Parallel Performance Studies for an Elliptic Test Problem on maya 2013

Samuel Khuvis and Matthias K. Gobbert (gobbert@umbc.edu)

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

Technical Report HPCF–2014–6, www.umbc.edu/hpcf > Publications

Abstract

The UMBC High Performance Computing Facility (HPCF) is the community-based, interdisciplinary core facility for scientific computing and research on parallel algorithms at UMBC. Released in Summer 2014, the current machine in HPCF is the 240-node distributed-memory cluster maya. The cluster is comprised of three uniform portions, one consisting of 72 nodes based on 2.6 GHz Intel E5-2650v2 Ivy Bridge CPUs from 2013, another consisting of 84 nodes based on 2.8 GHz Intel Nehalem X5560 CPUs from 2010, and another consisting of 84 nodes based on 2.6 GHz Intel Nehalem X5550 CPUs from 2009. All nodes are connected via InfiniBand to a central storage of more than 750 TB.

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 and classical test case of this interplay. This note considers the classical elliptic test problem of a Poisson equation with homogeneous 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 does not create a bottleneck.

We report parallel performance studies on the newest portion of maya, referred to as maya 2013. The results show very good performance up to 64 compute nodes and support several key conclusions: (i) The newer nodes are faster per core as well as per node, however, for most serial production code using one of the 2010 nodes with 2.8 GHz is a good default. (ii) The high-performance interconnect supports parallel scalability on at least 64 nodes optimally. (iii) It is often faster to use all cores on modern multi-core nodes but it is useful to track memory to determine if this is the case for memory-bound code. (iv) There is no disadvantage to several jobs sharing a node, which justifies the default scheduling setup.

1 Introduction

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

Released in Summer 2014, the current machine in HPCF is the 240-node distributed-memory cluster maya. The newest components of the cluster are the 72 nodes in maya 2013 with two eight-core 2.6 GHz Intel E5-2650v2 Ivy Bridge CPUs and 64 GB memory that include 19 hybrid nodes with two state-of-the-art NVIDIA K20 GPUs (graphics processing units) designed for scientific computing and 19 hybrid nodes with two cutting-edge 60-core Intel Phi 5110P accelerators. These new nodes are connected along with the 84 nodes in maya 2009 with two quad-core 2.6 GHz Intel Nehalem X5550 CPUs and 24 GB memory by a high-speed quad-data rate (QDR) InfiniBand network for research on parallel algorithms. The remaining 84 nodes in maya 2010 with two quad-core 2.8 GHz Intel Nehalem X5560 CPUs and 24 GB memory are designed for fastest number crunching and connected by a dual-data rate (DDR) InfiniBand network. All nodes are connected via InfiniBand to a central storage of more than 750 TB.

The studies in this reports use default Intel C compiler version 14.0 (compiler options `-std=c99 -Wall -O3`) with Intel MPI version 4.1. All results in this report use dedicated nodes with remaining cores idling using the `--exclusive` option in the SLURM submission script. The default is to use `--shared`, which allocates all processes to cores on one CPU, while `--exclusive` allocates tasks to cores on both CPUs. There is no significant advantage to `--shared` for production runs, that is, performance studies are the only time that this option should be used.

This report is an update to the technical report [8], which considered the same problem on the previous cluster tara. The problem is the numerical solution of the Poisson equation with homogeneous Dirichlet boundary conditions

Table 1.1: Wall clock time in HH:MM:SS on maya 2013 for mesh resolution $N \times N = 16384 \times 16384$.

$N \times N = 16384 \times 16384$	1 node	2 nodes	4 nodes	8 nodes	16 nodes	32 nodes	64 nodes
1 process per node	14:08:26	06:57:26	03:29:58	01:45:31	00:53:29	00:27:13	00:13:57
2 processes per node	07:01:30	03:31:53	01:46:36	00:54:13	00:27:20	00:14:12	00:07:15
4 processes per node	03:55:38	01:58:32	01:00:05	00:30:26	00:15:32	00:08:15	00:04:20
8 processes per node	02:55:26	01:28:52	00:44:32	00:22:30	00:11:26	00:06:23	00:03:22
16 processes per node	02:49:13	01:25:16	00:43:08	00:22:10	00:11:33	00:06:31	00:06:34

on a unit square domain in two spatial dimensions. Discretizing the spatial derivatives by the finite difference method yields a system of linear equations with a large, sparse, highly structured, symmetric positive definite system matrix. This linear system is a classical test problem for iterative solvers and contained in several textbooks including [3, 5, 6, 10]. The parallel, matrix-free implementation of the conjugate gradient method as appropriate iterative linear solver for this linear system involves necessarily communications both collectively among all parallel processes and between pairs of processes in every iteration. Therefore, this method provides an excellent test problem for the overall, real-life performance of a parallel computer, and we used it in the past to analyze previous clusters [1, 4, 8, 9] These results 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. 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.

