But because of its potential to perform better for large cases (codes with large memory) for large numbers of nodes and, importantly, with all cores on the nodes in use, MVAPICH2 was chosen as the default MPI implementation for tara. This choice has the potential for the most effective use of the cluster for production, because it is optimized for high throughout of jobs using all available computational cores.

Table 1.1: Wall clock time in HH:MM:SS on tara using MVAPICH2 for the solution of the elliptic test problem on $N \times N$ meshes using 1, 2, 4, 8, 16, 32, and 64 compute nodes with 1, 2, 4, 6, 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	00:00:29	00:00:14	00:00:06	00:00:04	00:00:02	00:00:01	00:00:01
2 processes per node	00:00:14	00:00:06	00:00:03	00:00:02	00:00:01	00:00:01	00:00:01
4 processes per node	00:00:08	00:00:03	00:00:02	00:00:01	00:00:01	00:00:01	00:00:01
6 processes per node	00:00:06	00:00:02	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01
8 processes per node	00:00:05	00:00:02	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	00:03:53	00:02:10	00:01:09	00:00:32	00:00:14	00:00:08	00:00:05
2 processes per node	00:01:57	00:01:05	00:00:29	00:00:14	00:00:07	00:00:04	00:00:03
4 processes per node	00:01:12	00:00:34	00:00:17	00:00:07	00:00:04	00:00:02	00:00:02
6 processes per node	00:00:58	00:00:35	00:00:14	00:00:05	00:00:03	00:00:02	00:00:02
8 processes per node	00:00:47	00:00:23	00:00:11	00:00:04	00:00:02	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	00:31:16	00:18:36	00:09:15	00:04:49	00:02:15	00:01:05	00:00:30
2 processes per node	00:15:47	00:08:19	00:04:26	00:02:05	00:01:02	00:00:29	00:00:15
4 processes per node	00:10:07	00:04:53	00:02:20	00:01:17	00:00:36	00:00:16	00:00:08
6 processes per node	00:07:51	00:03:33	00:01:50	00:01:01	00:00:33	00:00:13	00:00:07
8 processes per node	00:06:39	00:03:09	00:01:35	00:00:49	00:00:23	00:00:09	00:00:05

(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	04:36:37	02:20:54	01:14:14	00:38:47	00:19:31	00:09:20	00:04:36
2 processes per node	02:07:34	01:08:57	00:34:11	00:18:09	00:08:25	00:04:14	00:02:08
4 processes per node	01:15:55	00:40:23	00:20:55	00:09:38	00:05:52	00:03:00	00:01:30
6 processes per node	00:56:59	00:29:28	00:14:43	00:07:32	00:04:26	00:02:29	00:01:23
8 processes per node	00:53:55	00:26:26	00:12:54	00:06:30	00:03:20	00:01:43	00:00:50

(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	33:57:41	19:44:19	09:54:27	05:01:30	02:34:15	01:17:16	00:37:23
2 processes per node	16:21:30	08:31:15	04:31:04	02:23:42	01:09:54	00:34:00	00:17:01
4 processes per node	10:03:34	05:01:41	02:41:11	01:24:53	00:47:29	00:22:45	00:11:43
6 processes per node	08:20:03	04:04:07	02:02:50	01:02:55	00:32:32	00:17:35	00:08:59
8 processes per node	07:07:54	03:39:54	01:57:19	00:56:47	00:26:50	00:13:44	00:07:04

(f) Mesh resolution $N \times N = 32768 \times 32768$, system dimension 1,073,741,824

	1 node	2 nodes	4 nodes	8 nodes	16 nodes	32 nodes	64 nodes
1 process per node	N/A	N/A	N/A	N/A	N/A	N/A	05:21:27
2 processes per node	N/A	N/A	N/A	N/A	N/A	N/A	02:15:16
4 processes per node	N/A	N/A	N/A	N/A	N/A	N/A	01:27:53
6 processes per node	N/A	N/A	N/A	N/A	N/A	N/A	01:03:53
8 processes per node	N/A	N/A	N/A	N/A	N/A	N/A	00:55:07

The results in Table 1.1 can also be compared to corresponding ones obtained on the cluster hpc in 2008, reported in [7, Table 1]. For the resolutions that hpc could fit into memory of one node, that is, for $N = 1024$, 2048, 4096, and 8192, the serial runs in the entry for 1 process per node on tara are about 3 times as fast as the hpc results. This shows the core-by-core performance improvement of an Intel Nehalem processor from 2009 compared to an AMD Opteron from 2008. A similar factor of speed improvement can be observed for other directly comparable cases of 2 and 4 processes per node on 2, 4, 8, 16, and 32 nodes used. It is even more striking to compare the performance on a nodal basis, that is using all available cores per node, which is 8 on tara and 4 on hpc. In this comparison, tara running the maximum possible 8 processes per node is often from 3 to 5 times faster than hpc running the maximum possible 4 processes per node, albeit with a significant amount of experimental variability.

Table 1.2: Wall clock time in HH:MM:SS on tara using OpenMPI for the solution of the elliptic test problem on $N \times N$ meshes using 1, 2, 4, 8, 16, 32, and 64 compute nodes with 1, 2, 4, 6, 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	00:00:27	00:00:15	00:00:06	00:00:04	00:00:02	00:00:01	00:00:01
2 processes per node	00:00:16	00:00:06	00:00:03	00:00:02	00:00:01	00:00:01	00:00:01
4 processes per node	00:00:09	00:00:03	00:00:02	00:00:01	00:00:01	00:00:00	00:00:01
6 processes per node	00:00:07	00:00:02	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01
8 processes per node	00:00:06	00:00:02	00:00:01	00:00:01	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	00:04:15	00:02:02	00:01:02	00:00:31	00:00:15	00:00:08	00:00:05
2 processes per node	00:02:17	00:01:05	00:00:31	00:00:14	00:00:07	00:00:04	00:00:03
4 processes per node	00:01:26	00:00:37	00:00:23	00:00:08	00:00:04	00:00:02	00:00:02
6 processes per node	00:01:04	00:00:30	00:00:14	00:00:06	00:00:03	00:00:02	00:00:02
8 processes per node	00:00:53	00:00:24	00:00:13	00:00:05	00:00:02	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	00:30:37	00:16:30	00:08:37	00:04:25	00:02:11	00:01:04	00:00:31
2 processes per node	00:14:52	00:08:00	00:05:19	00:02:22	00:01:07	00:00:31	00:00:16
4 processes per node	00:10:31	00:05:09	00:02:38	00:01:47	00:00:50	00:00:17	00:00:09
6 processes per node	00:07:12	00:03:42	00:03:16	00:01:18	00:00:45	00:00:14	00:00:07
8 processes per node	00:06:30	00:03:17	00:01:47	00:01:04	00:00:28	00:00:10	00:00:05

(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	03:47:52	02:20:25	01:07:00	00:34:49	00:17:45	00:08:55	00:04:32
2 processes per node	02:01:22	01:06:54	00:38:13	00:21:28	00:11:27	00:04:49	00:02:44
4 processes per node	01:22:52	00:37:51	00:23:49	00:12:30	00:07:18	00:03:39	00:01:45
6 processes per node	00:59:35	00:32:43	00:16:30	00:13:43	00:06:50	00:03:27	00:01:35
8 processes per node	00:54:21	00:26:33	00:14:33	00:09:04	00:05:34	00:02:40	00:00:50

(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	35:50:59	19:14:26	09:16:08	04:47:07	02:29:53	01:16:47	00:39:42
2 processes per node	21:06:42	08:26:28	04:52:10	02:58:04	01:34:56	00:48:27	00:20:56
4 processes per node	12:22:37	07:21:12	02:49:45	01:37:28	00:59:10	00:29:55	00:15:03
6 processes per node	08:05:50	04:35:21	02:18:58	01:08:19	00:42:47	00:27:52	00:14:04
8 processes per node	08:11:19	03:36:54	01:55:42	00:58:25	00:35:20	00:17:47	00:11:23

(f) Mesh resolution $N \times N = 32768 \times 32768$, system dimension 1,073,741,824

	1 node	2 nodes	4 nodes	8 nodes	16 nodes	32 nodes	64 nodes
1 process per node	N/A	N/A	N/A	N/A	N/A	N/A	05:10:45
2 processes per node	N/A	N/A	N/A	N/A	N/A	N/A	03:16:56
4 processes per node	N/A	N/A	N/A	N/A	N/A	N/A	02:00:14
6 processes per node	N/A	N/A	N/A	N/A	N/A	N/A	01:52:48
8 processes per node	N/A	N/A	N/A	N/A	N/A	N/A	01:38:21

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 no. CNS–0821258) and the SCREMS program (grant no. DMS–0821311), with additional substantial support from the University of Maryland, Baltimore County (UMBC). See `www.umbc.edu/hpcf` for more information on HPCF and the projects using its resources. Andrew Raim additionally acknowledges financial support as HPCF RA.

