Update dated August 28, 2014

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

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.

We report parallel performance studies on each of the three uniform portions of the cluster maya. The results show very good performance up to 64 compute nodes on all portions 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 optimal to use all cores on a compute node. (iv) There is no disadvantage to several jobs sharing a node, which justifies the default scheduling setup. (v) In concert with this, the default behavior of assigning all processes of a job to one CPU (up to the available number of cores) is reasonable.

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 resport use default comp the Intel C compiler version 14.0 (compiler options -std=c99 -Wall -O3) with Intel MPI version 4.1.

This report is an update to the technical report [7], 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 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.

Table 1.1: Wall clock time in HH:MM:SS on maya (2013) using the Intel compiler with Intel MPI for mesh resolution $N \times N = 32768 \times 32768$.

This linear system is a classical test problem for iterative solvers and contained in several textbooks including [3, 5, 6, 9]. 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, 7, 8] 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. Note that it was not possible to compute the solution for this mesh in serial on previous clusters. Table 1.1 reports the observed wall clock time in HH:MM:SS 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 5 days to 23 hours and by using all cores on 64 nodes we can reduce the runtime to just 26 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 computer nodes; using accelerators such as the GPUs and the Intel Phi have the potential to shorten the run times 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 the job runs fastst,when using all 16 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 processses from 4 to 8. Since each node contains two 8-core CPUs, the default behavior of assigning all 8 processes to one CPU results in a bottleneck when these processes attempt to access memory through the 4 memory channels. However, this behavior is a reasonable default, since it allows more than one job to be run on a single node without competition for access to memory.

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. Finally, Section 5 provides a historical comparison of performance of maya and the previous clusters in HPCF.

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., [9, 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 $S = \operatorname{tridiag}(-1, 4, -1) \in \mathbb{R}^{N \times N}$ 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`.

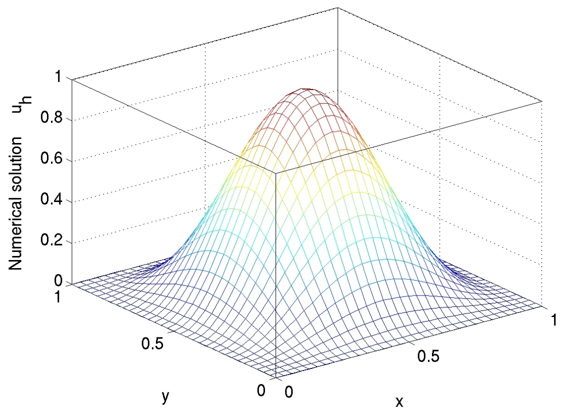
3 Convergence Study for the Model Problem

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

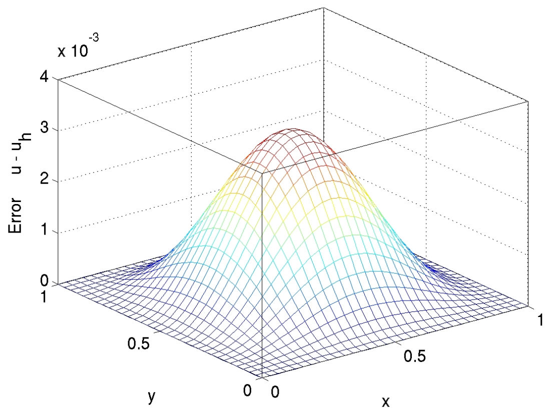
$$f(x_1, x_2) = (-2\pi^2) \Big(\cos(2\pi x_1)\, \sin^2(\pi x_2) + \sin^2(\pi x_1)\, \cos(2\pi x_2) \Big), \tag{3.1}$$

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 $h^2 \approx 0.98$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).

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	2.00
16384	268,435,456	1.1547e–08	4.10	26,316	14:08:25	50905.78	8	8.00

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. All cores on 64 nodes are used for this case, 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 exactly 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 and that there is little overhead memory cost. 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 [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.

4 Performance Studies on maya 2013

This section describes the parallel performance studies on maya (2013) for the solution of the test problem. Figure 4.1 contains a schematic of a maya (2013) node. A maya (2013) node consists of two eight-core 2.6 GHz Intel E5-2650v2 Ivy Bridge CPUs. Each CPU is connected to 32 GB of DDR3 memory through four memory channels. The two CPUs are connected by two quick path interconnect (QPI) links. Nodes are connected by a quad-data rate Infiniband connection.

The results in this section use the default assignment of MPI processes to the cores on the node. The default behavior is to first allocate processes to the cores in a single CPU. Only in the case of a job that requires more than 8 cores are both CPUs used.

Table 4.1 presents the results of a performance study by numbers of nodes for the default Intel compiler with Intel MPI. The table summarizes the observed wall clock time (total time to execute the code) in HH:MM:SS (hours:minutes:seconds) format. We consider 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 to over 4 billion equtions. For each mesh resolution, the parallel implementation of the test problem is run on increasing numbers of nodes from 1 to 64 by powers of 2 while varying the number of processes per node from 1 to 16 by powers of 2. The upper-left entry of each sub-table contains the runtime for the serial run of the code for that particular mesh. The lower-right entry of each sub-table 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. For the 1024×1024 mesh we observe a serial runtime of 9 seconds and a runtime of 1 second on all 1024 parallel processes. We observe the advantage of parallel computing for the 32768×32768 mesh where the serial run of about 120 hours can be reduced to approximately 25 minutes by using all 1024 parallel processes.