Table 1.1 contains an excerpt of the performance results reported in Table 4.1 of Section 4 for the studies on the newest portion maya 2013 of the cluster. This excerpt reports the results for one mesh resolution and using the default compiler and MPI implementation. Table 1.1 reports the observed wall clock time in HH:MM:SS (hours:minutes:seconds) for all possible combinations of numbers of nodes and processes per node (that are powers of 2), that is, for $1, 2, 4, 8, 16, 32$, and 64 nodes and $1, 2, 4, 8$, and 16 processes per node. It is conventional to restrict studies to powers of 2, since this makes it easy to judge if timings are halved when the number of parallel processes is doubled. We observe that by simply using all cores on one node we can reduce the runtime from approximately 14 hours to 14 minutes and by using 8 cores on 64 nodes we can reduce the runtime to under 4 minutes. This table demonstrates the power of parallel computing, in which by pooling the memory of several compute node to solve larger problems and to dramatically speed up the solution time. But it also demonstrates the potential for further advances: The studies in Table 1.1 only used the CPUs of the compute nodes; using accelerators such as the GPUs and the Intel Phi have the potential to shorten the runtimes even more.

More in detail, by reading along a row of Table 1.1, we see that the high-performance QDR InfiniBand interconnect supports parallel scalability on at least 64 nodes optimally, since each timing halves for each doubling of numbers of nodes. In turn, reading along a column of Table 1.1, it is clear that for jobs on a small number of nodes run fastest when using all 16 cores of each compute node, but for jobs on larger number of nodes run fastest when using 8 cores of each compute node. As we will discuss in greater detail in Section 4, we observe less than optimal halving of runtime by increasing the number of processes from 8 to 16 so memory usage should be tracked to determine the correct number of processes per node to use.

The remainder of this report is organized as follows: Section 2 details the test problem and discusses the parallel implementation in more detail, and Section 3 summarizes the solution and method convergence data. Section 4 contains the complete parallel performance studies on maya 2013, from which Table 1.1 was excerpted. Results on all three portions of maya and a comparison to previous clusters are contained in the report [7].

Acknowledgments

The hardware used in the computational studies is part of the UMBC High Performance Computing Facility (HPCF). The facility is supported by the U.S. National Science Foundation through the MRI program (grant nos. CNS–0821258 and CNS–1228778) 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. The first author 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., [10, Chapter 8])

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

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

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

for $k_i = 1, \ldots, N$, $i = 1, \ldots, d$, 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 $q = A\,p$ also computes only the block of the vector q that is local to each process. But since the matrix A has non-zero off-diagonal elements, each local block needs values of p that are local to the two processes that hold the neighboring blocks of p. The communications between parallel processes thus needed are so-called point-to-point communications, because not all processes participate in each of them, but rather only specific pairs of processes that exchange data needed for their local calculations. Observe now that it is only a few components of q that require data from p that is not local to the process. Therefore, it is possible and potentially very efficient to proceed to calculate those components that can be computed from local data only, while the communications with the neighboring processes are taking place. This technique is known as interleaving calculations and communications and can be implemented using the non-blocking MPI communications commands `MPI_Isend` and `MPI_Irecv`.

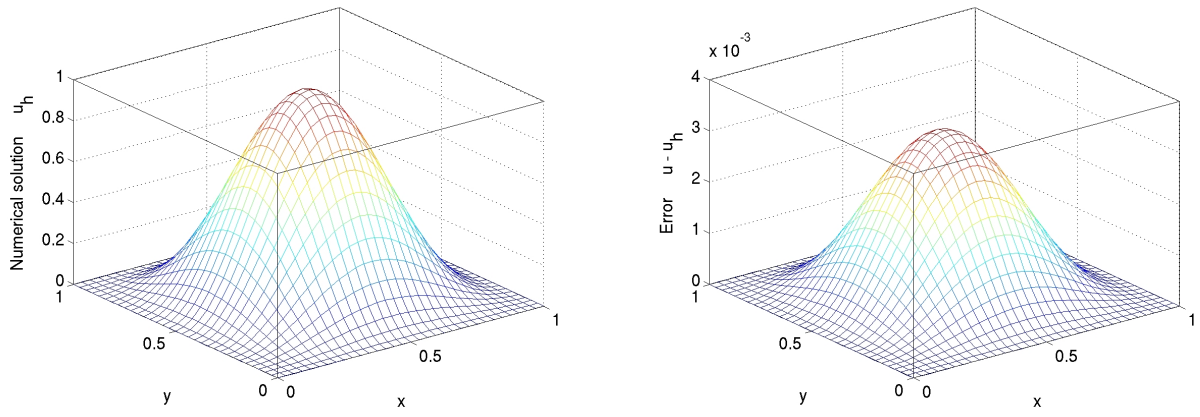
3 Convergence Study for the Model Problem