2 The Elliptic Test Problem

We consider the classical elliptic test problem of the Poisson equation with homogeneous Dirichlet boundary conditions (see, e.g., [9, 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 positive definite and thus the method is guaranteed to converge for this problem. In a careful implementation, the conjugate gradient method requires in each iteration exactly two inner products between vectors, three vector updates, and one matrix-vector product involving the system matrix A. In fact, this matrix-vector product is the only way, in which A enters into the algorithm. Therefore, a so-called matrix-free implementation of the conjugate gradient method is possible that avoids setting up any matrix, if one provides a function that computes as its output the product vector $q = A\,p$ component-wise directly from the components of the input vector p by using the explicit knowledge of the values and positions of the non-zero components of A, but without assembling A as a matrix.

Thus, without storing A, a careful, efficient, matrix-free implementation of the (unpreconditioned) conjugate gradient method only requires the storage of four vectors (commonly denoted as the solution vector x, the residual r, the search direction p, and an auxiliary vector q). In a parallel implementation of the conjugate gradient method, each vector is split into as many blocks as parallel processes are available and one block distributed to each process. That is, each parallel process possesses its own block of each vector, and normally no vector is ever assembled in full on any process. To understand what this means for parallel programming and the performance of the method, note that an inner product between two vectors distributed in this way is computed by first forming the local inner products between the local blocks of the vectors and second summing all local inner products across all parallel processors to obtain the global inner product. This summation of values from all processes is known as a reduce operation in parallel programming, which requires a communication among all parallel processes. This communication is necessary as part of the numerical method used, and this necessity is responsible for the fact that for fixed problem sizes eventually for very large numbers of processors the time needed for communication —

increasing with the number of processes — will unavoidably dominate over the time used for the calculations that are done simultaneously in parallel — decreasing due to shorter local vectors for increasing number of processes. By contrast, the vector updates in each iteration can be executed simultaneously on all processes on their local blocks, because they do not require any parallel communications. However, this requires that the scalar factors that appear in the vector updates are available on all parallel processes. This is accomplished already as part of the computation of these factors by using a so-called Allreduce operation, that is, a reduce operation that also communicates the result to all processes. This is implemented in the MPI function `MPI_Allreduce`. Finally, the matrix-vector product $q = Ap$ also computes only the block of the vector q that is local to each process. But since the matrix A has non-zero off-diagonal elements, each local block needs values of p that are local to the two processes that hold the neighboring blocks of p. The communications between parallel processes thus needed are so-called point-to-point communications, because not all processes participate in each of them, but rather only specific pairs of processes that exchange data needed for their local calculations. Observe now that it is only a few components of q that require data from p that is not local to the process. Therefore, it is possible and potentially very efficient to proceed to calculate those components that can be computed from local data only, while the communications with the neighboring processes are taking place. This technique is known as interleaving calculations and communications and can be implemented using the non-blocking MPI communications commands `MPI_Isend` and `MPI_Irecv`.

3 Convergence Study for the Model Problem

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

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

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

To check the convergence of the finite difference method as well as to analyze the performance of the conjugate gradient method, we solve the problem on a sequence of progressively finer meshes. The conjugate gradient method is started with a zero vector as initial guess and the solution is accepted as converged when the Euclidean vector norm of the residual is reduced to the fraction 10^{-6} of the initial residual. Table 3.1 lists the mesh resolution N of the $N \times N$ mesh, the number of degrees of freedom N^2 (DOF; i.e., the dimension of the linear system), the norm of the finite difference error $\|u - u_h\|_{L^\infty(\Omega)}$, 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

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

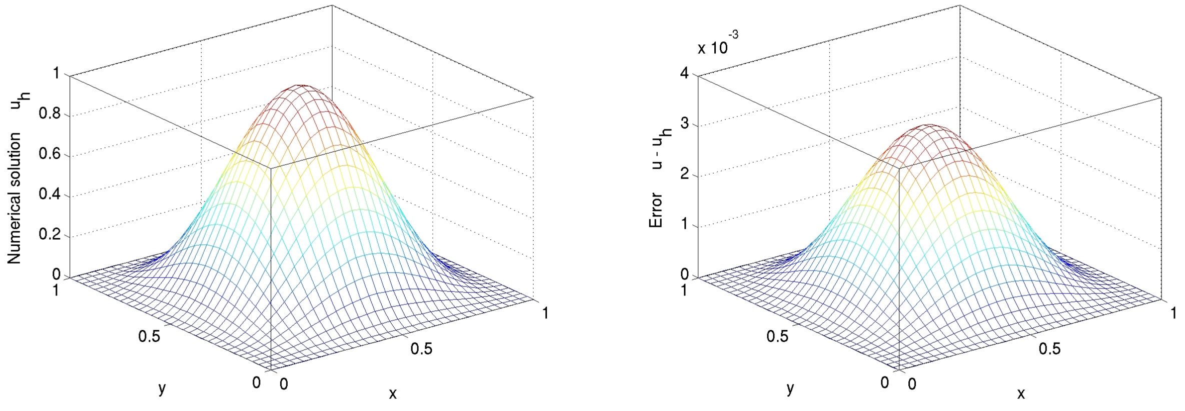


Figure 3.1: Mesh plots of (a) the numerical solution u_h vs. (x_1, x_2) and (b) the error $u - u_h$ vs. (x_1, x_2).

Table 3.1: Convergence study (using MVAPICH2) 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	3.2189e–3	47	<00:00:01	< 0.01	< 1	12
64	4,096	8.0356e–4	95	<00:00:01	0.01	< 1	12
128	16,384	2.0081e–4	191	<00:00:01	0.06	< 1	12
256	65,536	5.0191e–5	385	<00:00:01	0.39	2	13
512	262,144	1.2543e–5	781	00:00:04	3.59	8	19
1024	1,048,576	3.1327e–6	1,579	00:00:29	28.75	32	44
2048	4,194,304	7.8097e–7	3,191	00:03:53	233.42	128	140
4096	16,777,216	1.9356e–7	6,447	00:31:16	1,876.06	512	524
8192	67,108,864	4.6817e–8	13,028	04:36:37	16,596.76	2,048	2,061
16384	268,435,456	8.0469e–9	26,321	33:57:41	122,261.31	8,192	8,207
†32768	1,073,741,824	2.9562e–9	53,136	N/A	N/A	32,768	33,923

† A serial run is not possible due to memory limitations. This result uses 64 nodes with one process per node. The timing is not shown here, since it should not be compared to the serial runs.

in serial. More precisely, the runs used the parallel code run on one process only, on a dedicated node (no other processes running on the node), and with all parallel 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 [6, 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. For the two finest mesh resolutions, the reduction in error appears slighly more erratic, which points to the tolerance not being tight enough beyond these resolutions. The increasing numbers of iterations needed to achieve the convergence of the linear solver highlights the fundamental computational challenge with methods in the family of Krylov subspace methods, of which the conjugate gradient method is the most important example: Refinements of the mesh imply more mesh points, where the solution approximation needs to be found, and makes the computation of each iteration of the linear solver more expensive. Additionally, more of these more expensive iterations are required to achieve convergence to the desired tolerance for finer meshes. And it is not possible to relax the solver tolerance too much, because otherwise its solution would not be accurate enough and the norm of the finite difference error would not show a second-order convergence behavior, as required by its theory. 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 challenging computationally. 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.

We finally note that the results for the finite difference error and the conjugate gradient iterations in Table 3.1 agree with past results for this problem; see [7] and the references therein. This ensures that the parallel performance studies in the next section are practically relevant in that a correct solution of the test problem is computed.

4 Performance Studies on tara with MVAPICH2

The run times for the finer meshes observed for serial runs in Table 3.1 bring 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 that it be p times as fast. If $T_p(N)$ denotes the wall clock time for a problem of a fixed size parametrized by N using p processes, then the quantity $S_p = T_1(N)/T_p(N)$ measures the speedup of the code from 1 to p processes, whose optimal value is $S_p = p$. The efficiency $E_p = S_p/p$ characterizes in relative terms how close a run with p parallel processes is to this optimal value, for which $E_p = 1$. The behavior described here for speedup for a fixed problem size is known as strong scalability of parallel code.