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 $\|u - u_h\| \equiv \|u - u_h\|_{L^\infty(\Omega)}$, the ratio of consecutive errors $\|u - u_{2h}\| / \|u - u_h\|$, the number of conjugate gradient iterations `#iter`, the observed wall clock time in HH:MM:SS and in seconds, and the predicted and observed memory usage in GB for studies performed in serial. More precisely, the 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



(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 the Intel compiler with Intel MPI with serial code except where noted).

N	DOF	$\|u - u_h\|$	Ratio	`#iter`	wall clock time		memory usage (GB)	
					HH:MM:SS	seconds	predicted	observed
32	1,024	3.0128e–03	N/A	48	<00:00:01	< 0.01	< 1	< 1
64	4,096	7.7811e–04	3.87	96	<00:00:01	< 0.01	< 1	< 1
128	16,384	1.9765e–04	3.94	192	<00:00:01	0.01	< 1	< 1
256	65,536	4.9797e–05	3.97	387	<00:00:01	0.10	< 1	< 1
512	262,144	1.2494e–05	3.99	783	<00:00:01	0.81	< 1	< 1
1024	1,048,576	3.1266e–06	4.00	1,581	00:00:09	9.32	< 1	< 1
2048	4,194,304	7.8019e–07	4.01	3,192	00:01:34	94.18	< 1	< 1
4096	16,777,216	1.9366e–07	4.03	6,452	00:12:25	745.84	< 1	< 1
8192	67,108,864	4.7377e–08	4.09	13,033	01:41:07	6,067.00	2	2.02
16384	268,435,456	1.1547e–08	4.10	26,316	14:08:25	50905.78	8	8.02
32768	1,073,741,824	1.7321e–09	6.67	53,141	117:02:56	421375.93	32	32.02
*65536	4,294,967,296	8.9078e–10	1.94	107,261	*03:09:24	*11,364.40	128	*139.41

*The case $N = 65536$ uses 8 cores on 64 nodes; the observed memory is the total over all processes.

communication commands disabled by if-statements. The wall clock time is measured using the `MPI_Wtime` command (after synchronizing all processes by an `MPI_Barrier` command). The memory usage of the code is predicted by noting that there are $4N^2$ double-precision numbers needed to store the four vectors of significant length N^2 and that each double-precision number requires 8 bytes; dividing this result by 1024^3 converts its value to units of GB, as quoted in the table. The memory usage is observed in the code by checking the `VmRSS` field in the the special file `/proc/self/status`. The case $N = 65536$ requires more memory than is available on a compute node with 64 GB. For this case, 8 cores on 64 nodes are used, with observed memory summed across all running processes to get the total usage.

In nearly all cases, the norms of the finite difference errors in Table 3.1 decrease by a factor of about 4 each time that the mesh is refined by a factor 2. This confirms that the finite difference method is second-order convergent, as predicted by the numerical theory for the finite difference method [2, 6]. The fact that this convergence order is attained also confirms that the tolerance of the iterative linear solver is tight enough to ensure a sufficiently accurate solution of the linear system. For the two finest mesh resolutions, the reduction in error appears slightly 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. For the cases $N \leq 32768$, the observed memory usage in units of GB rounds to within 0.02 GB of the predicted usage, while the 64-node run for $N = 65536$ is larger due to the accumulation of overhead from all parallel processes. 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 in the serial case. The wall clock times and the memory usages for these serial runs indicate for which mesh resolutions this elliptic test problem becomes challenging computationally. Notice that the very fine meshes show very significant runtimes and memory usage; parallel computing clearly offers opportunities to decrease runtimes as well as to decrease memory usage per process by spreading the problem over the parallel processes.

We 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 [4] 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. It also already clear from Table 3.1 that the larger memory of the new nodes in maya allows for the first time the solution of the $N = 32768$ case in serial and of the $N = 65536$ at all.

4 Performance Studies on maya 2013

This section describes the parallel performance studies for the solution of the test problem on the 2013 portion of maya. The 72 nodes of this portion are set up as 67 compute nodes, 2 develop nodes, 1 user node, and 1 management node. Figure 4.1 shows a schematic of one of the compute nodes that is made up of two eight-core 2.6 GHz Intel E5-2650v2 Ivy Bridge CPUs. Each core of each CPU has dedicated 32 kB of L1 and 256 kB of L2 cache. All eight cores of each CPU share 20 MB of L3 cache. The 64 GB of the node's memory is formed by eight 8 GB DIMMs, four of which are connected to each CPU. The two CPUs of a node are connected to each other by two QPI (quick path interconnect) links. The nodes in maya 2013 are connected by a quad-data rate InfiniBand interconnect.

The results in this section use the default Intel compiler and Intel MPI. The SLURM submission script uses the `srun` command to start the job. The number of nodes are controlled by the `--nodes` option in the SLURM submission script, and the number of processes per node by the `--ntasks-per-node` option. Each node that is used is dedicated to the job with remaining cores idling, if not all of them are used using `--exclusive`. The assignment of the MPI processes to the cores of the two CPUs on the node uses the default assignment, in which consecutive processes are distributed in alternating fashion between the two CPUs.

We conduct numerical experiments of the test problem for seven progressively finer meshes of $N = 1024$, 2048, 4096, 8192, 16384, 32768, and 65536. This results in progressively larger systems of linear equations with system dimensions ranging from about 1 million for $N = 1024$ to over 1 billion for $N = 32768$ and over 4 billion equations for $N = 65536$. For each mesh resolution, the parallel implementation of the test problem is run on all possible combinations of nodes from 1 to 64 by powers of 2 and processes per node from 1 to 16 by powers of 2, except for the case of $N = 65536$. As shown in Table 3.1, cases up to $N = 32768$ take up to 32 GB of memory and fit in the memory of one compute node with 64 GB, but the case of $N = 65536$ is estimated to require at least 128 GB and does not.