The results given in this section are based on the MVAPICH2 implementation of MPI. We have also presented the same study in Appendix 5 using the OpenMPI implementation. Table 4.1 lists the results of a performance study for strong scalability. Each row lists the results for one problem size, parametrized by the mesh resolution N. Each column corresponds to the number of parallel processes p used in the run. The runs for Table 4.1 distribute these processes as widely as possible over the available nodes, that is, each process is run on a different node up to a maximum number of 64 nodes. In other words, up to $p = 64$, seven of the eight cores available on each node are idling, and only one core performs calculations. For $p = 128$, $p = 256$, and $p = 512$, this cannot be accommodated on 64 nodes, thus 2 processes run on each node for $p = 128$, 4 processes per node for $p = 256$, and 8 processes per node for $p = 512$. Comparing adjacent columns in the raw timing data in Table 4.1 (a) indicates that using twice as many processes speeds up the code by a factor two approximately, at least up to $p = 32$. To quantify this more clearly, the speedup in Table 4.1 (b) is computed, which shows near-optimal speedup with $S_p \approx p$ for all cases except $N = 1024$, up to $p = 64$, which is expressed in terms of efficiency $E_p \approx 1$ in Table 4.1 (c). The customary visualizations of speedup and efficiency are presented in Figure 4.1 (a) and (b), respectively. Figure 4.1 (a) shows very clearly the excellent speedup up to $p = 64$ parallel processes. The efficiency plotted in Figure 4.1 (b) is directly derived from the speedup, but the plot is still useful because it can better bring out any interesting features for small values of p that are hard to tell in a speedup plot. Here, we notice that the variability of the results for small p is visible. In fact, we notice here, as in the table, that a number of results show apparently better than optimal behavior, with efficiency greater than 1.0. This can happen due to experimental variability of the runs, for instance, if the single-process timing $T_1(N)$ used in the computation of $S_p = T_1(N)/T_p(N)$ happens to be slowed down in some way. Another reason for excellent performance can also be that runs on many processes result in local problems that fit or nearly fit into the cache of the processor, which leads to fewer cache misses and thus potentially dramatic improvement of the run time, beyond merely distributing the calculations to more processes. It is customary in results for fixed problem sizes that the speedup is better for larger problems, since the increased communication time for more parallel processes does not dominate over the calculation time as quickly as it does for small problems. Thus, it is in fact remarkable how well the speedup is for the smaller values of mesh resolution N here. To see this clearly, it is vital to have the precise data in Table 4.1 (b) and (c) available and not just their graphical representation in Figure 4.1. The conclusions discussed so far apply to up to $p = 64$ parallel processes. In each case, only 1 parallel process is run on each node, with the other seven cores available to handle all other operating system or other duties. For $p = 128$, $p = 256$, and $p = 512$, 2, 4, or 8 processes share each node necessarily, as 64 nodes are available, thus one expects slightly degraded performance as we go from $p = 64$ to $p = 128$, $p = 256$, and $p = 512$. This is born out by all data in Table 4.1 as well as clearly visible in Figure 4.1 for $p > 64$. However, the times in Table 4.1 (a) for all finer meshes with $N > 1024$ clearly demonstrate an improvement by using more cores, just not at the optimal rate of halving the timing any more.

To analyze the impact of using more than one core per node, we use 2 processes per node in Table 4.2 and Figure 4.2, 4 processes per node in Table 4.3 and Figure 4.3, and we use 8 processes per node in Table 4.5 and Figure 4.5 wherever possible. That is, $p = 512$ requires 8 processes per node also in Table 4.2 and Figure 4.2, as 64 nodes are available. And $p = 1$ is always computed on a dedicated node, i.e., using running only this job on the node, and $p = 2$ in Table 4.3 and Figure 4.3 runs only this two-process job on its node, and so forth. The results in the efficiency plots of Tables 4.2 (b), 4.3 (b), and 4.5 (b) show generally good efficiency for the larger problem sizes. However, it is curious that the efficiency for larger problem sizes degrades as p increases, and seems to peak at $N = 4096$. Notably, when $p = 512$ and 1 process per node is in use the efficiency for $N = 16384$ is below that of $N = 2048$. Scanning across the row for $N = 16384$ in Table 4.1, we can see that the efficiency drops dramatically after $p = 128$. A similar pattern occurs for 2 processes per node. However when using 4, 6, or 8 processes per node, the drop in efficiency starts at a much smaller p, and is much more gradual. A drop in efficiency for larger problem sizes defies our intuition, but focusing on the raw timings in Tables 4.1 (a), 4.2 (a), 4.3 (a), and 4.5 (a), we can see that increasing the number of parallel processes is very effective for reducing the required wall time to solve the larger problem sizes.

We have shown the performance in solving a very fine 32768×32768 mesh as well. This has been done for 64 nodes when $p = 64, 128, 256, 512$, and the full results can be seen in Table 4.1. It is clear that an optimal halving of run times is not occuring — for example when in increasing from $p = 256$ to $p = 512$, the wall time only decreases from about $1\frac{1}{2}$ hours to 55 minutes. However, this result would likely have taken on the order of two weeks if it could have been run in serial (note however from Table 3.1 that it can not fit on a single node). To make this rough estimate of two weeks, notice that the serial timings in Table 4.1 seem to be increasing by a factor of $8\frac{1}{2}$ to 10 each time the problem size increases.

We have also shown the scaling behavior when 6 processes per node are used, see Table 4.4, and Figure 4.4. Intel Nehalem processors have three memory channels per processor [5]. One might reason that running 6 processes per node, which limits the number of parallel processes to the number of memory channels per node, might show a better performance than using more. Comparing Table 4.4 to Table 4.5, we notice similar scaling behavior as p increases. However, focusing on the results for the $p = 384$ and $p = 512$ runs, we notice that the efficiency is slightly better when $p = 8$. Therefore, it does not appear that limiting ourselves to 6 processes per node gives any special boost in performance.

The results presented so far indicate clearly the well-known conclusion that best performance improvements, in the sense of halving the time when doubling the number of processes, is achieved by only running one parallel process on each node. But for production runs, we are not interested in this improvement being optimal, but we are interested in the run time being the smallest on a given number of nodes. Thus, given a fixed number of nodes, the question is if one should run 1, 2, 4, 6, or 8 processes per node. This is answered by the data organized in the form of Table 1.1 in the Introduction. It is these results, which are the same raw timing data as in Tables 4.1, 4.2, 4.3, 4.4 and 4.5, that make it clear that using 8 processes per node is in fact the best way to use this cluster, and that in fact the improvement of timings is nearly optimal by doubling processes per node, as well. This is an excellent result, and it is remarkable that it is applicable for cases as small as $N = 2048$, which is not a particularly large problem by today's standards; see Table 3.1.

Table 4.1: MVAPICH2 performance 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	00:00:29	00:00:14	00:00:06	00:00:04	00:00:02	00:00:01	00:00:01	00:00:01	00:00:01	00:00:00
2048	00:03:53	00:02:10	00:01:09	00:00:32	00:00:14	00:00:08	00:00:05	00:00:03	00:00:02	00:00:01
4096	00:31:16	00:18:36	00:09:15	00:04:49	00:02:16	00:01:05	00:00:30	00:00:15	00:00:08	00:00:05
8192	04:36:37	02:20:54	01:14:14	00:38:47	00:19:31	00:09:20	00:04:36	00:02:08	00:01:30	00:00:50
16384	33:57:41	19:44:19	09:54:27	05:01:30	02:34:15	01:17:16	00:37:23	00:17:01	00:11:43	00:07:04
32768	N/A	N/A	N/A	N/A	N/A	N/A	05:21:27	02:15:16	01:27:53	00:55:07

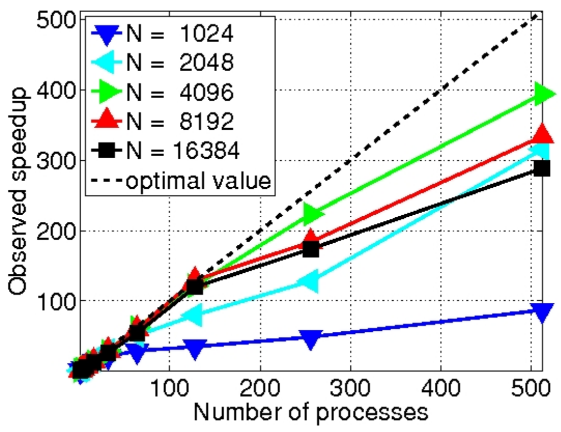
(b) Observed speedup S_p

N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$	$p = 256$	$p = 512$
1024	1.0000	2.0035	4.5927	8.1676	14.8196	21.4552	29.0404	35.0610	48.7288	87.1212
2048	1.0000	1.7964	3.4016	7.3588	16.2097	30.1187	50.7435	79.9384	128.2527	315.4324
4096	1.0000	1.6811	3.3794	6.4934	13.8455	28.7651	62.4729	121.4278	223.3405	394.1303
8192	1.0000	1.9632	3.7264	7.1310	14.1746	29.6418	60.0592	129.5711	183.7958	333.8717
16384	1.0000	1.7206	3.4279	6.7584	13.2099	26.3748	54.5196	119.7103	173.8222	288.4474
32768	N/A	N/A	N/A	N/A	N/A	N/A	64.0000	152.0894	234.0873	373.2662