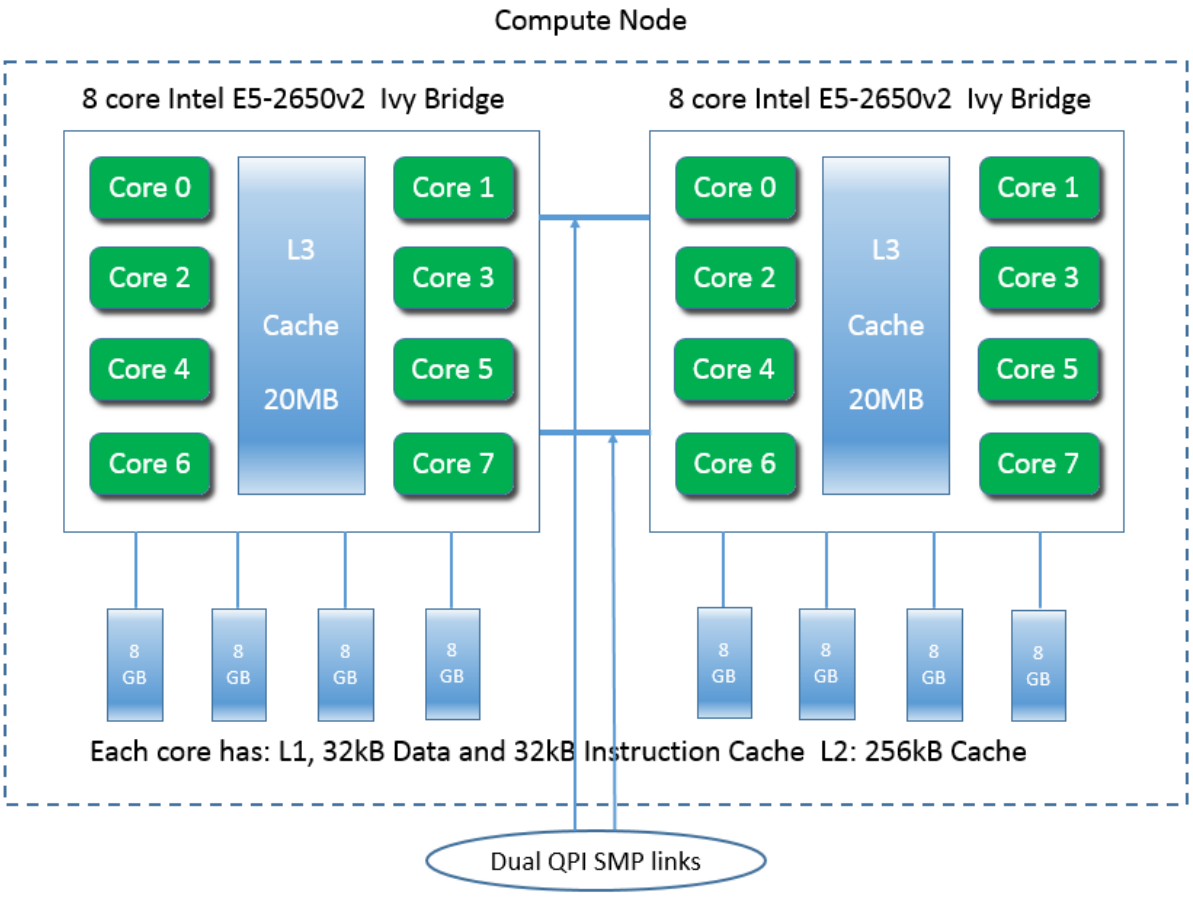


Figure 4.1: Schematic of a maya 2013 node.

Table 4.1 collects the results of the performance study. The table summarizes the observed wall clock time (total time to execute the code) in HH:MM:SS (hours:minutes:seconds) format. The upper-left entry of each subtable contains the runtime for the serial run of the code for that particular mesh. The lower-right entry of each subtable lists the runtime using all cores of both 8-core processors on 64 nodes for a total of 1024 parallel processes working together to solve the problem.

We choose the mesh resolution of 16384×16384 to discuss in detail as example. Reading along the first column of this mesh subtable, we observe that by doubling the number of processes from 1 to 2 we approximately halve the runtime from each column to the next. We observe the same improvement from 2 to 4 processes. We also observe that by doubling the number of processes from 4 to 8 there is still a significant improvement in runtime, although not the halving we observed previously. Finally, by doubling the number of processes from 8 to 16 we observe only a small improvement in runtime, which is a typical characteristic of memory-bound code such as this. The limiting factor in performance of memory-bound code is memory access, so we would expect a bottleneck when the 8 processes on each CPU attempt to access the memory through only 4 memory channels.

Reading along the first row of the 16384×16384 mesh subtable, we observe that by doubling the number of nodes used, and thus also doubling the number of parallel processes, we approximately halve the runtime all the way up to 64 nodes. This behavior observed for increasing the number of nodes confirms the quality of the high-performance InfiniBand interconnect.

Now, if we read along any of the other columns of this subtable, we observe similar behavior as in the first column. By doubling the number of processes per node from 1 to 2 and from 2 to 4 we halve the runtime. By doubling the number of processes per node from 4 to 8 we still observe a significant improvement in runtime, although less than the halving observed previously. However, for the case of 16 processes per node the behavior changes as the number of nodes increases. On 1, 2, 4, and 8 nodes, we still observe a small improvement in runtime from 8 to 16 processes per node. The improvement in runtime then deteriorates as the number of nodes increases until jobs on 16, 32, and 64 nodes experience an increase in runtime from 8 to 16 processes per node.

The other subtables in Table 4.1 exhibit largely analogous behavior to the 16384×16384 mesh. In particular, the 32768×32768 mesh subtable shows similar performance behavior, except that the performance for 16 processes per node compared to 8 processes per node on 64 nodes deteriorates more drastically. For this reason, we restrict our computations for the 65536×65536 mesh to the configuration of 8 processes per node on 64 nodes only, which results in the runtime of 03:09:04 reported in Table 3.1. The underlying reason for the severe deterioration of performance for 16 processes per node compared on 8 processes per node when using 64 nodes is not clear, but memory observations for the total memory indicate significant overhead associated with this many MPI processes, which might point to an explanation.

The technical report [8] considers the same problem on the previous cluster tara. Both reports contain results for meshes of $N = 1024$, 2048, 4096, 8192, and 16384 on 1, 2, 4, 8, 16, 32, and 64 nodes using 1, 2, 4, and 8 processes per node. Since full results for $N = 16384$ are available on both clusters we will compare results for this mesh. We observe a significant improvement in runtime from the results on tara to those on maya 2013. In fact, we observe at least a halving in runtime on maya 2013 compared to tara for most runs on this mesh. Since the compute nodes in maya 2013 have 64 GB of memory, a serial run for $N = 32768$ is possible for the first time. For this mesh, we observe that for results available on tara, running on maya 2013 again results in at least a halving in runtime. As a result, this report contains a complete subtable of results for $N = 32768$ while [8] contains results only on 64 nodes. Also, this report contains results for 16 processes per node, which was not possible on the cluster tara since the compute nodes on tara only contained 8 cores per node. Finally, this report contains results for $N = 65536$ for the first time since the increase in memory allows these jobs to be completed in a reasonable amount of time.

Overall, we can conclude that using larger numbers of nodes shows optimal speedup up to the number of nodes available on maya 2013. For memory-bound code, memory usage should be tracked to determine if there is any disadvantage to using all 16 cores on a node.

Table 4.1: Wall clock time in HH:MM:SS on maya 2013 using the Intel compiler with Intel MPI.

(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	00:00:09	00:00:03	00:00:02	00:00:01	00:00:01	00:00:00	00:00:00
2 processes per node	00:00:03	00:00:02	00:00:01	00:00:01	00:00:00	00:00:00	00:00:00
4 processes per node	00:00:02	00:00:01	00:00:01	00:00:00	00:00:00	00:00:00	00:00:00
8 processes per node	00:00:01	00:00:01	00:00:00	00:00:00	00:00:00	00:00:01	00:00:01
16 processes per node	00:00:01	00:00:00	00:00:00	00:00:01	00:00:01	00:00:01	00:00:01

(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	00:01:34	00:00:46	00:00:20	00:00:07	00:00:04	00:00:02	00:00:01
2 processes per node	00:00:47	00:00:19	00:00:07	00:00:04	00:00:02	00:00:01	00:00:01
4 processes per node	00:00:28	00:00:11	00:00:04	00:00:02	00:00:02	00:00:01	00:00:01
8 processes per node	00:00:21	00:00:08	00:00:02	00:00:02	00:00:01	00:00:01	00:00:01
16 processes per node	00:00:20	00:00:07	00:00:02	00:00:02	00:00:02	00:00:02	00:00:03

(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	00:12:26	00:06:18	00:03:11	00:01:32	00:00:41	00:00:15	00:00:09
2 processes per node	00:06:18	00:03:14	00:01:36	00:00:41	00:00:15	00:00:09	00:00:05
4 processes per node	00:03:30	00:01:50	00:00:57	00:00:25	00:00:09	00:00:05	00:00:04
8 processes per node	00:02:44	00:01:24	00:00:44	00:00:20	00:00:06	00:00:04	00:00:04
16 processes per node	00:02:44	00:01:25	00:00:45	00:00:17	00:00:06	00:00:12	00:00:09

(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	01:41:07	00:50:52	00:25:47	00:13:13	00:06:30	00:03:21	00:01:29
2 processes per node	00:51:00	00:25:49	00:13:09	00:06:39	00:03:25	00:01:29	00:00:35
4 processes per node	00:28:29	00:14:43	00:07:33	00:03:54	00:02:02	00:00:55	00:00:22
8 processes per node	00:21:28	00:10:56	00:05:38	00:02:58	00:01:29	00:00:41	00:00:17
16 processes per node	00:20:57	00:10:42	00:05:38	00:02:48	00:01:37	00:00:44	00:00:32

(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	14:08:26	06:57:26	03:29:58	01:45:31	00:53:29	00:27:13	00:13:57
2 processes per node	07:01:30	03:31:53	01:46:36	00:54:13	00:27:20	00:14:12	00:07:15
4 processes per node	03:55:38	01:58:32	01:00:05	00:30:26	00:15:32	00:08:15	00:04:20
8 processes per node	02:55:26	01:28:52	00:44:32	00:22:30	00:11:26	00:06:23	00:03:22
16 processes per node	02:49:13	01:25:16	00:43:08	00:22:10	00:11:33	00:06:31	00:06:34

(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	117:02:56	56:06:06	28:16:00	14:05:44	07:04:47	03:34:15	01:49:15
2 processes per node	56:21:35	28:25:53	14:13:30	07:09:54	03:37:11	01:50:35	00:56:55
4 processes per node	32:05:13	16:08:41	08:04:43	04:01:56	02:02:31	01:02:51	00:32:32
8 processes per node	23:53:02	11:52:48	05:56:34	03:00:00	01:30:57	00:46:32	00:23:40
16 processes per node	22:53:13	11:29:47	05:47:00	02:55:12	01:29:07	00:46:37	02:55:46