(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	1.0017	1.1482	1.0210	0.9262	0.6705	0.4538	0.2739	0.1903	0.1702
2048	1.0000	0.8982	0.8504	0.9198	1.0131	0.9412	0.7929	0.6245	0.5010	0.6161
4096	1.0000	0.8406	0.8449	0.8117	0.8653	0.8989	0.9761	0.9487	0.8724	0.7698
8192	1.0000	0.9816	0.9316	0.8914	0.8859	0.9263	0.9384	1.0123	0.7180	0.6521
16384	1.0000	0.8603	0.8570	0.8448	0.8256	0.8242	0.8519	0.9352	0.6790	0.5634
32768	N/A	N/A	N/A	N/A	N/A	N/A	1.0000	1.1882	0.9144	0.7290

(a) Observed speedup S_p



(b) Observed efficiency E_p

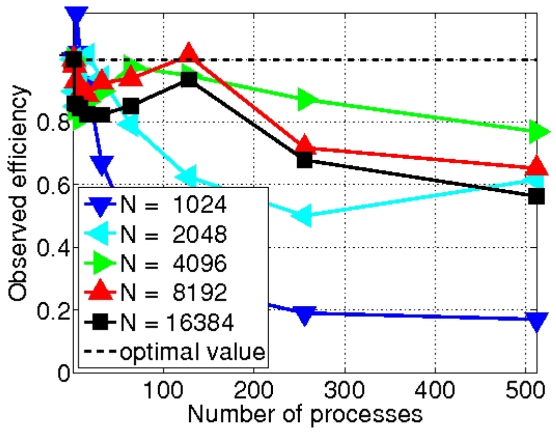


Figure 4.1: MVAPICH2 performance on tara by number of processes used with 1 process per node, except for $p = 128$ which uses 2 processes per node $p = 256$ which uses 4 processes per node, and $p = 512$ which uses 8 processes per node.

Table 4.2: MVAPICH2 performance 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	00:00:29	00:00:14	00:00:06	00:00:03	00:00:02	00:00:01	00:00:01	00:00:01	00:00:01	00:00:00
2048	00:03:53	00:01:57	00:01:05	00:00:29	00:00:14	00:00:07	00:00:04	00:00:03	00:00:02	00:00:01
4096	00:31:16	00:15:47	00:08:19	00:04:26	00:02:05	00:01:02	00:00:29	00:00:15	00:00:08	00:00:05
8192	04:36:37	02:07:34	01:08:57	00:34:11	00:18:09	00:08:25	00:04:14	00:02:08	00:01:30	00:00:50
16384	33:57:41	16:21:30	08:31:15	04:31:04	02:23:42	01:09:54	00:34:00	00:17:01	00:11:43	00:07:04
32768	N/A	N/A	N/A	N/A	N/A	N/A	05:21:27	02:15:16	01:27:53	00:55:07

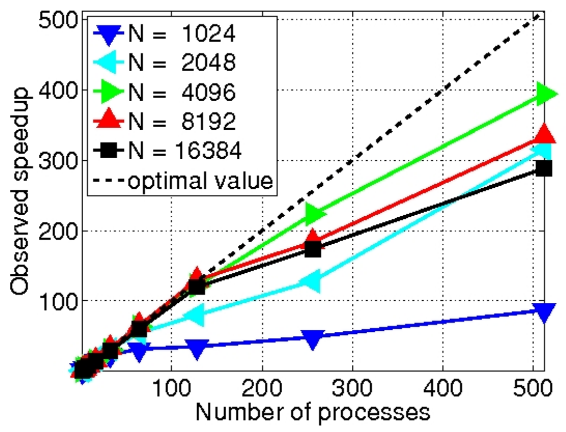
(b) Observed speedup S_p

N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$	$p = 256$	$p = 512$
1024	1.0000	2.0246	4.6672	9.0125	15.0524	23.7603	31.5934	35.0610	48.7288	87.1212
2048	1.0000	1.9978	3.5702	7.9179	16.5546	31.3737	54.4103	79.9384	128.2527	315.4324
4096	1.0000	1.9818	3.7566	7.0560	14.9917	30.0747	63.7682	121.4278	223.3405	394.1303
8192	1.0000	2.1683	4.0117	8.0907	15.2404	32.8590	65.2158	129.5711	183.7958	333.8717
16384	1.0000	2.0761	3.9857	7.5172	14.1805	29.1543	59.9344	119.7103	173.8222	288.4474
32768	N/A	N/A	N/A	N/A	N/A	N/A	64.0000	152.0894	234.0873	373.2662

(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	1.0123	1.1668	1.1266	0.9408	0.7425	0.4936	0.2739	0.1903	0.1702
2048	1.0000	0.9989	0.8926	0.9897	1.0347	0.9804	0.8502	0.6245	0.5010	0.6161
4096	1.0000	0.9909	0.9392	0.8820	0.9370	0.9398	0.9964	0.9487	0.8724	0.7698
8192	1.0000	1.0841	1.0029	1.0113	0.9525	1.0268	1.0190	1.0123	0.7180	0.6521
16384	1.0000	1.0380	0.9964	0.9397	0.8863	0.9111	0.9365	0.9352	0.6790	0.5634
32768	N/A	N/A	N/A	N/A	N/A	N/A	1.0000	1.1882	0.9144	0.7290

(a) Observed speedup S_p



(b) Observed efficiency E_p

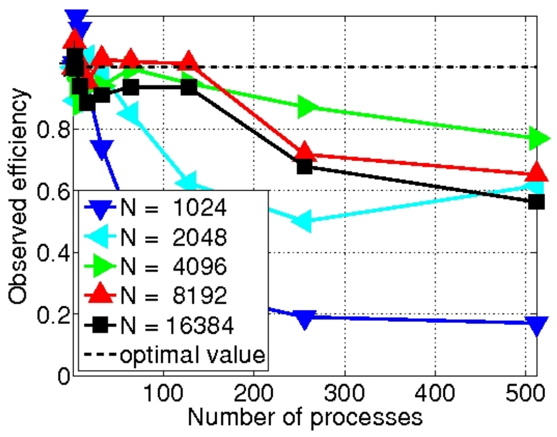


Figure 4.2: MVAPICH2 performance on tara by number of processes used with 2 processes per node, except for $p = 1$ which uses 1 process per node, $p = 256$ which uses 4 processes per node, and $p = 512$ which uses 8 processes per node.

Table 4.3: MVAPICH2 performance 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, $p = 512$ 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	00:00:29	00:00:14	00:00:08	00:00:03	00:00:02	00:00:01	00:00:01	00:00:01	00:00:01	00:00:00
2048	00:03:53	00:01:57	00:01:12	00:00:34	00:00:17	00:00:08	00:00:04	00:00:02	00:00:02	00:00:01
4096	00:31:16	00:15:47	00:10:07	00:04:53	00:02:20	00:01:17	00:00:36	00:00:16	00:00:08	00:00:05
8192	04:36:37	02:07:34	01:15:55	00:40:23	00:20:55	00:09:38	00:05:52	00:03:00	00:01:30	00:00:50
16384	33:57:41	16:21:30	10:03:34	05:01:41	02:41:11	01:24:53	00:47:29	00:22:45	00:11:43	00:07:04
32768	N/A	N/A	N/A	N/A	N/A	N/A	05:21:27	02:15:16	01:27:53	00:55:07

(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	2.0246	3.7097	8.9564	18.0818	33.0460	44.2308	50.4386	48.7288	87.1212
2048	1.0000	1.9978	3.2339	6.8754	13.5473	31.1227	58.3550	96.0576	128.2527	315.4324
4096	1.0000	1.9818	3.0918	6.4058	13.4129	24.2542	52.0549	117.9170	223.3405	394.1303
8192	1.0000	2.1683	3.6433	6.8509	13.2258	28.7032	47.1231	92.1530	183.7958	333.8717
16384	1.0000	2.0761	3.3761	6.7544	12.6415	24.0038	42.9121	89.5445	173.8222	288.4474
32768	N/A	N/A	N/A	N/A	N/A	N/A	64.0000	152.0894	234.0873	373.2662

(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	1.0123	0.9274	1.1195	1.1301	1.0327	0.6911	0.3941	0.1903	0.1702
2048	1.0000	0.9989	0.8085	0.8594	0.8467	0.9726	0.9118	0.7505	0.5010	0.6161
4096	1.0000	0.9909	0.7729	0.8007	0.8383	0.7579	0.8134	0.9212	0.8724	0.7698
8192	1.0000	1.0841	0.9108	0.8564	0.8266	0.8970	0.7363	0.7199	0.7180	0.6521
16384	1.0000	1.0380	0.8440	0.8443	0.7901	0.7501	0.6705	0.6996	0.6790	0.5634
32768	N/A	N/A	N/A	N/A	N/A	N/A	1.0000	1.1882	0.9144	0.7290

(a) Observed speedup S_p (b) Observed efficiency E_p

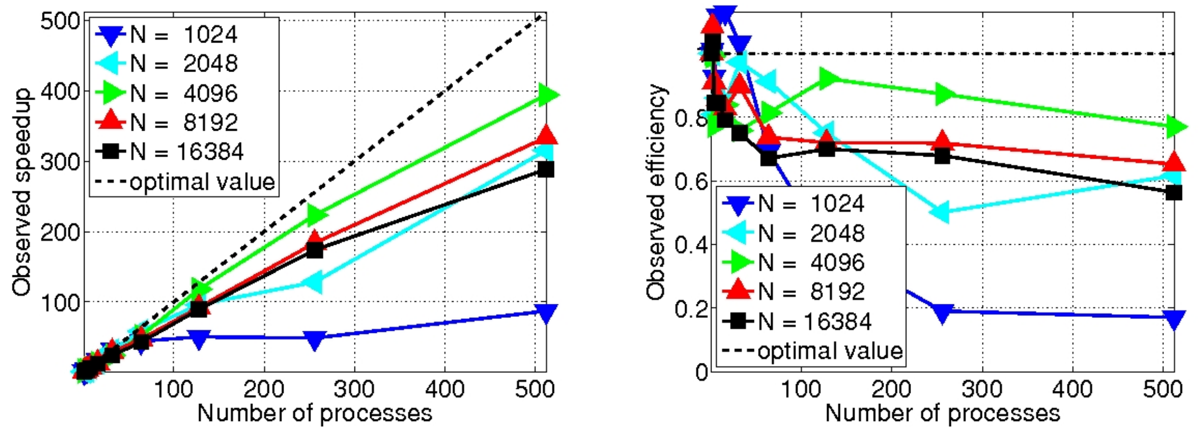


Figure 4.3: MVAPICH2 performance on tara by number of processes used with 4 processes per node, except for $p = 1$ which uses 1 process per node, $p = 2$ which uses 2 processes per node, and $p = 512$ which uses 8 processes per node.

Table 4.4: MVAPICH2 performance on tara by number of processes used with 6 processes per node, except for $p = 1$ which uses 1 process per node.

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

N	$p=1$	$p=6$	$p=12$	$p=24$	$p=48$	$p=96$	$p=192$	$p=384$
1024	00:00:29	00:00:06	00:00:02	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01
2048	00:03:53	00:00:58	00:00:35	00:00:14	00:00:05	00:00:03	00:00:02	00:00:02
4096	00:31:16	00:07:51	00:03:33	00:01:50	00:01:01	00:00:33	00:00:13	00:00:07
8192	04:36:37	00:56:59	00:29:28	00:14:43	00:07:32	00:04:26	00:02:29	00:01:23
16384	33:57:41	08:20:03	04:04:07	02:02:50	01:02:55	00:32:32	00:17:35	00:08:59
32768	N/A	N/A	N/A	N/A	N/A	N/A	N/A	01:03:53

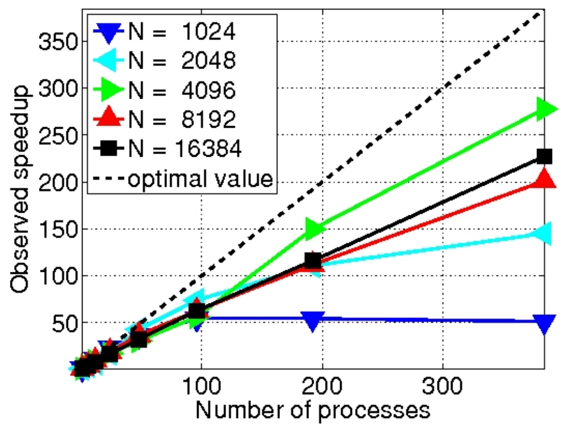
(b) Observed speedup S_p

N	$p=1$	$p=6$	$p=12$	$p=24$	$p=48$	$p=96$	$p=192$	$p=384$
1024	1.0000	4.4782	11.5927	23.5656	36.3924	54.2453	54.2453	51.3393
2048	1.0000	4.0287	6.7462	17.0878	42.8294	73.8671	110.6256	144.9814
4096	1.0000	3.9806	8.8231	17.0566	30.6096	56.2031	149.4869	277.5237
8192	1.0000	4.8542	9.3888	18.7972	36.6885	62.5066	111.7100	200.8807
16384	1.0000	4.0750	8.3470	16.5891	32.3832	62.6368	115.8589	226.9309
32768	N/A	N/A	N/A	N/A	N/A	N/A	N/A	384.0000

(c) Observed efficiency E_p

N	$p=1$	$p=6$	$p=12$	$p=24$	$p=48$	$p=96$	$p=192$	$p=384$
1024	1.0000	0.7464	0.9661	0.9819	0.7582	0.5651	0.2825	0.1337
2048	1.0000	0.6714	0.5622	0.7120	0.8923	0.7694	0.5762	0.3776
4096	1.0000	0.6634	0.7353	0.7107	0.6377	0.5854	0.7786	0.7227
8192	1.0000	0.8090	0.7824	0.7832	0.7643	0.6511	0.5818	0.5231
16384	1.0000	0.6792	0.6956	0.6912	0.6747	0.6525	0.6034	0.5910
32768	N/A	N/A	N/A	N/A	N/A	N/A	N/A	1.0000

(a) Observed speedup S_p



(b) Observed efficiency E_p

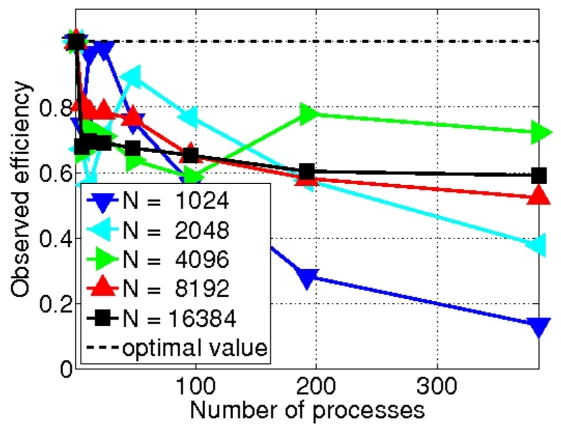


Figure 4.4: MVAPICH2 performance on tara by number of processes used with 6 processes per node, except for $p = 1$ which uses 1 process per node.

Table 4.5: MVAPICH2 performance 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	00:00:29	00:00:14	00:00:08	00:00:05	00:00:02	00:00:01	00:00:00	00:00:00	00:00:00	00:00:00
2048	00:03:53	00:01:57	00:01:12	00:00:47	00:00:23	00:00:11	00:00:04	00:00:02	00:00:01	00:00:01
4096	00:31:16	00:15:47	00:10:07	00:06:39	00:03:09	00:01:35	00:00:49	00:00:23	00:00:09	00:00:05
8192	04:36:37	02:07:34	01:15:55	00:53:55	00:26:26	00:12:54	00:06:30	00:03:20	00:01:43	00:00:50
16384	33:57:41	16:21:30	10:03:34	07:07:54	03:39:54	01:57:19	00:56:47	00:26:50	00:13:44	00:07:04
32768	N/A	N/A	N/A	N/A	N/A	N/A	05:21:27	02:15:16	01:27:53	00:55:07

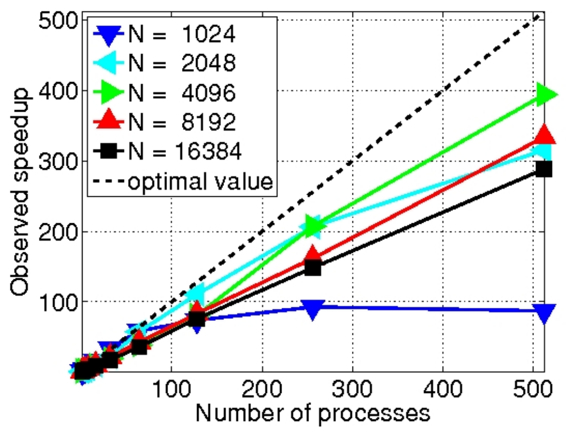
(b) Observed speedup S_p

N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$	$p = 256$	$p = 512$
1024	1.0000	2.0246	3.7097	5.4554	15.6250	33.4302	58.6735	73.7179	92.7419	87.1212
2048	1.0000	1.9978	3.2339	4.9245	9.9923	21.5730	57.4926	110.6256	206.5664	315.4324
4096	1.0000	1.9818	3.0918	4.7037	9.9053	19.7792	38.4124	80.9344	207.2994	394.1303
8192	1.0000	2.1683	3.6433	5.1309	10.4665	21.4550	42.5198	83.1584	161.5571	333.8717
16384	1.0000	2.0761	3.3761	4.7621	9.2663	17.3688	35.8901	75.9486	148.3322	288.4474
32768	N/A	N/A	N/A	N/A	N/A	N/A	64.0000	152.0894	234.0873	373.2662