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

Table 4.2 organizes the results of Table 4.1 in the form of a strong scalability study, that is, there is one row for each problem size, with columns for increasing number of parallel processes p. Table 4.2 (a) lists the raw timing data, like Table 4.1, but organized by numbers of parallel processes p. Tables 4.2 (b) and (c) show the numbers for speedup and efficiency, respectively, that will be visualized in Figures 4.2 (a) and (b), respectively. It becomes clear that there are several choices for most values of p, such as for instance for $p = 4$, one could use 2 nodes with 2 processes per node or 1 node with 4 processes per node. Due to our observation that 8 processes per node performs better than 16 processes per node in certain situations for memory-bound code, for $p \geq 8$ we use 8 processes per node, until the final $p = 1024$ that can only be achieved by 16 processes on 64 nodes. For $p < 8$, only one node is used, with the remaining cores idle. Comparing adjacent columns in the raw timing data in Table 4.2 (a) confirms our previous observation that performance improvement is very good from 1 to 2 processes and from 2 to 4 processes, but not quite as good from 4 to 8 processes. Since the table collects the results using 8 processes per node, the next columns double the numbers of nodes, and we see near-perfect halving of runtimes, except the very last column that uses 16 processes per node on 64 nodes. The speedup numbers in Table 4.2 (b) help reach the same conclusions when speedup is near-optimal with $S_p \approx p$ for $p \leq 8$. For $p = 16$, sub-optimal speedup is clear. The speedup numbers also indicate sub-optimal speedup for $p > 16$, but recall that the runtimes clearly showed halving from each column to the next one; the speedup numbers can only give this indication qualitatively. The efficiency data in Table 4.2 (c) can bring out these effects more quantitatively, namely efficiency is near-optimal $E_p \approx 1$ for $p \leq 8$, then clearly identifies the efficiency drop taking place from $p = 8$ to $p = 16$. But for $p > 16$, the efficiency numbers stay essentially constant, which confirms quantitatively the aforementioned halving of runtimes from each column to the next one. The final column shows again the dramatic drop-off in performance when going from 8 to 16 processes on 64 nodes.

The plots in Figures 4.2 (a) and (b) visualize the numbers in Tables 4.2 (b) and (c), respectively. These plots do not provide new data but simply provide a graphical representation of the results in Table 4.1. It is customary in results for fixed problem sizes that the speedup is better for larger problems, since the increased communication time for more parallel processes does not dominate over the calculation time as quickly as it does for small problems. This is born out generally by both plots in Figure 4.2. Specifically, the speedup in Figure 4.2 (a) appears near-optimal up to $p = 512$ for all problem sizes $N \geq 4096$. From $p = 512$ to $p = 1024$, we see the expected dramatic decrease in speedup that the raw run times exhibit. One would expect that the efficiency plot in Figure 4.2 (b) would not add much clarity, since its data are directly derived from the speedup data. But the efficiency plot can provide insight into behavior for small p, where the better-than-optimal behavior is noticable now. This can happen due to experimental variability of the runs, for instance, if the single-process timing $T_1(N)$ used in the computation of $S_p = T_1(N)/T_p(N)$ happens to be slowed down in some way. Another reason for excellent performance can also be that runs on several processes result in local problems that fit better into the cache of each processor, which leads to fewer cache misses and thus potentially dramatic improvement of the run time, beyond merely distributing the calculations to more processes. For larger values of p, excluding the final value at $p = 1024$, the horizontal shape of the lines in the efficiency plot brings out that no further degradation of performance occurs as p increases for large N. Figure 4.2 (b) also exhibits again the significant reduction in efficiency by going from 8 to 16 processes per node on 64 nodes.

Table 4.2: Intel compiler with Intel MPI performance on maya 2013 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, $p = 4$ which uses 4 processes per node, and $p = 1024$ which uses 16 processes per node.

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

N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$	$p = 256$	$p = 512$	$p = 1024$
1024	00:00:09	00:00:03	00:00:02	00:00:01	00:00:01	00:00:00	00:00:00	03:00:00	00:00:01	00:00:01	00:00:01
2048	00:01:34	00:00:47	00:00:28	00:00:21	00:00:08	00:00:02	00:00:02	00:00:01	00:00:01	00:00:01	00:00:03
4096	00:12:26	00:06:18	00:03:30	00:02:44	00:01:24	00:00:44	00:00:20	00:00:06	00:00:04	00:00:04	00:00:09
8192	01:41:07	00:51:00	00:28:29	00:21:28	00:10:56	00:05:38	00:02:58	00:01:29	00:00:41	00:00:17	00:00:32
16384	14:08:26	07:01:30	03:55:38	02:55:26	01:28:52	00:44:32	00:22:30	00:11:26	00:06:23	00:03:22	00:06:34
32768	117:02:56	56:21:35	32:05:13	23:53:02	11:52:48	05:56:34	03:00:00	01:30:57	00:46:32	00:23:40	02:55:46