(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	1.0123	0.9274	0.6819	0.9766	1.0447	0.9168	0.5759	0.3623	0.1702
2048	1.0000	0.9989	0.8085	0.6156	0.6245	0.6742	0.8983	0.8643	0.8069	0.6161
4096	1.0000	0.9909	0.7729	0.5880	0.6191	0.6181	0.6002	0.6323	0.8098	0.7698
8192	1.0000	1.0841	0.9108	0.6414	0.6542	0.6705	0.6644	0.6497	0.6311	0.6521
16384	1.0000	1.0380	0.8440	0.5953	0.5791	0.5428	0.5608	0.5933	0.5794	0.5634
32768	N/A	N/A	N/A	N/A	N/A	N/A	1.0000	1.1882	0.9144	0.7290

(a) Observed speedup S_p



(b) Observed efficiency E_p

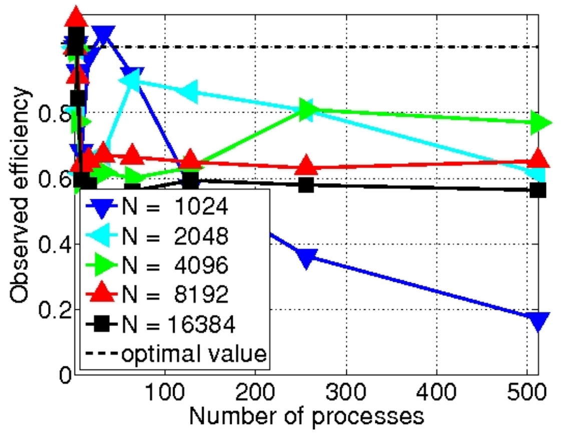


Figure 4.5: MVAPICH2 performance on tara by number of processes used with 8 processes per node, except for $p = 1$ which uses 1 process per node, $p = 2$ which uses 2 processes per node, and $p = 4$ which uses 4 processes per node.

5 Performance Studies on tara with OpenMPI

This section summarizes results of analogous studies to the previous sections, using the OpenMPI implementation of the MPI standard. Otherwise, the same cluster and compiler have been used. The goal of this study is to answer the question, between MVAPICH2 and OpenMPI, which MPI implementation should be the default?

An analogous study for the convergence of the method as reported in Table 3.1 using MVAPICH2 was performed, and identical numerical results were observed when using OpenMPI, which confirms the correctness of the studies. Table 1.2 in the Introduction summarizes the raw timing results for our OpenMPI studies, analogously to Table 1.1 in the Introduction. Reading the data row-wise (varying number of nodes) or column-wise (varying processes per node), we again observe excellent scalability, Tables 5.1, 5.2, 5.3, 5.4, 5.5 and Figures 5.1, 5.2, 5.3, 5.4, 5.5 show detailed performance results, including speedup and efficiency. These results at first glance appear very similar to the 1, 2, 4, 6, and 8 process per node results from Section 4. However, comparing the two carefully reveals some major differences. Consider Table 5.5, and notice the poor efficiency for $N = 8192$ as p increases. When $p = 256$, the efficiency is down to about 33%, while the efficiency for the same case with MVAPICH2 is about 63%. There are two results influencing this calculation; one is the serial case, which finished much more quickly for OpenMPI than for MVAPICH2 (03:47:52 vs. 04:36:37). The other is the $p = 256$ result itself, which was relatively much faster for MVAPICH2 than OpenMPI (00:01:43 vs. 00:02:40), although less than a minute faster in actual wall time. It is also very interesting to note that in Table 5.5, the efficiency greatly improves between the $p = 256$ and $p = 512$ runs, where 64 nodes are used with all 8 processor cores engaged, rather than just 32 nodes. This effect is present in several of the other MVAPICH2 and OpenMPI results, but not nearly as dramatically. As in the MVAPICH2 results, we notice that the efficiency seems to be degrading as the problem size increases, peaking at about $N = 4096$. For OpenMPI the pattern seems to be more drastic however, with efficiencies dropping down below 40%.

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] Kevin P. Allen and Matthias K. Gobbert. A matrix-free conjugate gradient method for cluster computing. Technical Report, University of Maryland, Baltimore County, 2003.

[4] Kevin P. Allen and Matthias K. Gobbert. Coarse-grained parallel matrix-free solution of a three-dimensional elliptic prototype problem. In Vipin Kumar, Marina L. Gavrilova, Chih Jeng Kenneth Tan, and Pierre L'Ecuyer, editors, *Computational Science and Its Applications—ICCSA 2003*, vol. 2668 of *Lecture Notes in Computer Science*, pp. 290–299. Springer-Verlag, 2003.

[5] Ganesh Balakrishnan and Ralph M. Begun. Optimizing the performance of IBM System x and BladeCenter servers using Intel Xeon 5500 series processors. IBM Corporation, March 2009.

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

[7] 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.

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

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

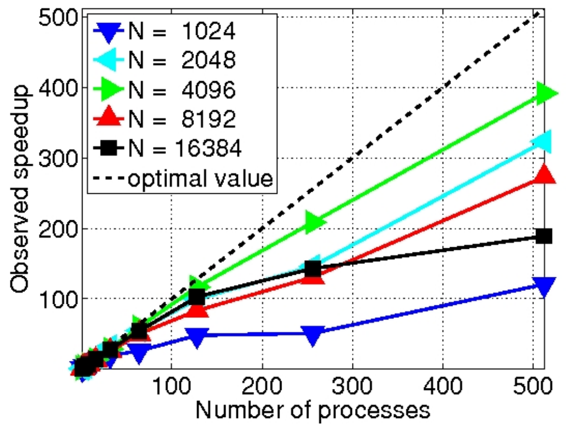
Table 5.1: OpenMPI performance on tara by number of processes used with 1 process per node, except for $p = 128$ which uses 2 processes per node, $p = 256$ which uses 4 processes per node, and $p = 512$ which uses 8 processes per node.

(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	00:00:27	00:00:15	00:00:06	00:00:04	00:00:02	00:00:01	00:00:01	00:00:01	00:00:01	00:00:00
2048	00:04:15	00:02:02	00:01:02	00:00:31	00:00:15	00:00:08	00:00:05	00:00:03	00:00:02	00:00:01
4096	00:30:37	00:16:30	00:08:37	00:04:25	00:02:11	00:01:04	00:00:31	00:00:16	00:00:09	00:00:05
8192	03:47:52	02:20:25	01:07:00	00:34:49	00:17:45	00:08:55	00:04:32	00:02:44	00:01:45	00:00:50
16384	35:50:59	19:14:26	09:16:08	04:47:07	02:29:53	01:16:47	00:39:42	00:20:56	00:15:03	00:11:23
32768	N/A	N/A	N/A	N/A	N/A	N/A	05:10:45	03:16:56	02:00:14	01:38:21

(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	1.7670	4.1664	7.5613	12.9463	19.6593	26.0196	48.2545	51.0385	120.6364
2048	1.0000	2.0932	4.1417	8.3429	17.4733	32.5842	54.9376	99.4008	145.9771	323.3671
4096	1.0000	1.8560	3.5508	6.9320	14.0371	28.8480	60.1519	116.6375	208.5176	391.6930
8192	1.0000	1.6228	3.4010	6.5449	12.8423	25.5326	50.1989	83.2806	130.6467	273.0613
16384	1.0000	1.8632	3.8678	7.4917	14.3517	28.0113	54.1884	102.7522	142.9730	188.9394
32768	N/A	N/A	N/A	N/A	N/A	N/A	64.0000	100.9910	165.4030	202.2180

(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	0.8835	1.0416	0.9452	0.8091	0.6144	0.4066	0.3770	0.1994	0.2356
2048	1.0000	1.0466	1.0354	1.0429	1.0921	1.0183	0.8584	0.7766	0.5702	0.6316
4096	1.0000	0.9280	0.8877	0.8665	0.8773	0.9015	0.9399	0.9112	0.8145	0.7650
8192	1.0000	0.8114	0.8502	0.8181	0.8026	0.7979	0.7844	0.6506	0.5103	0.5333
16384	1.0000	0.9316	0.9669	0.9365	0.8970	0.8754	0.8467	0.8028	0.5585	0.3690
32768	N/A	N/A	N/A	N/A	N/A	N/A	1.0000	0.7890	0.6461	0.3950

(a) Observed speedup S_p



(b) Observed efficiency E_p

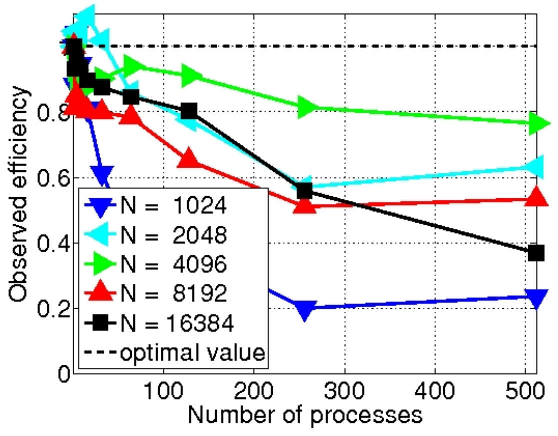


Figure 5.1: OpenMPI performance on tara by number of processes used with 1 process per node, except for $p = 128$ which uses 2 processes per node $p = 256$ which uses 4 processes per node, and $p = 512$ which uses 8 processes per node.

Table 5.2: OpenMPI performance 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	00:00:27	00:00:16	00:00:06	00:00:03	00:00:02	00:00:01	00:00:01	00:00:01	00:00:01	00:00:00
2048	00:04:15	00:02:17	00:01:05	00:00:31	00:00:14	00:00:07	00:00:04	00:00:03	00:00:02	00:00:01
4096	00:30:37	00:14:52	00:08:00	00:05:19	00:02:22	00:01:07	00:00:31	00:00:16	00:00:09	00:00:05
8192	03:47:52	02:01:22	01:06:54	00:38:13	00:21:28	00:11:27	00:04:49	00:02:44	00:01:45	00:00:50
16384	35:50:59	21:06:42	08:26:28	04:52:10	02:58:04	01:34:56	00:48:27	00:20:56	00:15:03	00:11:23
32768	N/A	N/A	N/A	N/A	N/A	N/A	N/A	03:16:56	02:00:14	01:38:21

(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	1.6536	4.1730	8.2679	13.6804	23.0783	43.5082	48.2545	51.0385	120.6364
2048	1.0000	1.8659	3.9545	8.2700	18.0665	34.3360	60.9690	99.4008	145.9771	323.3671
4096	1.0000	2.0602	3.8276	5.7653	12.8988	27.2760	59.4127	116.6375	208.5176	391.6930
8192	1.0000	1.8776	3.4061	5.9631	10.6149	19.8984	47.2334	83.2806	130.6467	273.0613
16384	1.0000	1.6981	4.2471	7.3623	12.0802	22.6590	44.3924	102.7522	142.9730	188.9394
32768	N/A	N/A	N/A	N/A	N/A	N/A	N/A	128.0000	209.6383	256.2991

(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	0.8268	1.0432	1.0335	0.8550	0.7212	0.6798	0.3770	0.1994	0.2356
2048	1.0000	0.9329	0.9886	1.0337	1.1292	1.0730	0.9526	0.7766	0.5702	0.6316
4096	1.0000	1.0301	0.9569	0.7207	0.8062	0.8524	0.9283	0.9112	0.8145	0.7650
8192	1.0000	0.9388	0.8515	0.7454	0.6634	0.6218	0.7380	0.6506	0.5103	0.5333
16384	1.0000	0.8490	1.0618	0.9203	0.7550	0.7081	0.6936	0.8028	0.5585	0.3690
32768	N/A	N/A	N/A	N/A	N/A	N/A	N/A	1.0000	0.8189	0.5006

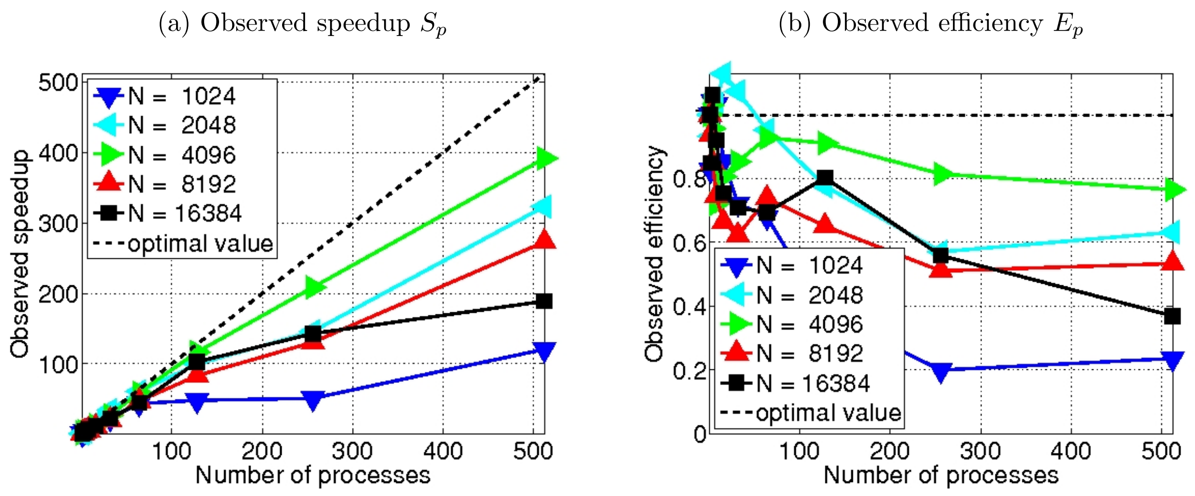


Figure 5.2: OpenMPI performance on tara by number of processes used with 2 processes per node, except for $p = 1$ which uses 1 process per node, $p = 256$ which uses 4 processes per node, and $p = 512$ which uses 8 processes per node.

Table 5.3: OpenMPI performance 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, $p = 512$ 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	00:00:27	00:00:16	00:00:09	00:00:03	00:00:02	00:00:01	00:00:01	00:00:00	00:00:01	00:00:00
2048	00:04:15	00:02:17	00:01:26	00:00:37	00:00:23	00:00:08	00:00:04	00:00:02	00:00:02	00:00:01
4096	00:30:37	00:14:52	00:10:31	00:05:09	00:02:38	00:01:47	00:00:50	00:00:17	00:00:09	00:00:05
8192	03:47:52	02:01:22	01:22:52	00:37:51	00:23:49	00:12:30	00:07:18	00:03:39	00:01:45	00:00:50
16384	35:50:59	21:06:42	12:22:37	06:21:11	02:49:45	01:37:28	00:59:10	00:29:55	00:15:03	00:11:23
32768	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	02:00:14	01:38:21

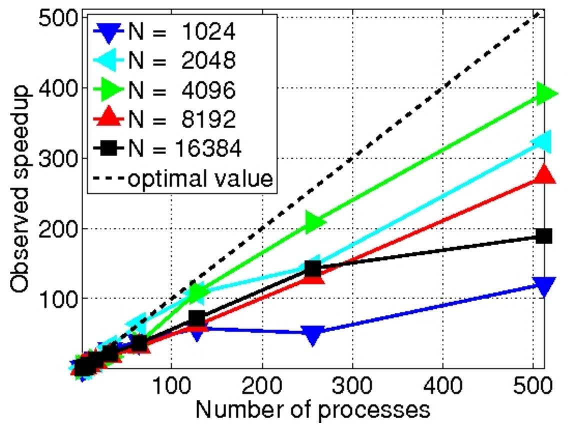
(b) Observed speedup S_p

N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$	$p = 256$	$p = 512$
1024	1.0000	1.6536	2.9456	8.1162	15.3410	27.3608	39.6119	57.6957	51.0385	120.6364
2048	1.0000	1.8659	2.9698	6.8820	11.0302	31.3064	64.0251	107.7890	145.9771	323.3671
4096	1.0000	2.0602	2.9108	5.9407	11.6327	17.2056	36.7041	109.0879	208.5176	391.6930
8192	1.0000	1.8776	2.7499	6.0214	9.5699	18.2237	31.2478	62.3560	130.6467	273.0613
16384	1.0000	1.6981	2.8965	5.6428	12.6716	22.0681	36.3517	71.8835	142.9730	188.9394
32768	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	256.0000	312.9798

(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	0.8268	0.7364	1.0145	0.9588	0.8550	0.6189	0.4507	0.1994	0.2356
2048	1.0000	0.9329	0.7424	0.8603	0.6894	0.9783	1.0004	0.8421	0.5702	0.6316
4096	1.0000	1.0301	0.7277	0.7426	0.7270	0.5377	0.5735	0.8522	0.8145	0.7650
8192	1.0000	0.9388	0.6875	0.7527	0.5981	0.5695	0.4882	0.4872	0.5103	0.5333
16384	1.0000	0.8490	0.7241	0.7054	0.7920	0.6896	0.5680	0.5616	0.5585	0.3690
32768	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	1.0000	0.6113

(a) Observed speedup S_p



(b) Observed efficiency E_p

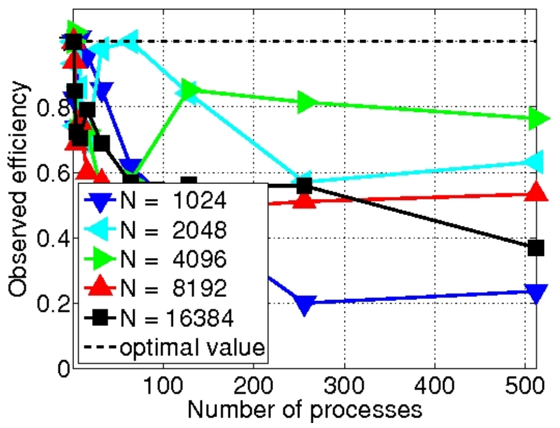


Figure 5.3: OpenMPI performance on tara by number of processes used with 4 processes per node, except for $p = 1$ which uses 1 process per node, $p = 2$ which uses 2 processes per node, and $p = 512$ which uses 8 processes per node.