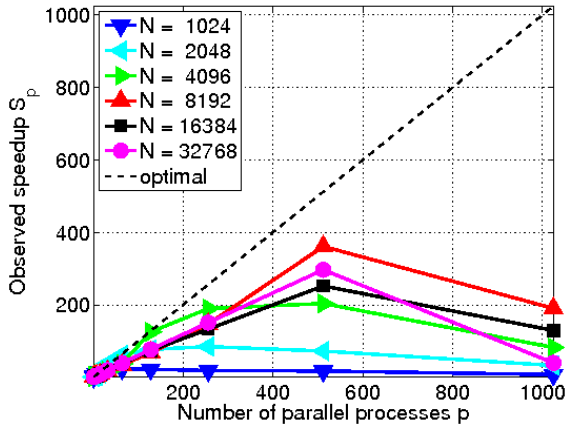
(b) Observed speedup S_p

N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$	$p = 256$	$p = 512$	$p = 1024$
1024	1.00	2.78	5.24	9.05	15.80	22.73	23.90	22.73	18.64	17.58	7.58
2048	1.00	2.02	3.41	4.57	11.23	38.28	61.56	77.20	84.85	72.45	32.93
4096	1.00	1.97	3.56	4.55	8.87	16.95	37.46	125.35	189.78	203.78	82.41
8192	1.00	1.98	3.55	4.71	9.24	17.98	34.05	67.91	147.23	361.15	190.32
16384	1.00	2.01	3.60	4.84	9.55	19.05	37.72	74.18	132.78	252.07	129.32
32768	1.00	2.08	3.65	4.90	9.85	19.70	39.02	77.21	150.90	296.70	39.95

(c) Observed efficiency E_p

N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$	$p = 256$	$p = 512$	$p = 1024$
1024	1.00	1.39	1.31	1.13	0.99	0.71	0.37	0.18	0.07	0.03	0.01
2048	1.00	1.01	0.85	0.57	0.70	1.20	0.96	0.60	0.33	0.14	0.03
4096	1.00	0.99	0.89	0.57	0.55	0.53	0.59	0.98	0.74	0.40	0.08
8192	1.00	0.99	0.89	0.59	0.58	0.56	0.53	0.53	0.58	0.71	0.19
16384	1.00	1.01	0.90	0.60	0.60	0.60	0.59	0.58	0.52	0.49	0.13
32768	1.00	1.04	0.91	0.61	0.62	0.62	0.61	0.60	0.59	0.58	0.04

(a) Observed speedup S_p



(b) Observed efficiency E_p

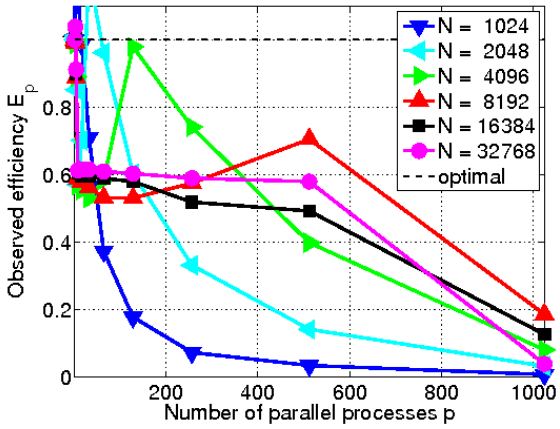


Figure 4.2: Intel compiler with Intel MPI performance on maya 2013 by number of processes used with 8 processes per node, except for $p = 1$ which uses 1 process per node, $p = 2$ which uses 2 processes per node, $p = 4$ which user 4 processes per node, and $p = 1024$ which used 16 processes per node.

References

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

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

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

[4] 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.

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

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

[7] Samuel Khuvis and Matthias K. Gobbert. Parallel performance studies for an elliptic test problem on the cluster maya. Technical Report HPCF–2015–6, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2015.

[8] 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.

[9] Hafez Tari and Matthias K. Gobbert. A comparative study of the parallel performance of the blocking and non-blocking MPI communication commands on an elliptic test problem on the cluster tara. Technical Report HPCF–2010–6, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2010.

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