Table 5.4: OpenMPI performance on tara by number of processes used with 6 processes per node, except for $p = 1$ which uses 1 process per node

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

N	$p = 1$	$p = 6$	$p = 12$	$p = 24$	$p = 48$	$p = 96$	$p = 192$	$p = 384$
1024	00:00:27	00:00:07	00:00:02	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01
2048	00:04:15	00:01:04	00:00:30	00:00:14	00:00:06	00:00:03	00:00:02	00:00:02
4096	00:30:37	00:07:12	00:03:42	00:03:16	00:01:18	00:00:45	00:00:14	00:00:07
8192	03:47:52	00:59:35	00:32:43	00:16:30	00:13:43	00:06:50	00:03:27	00:01:35
16384	35:50:59	08:05:50	04:35:21	02:18:58	01:08:19	00:42:47	00:27:52	00:14:04
32768	N/A	N/A	N/A	N/A	N/A	N/A	N/A	01:52:48

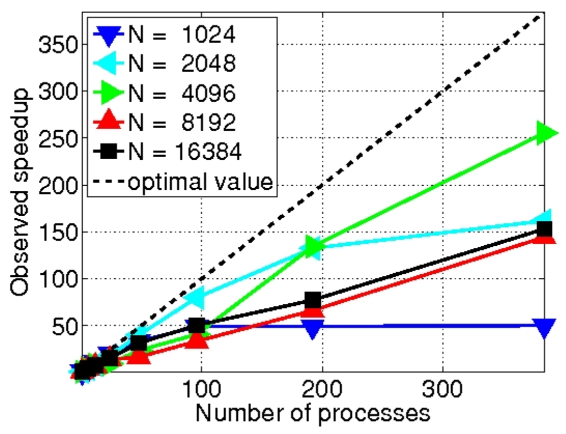
(b) Observed speedup S_p

N	$p = 1$	$p = 6$	$p = 12$	$p = 24$	$p = 48$	$p = 96$	$p = 192$	$p = 384$
1024	1.0000	3.8464	10.8327	20.1061	33.1750	49.1481	49.1481	50.0755
2048	1.0000	4.0230	8.5210	18.0155	39.8534	79.8312	132.3627	161.6835
4096	1.0000	4.2476	8.2668	9.3880	23.5096	40.7236	134.2865	255.1444
8192	1.0000	3.8246	6.9632	13.8104	16.6177	33.3696	66.1450	144.1757
16384	1.0000	4.4274	7.8116	15.4789	31.4869	50.2840	77.1888	152.9115
32768	N/A	N/A	N/A	N/A	N/A	N/A	N/A	384.0000

(c) Observed efficiency E_p

N	$p = 1$	$p = 6$	$p = 12$	$p = 24$	$p = 48$	$p = 96$	$p = 192$	$p = 384$
1024	1.0000	0.6411	0.9027	0.8378	0.6911	0.5120	0.2560	0.1304
2048	1.0000	0.6705	0.7101	0.7506	0.8303	0.8316	0.6894	0.4211
4096	1.0000	0.7079	0.6889	0.3912	0.4898	0.4242	0.6994	0.6644
8192	1.0000	0.6374	0.5803	0.5754	0.3462	0.3476	0.3445	0.3755
16384	1.0000	0.7379	0.6510	0.6450	0.6560	0.5238	0.4020	0.3982
32768	N/A	N/A	N/A	N/A	N/A	N/A	N/A	1.0000

(a) Observed speedup S_p



(b) Observed efficiency E_p

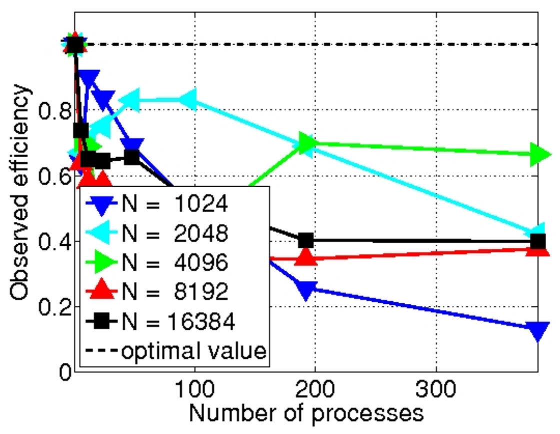


Figure 5.4: OpenMPI performance on tara by number of processes used with 6 processes per node, except for $p = 1$ which uses 1 process per node.

Table 5.5: OpenMPI performance 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	00:00:27	00:00:16	00:00:09	00:00:06	00:00:02	00:00:01	00:00:01	00:00:00	00:00:00	00:00:00
2048	00:04:15	00:02:17	00:01:26	00:00:53	00:00:24	00:00:13	00:00:05	00:00:02	00:00:01	00:00:01
4096	00:30:37	00:14:52	00:10:31	00:06:30	00:03:17	00:01:48	00:01:04	00:00:28	00:00:10	00:00:05
8192	03:47:52	02:01:22	01:22:52	00:54:21	00:26:33	00:14:33	00:09:04	00:05:34	00:02:40	00:00:50
16384	35:50:59	21:06:42	12:22:37	08:11:19	03:36:55	01:55:42	00:58:25	00:35:20	00:17:47	00:11:23
32768	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	01:38:21

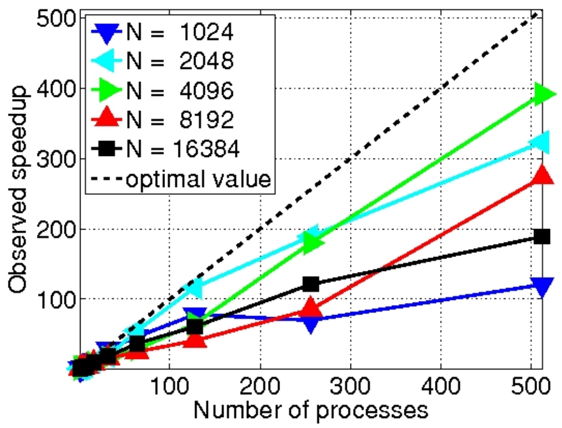
(b) Observed speedup S_p

N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$	$p = 256$	$p = 512$
1024	1.0000	1.6536	2.9456	4.5368	13.3367	28.8478	46.5614	78.0588	69.8421	120.6364
2048	1.0000	1.8659	2.9698	4.8465	10.7201	19.2509	55.0560	116.6484	189.2296	323.3671
4096	1.0000	2.0602	2.9108	4.7152	9.3279	17.0887	28.7397	64.9820	179.9256	391.6930
8192	1.0000	1.8776	2.7499	4.1931	8.5812	15.6678	25.1285	40.9322	85.2752	273.0613
16384	1.0000	1.6981	2.8965	4.3781	9.9165	18.5906	36.8164	60.8708	120.9866	188.9394
32768	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	512.0000

(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	0.8268	0.7364	0.5671	0.8335	0.9015	0.7275	0.6098	0.2728	0.2356
2048	1.0000	0.9329	0.7424	0.6058	0.6700	0.6016	0.8603	0.9113	0.7392	0.6316
4096	1.0000	1.0301	0.7277	0.5894	0.5830	0.5340	0.4491	0.5077	0.7028	0.7650
8192	1.0000	0.9388	0.6875	0.5241	0.5363	0.4896	0.3926	0.3198	0.3331	0.5333
16384	1.0000	0.8490	0.7241	0.5473	0.6198	0.5810	0.5753	0.4756	0.4726	0.3690
32768	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	1.0000

(a) Observed speedup S_p



(b) Observed efficiency E_p

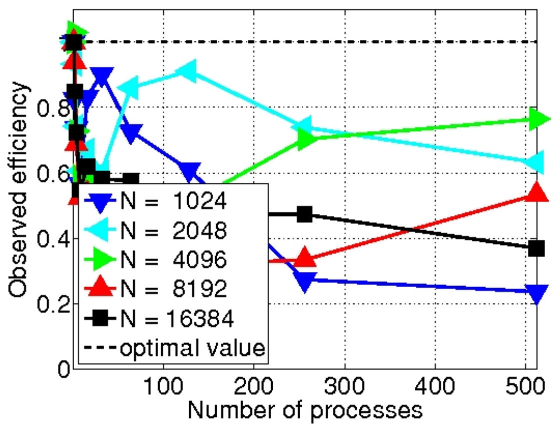


Figure 5.5: OpenMPI performance on tara by number of processes used with 8 processes per node, except for $p = 1$ which uses 1 process per node, $p = 2$ which uses 2 processes per node, and $p = 4$ which uses 4 processes per node.