Reading along each row of the table, we observe that by doubling the number of nodes used, and thus also doubling the number of parallel process, we approximately halve the runtime. For instance, if we take 1 process per node on the 32768×32768 mesh, we observe that doubling the number of nodes from 1 node to 2 nodes results in an improvement in runtime from 120:33:23 to 56:10:03, an improvement by a factor of 2.15. This continues along the row with factors of improvement of 1.98, 2.00, 1.99, 1.98, 1.97 all the way to 1:49:42. This is a speedup of 65.94 from 1 node to 64 nodes. These speedups for using 1 process per node are slightly better than the optimal values,

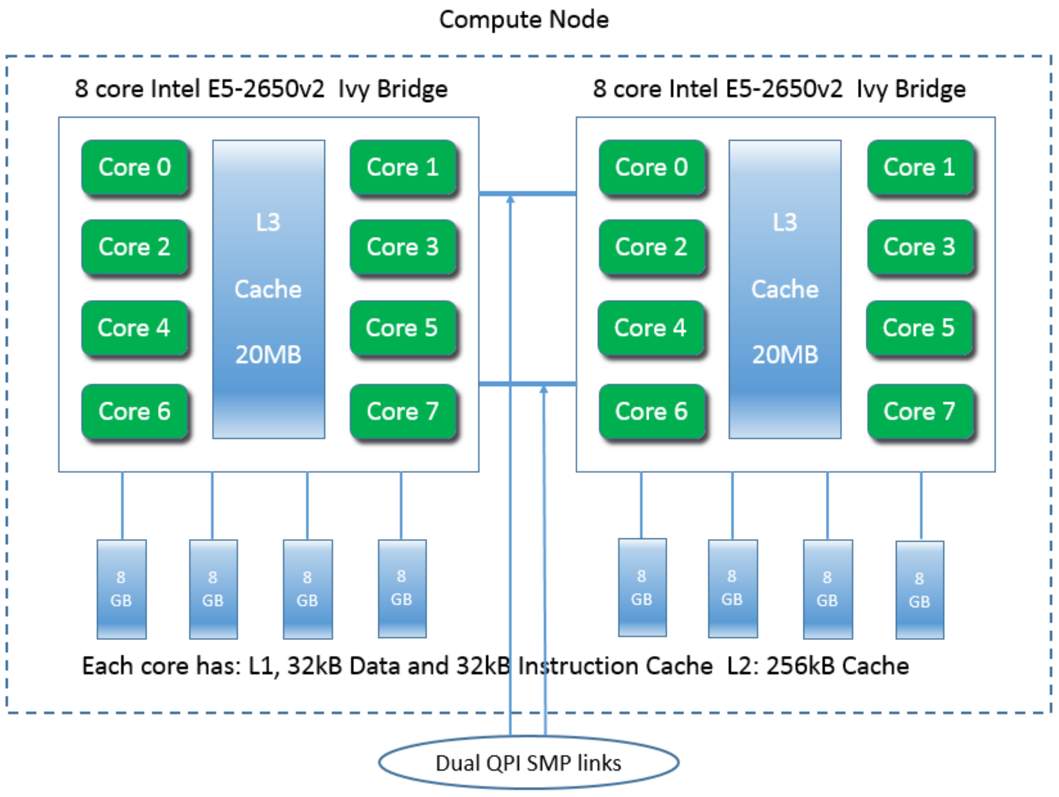


Figure 4.1: Schematic of a maya (2013) node.

which can be explained by considering the large memory requirement of this fine resolution: Code such as this is memory-bound, meaning that memory access is the limiting factor in performance. This becomes less of a problem if the same memory is distributed across more nodes and therefore speedup appears better than optimal. We can also consider the case of running all 16 processes per node. This results in an improvement from 22:56:40 to 11:31:18 or a factor of 1.99 from 1 node to 2 nodes. This behavior continues in the same fashion to 00:25:52 on 64 nodes. This is equivalent to a speedup of 53.22 from 1 node to 64 nodes using 16 processes per node. These speedups for using 16 processes per node are also excellent, since they are very close to their optimal values. The behavior observed for increasing the number of nodes confirms the quality of the high-performance InfiniBand interconnect.

In order to observe the effect of running different numbers of processes per node we read along each column of a sub-table. We observe that in most columns the runtime is approximately halved by doubling the processes per node from 1 to 2 and from 8 to 16. We also observe a significant speedup when doubling the processes per node from 2 to 4. However we observe only a small improvement in runtime by doubling the processes per node from 4 to 8. This is due to the assignment of processes to the cores of the two CPUs on each node. The default behavior on maya for jobs with up to 8 processes per node is to assign all of the processes to cores on only one of the CPUs. Since our code is memory-bound, we observe a bottleneck when 8 processes attempt to access the memory through only 4 memory channels. For most code which is not memory-bound, or if the 8 processes were distributed among the two CPUs, we would observe a halving of runtime also when doubling the processes per node from 4 to 8. In that case though, there would be only minimal speedup from 8 to 16 processes per node, which is the typical characteristic of memory-bound code.

Overall, we can conclude that the default behavior of code on maya is good: For fastest runs, using all 16 cores on a node has no disadvantage and using larger numbers of nodes shows optimal speedup up to the number of nodes available on maya (2013). At the same time, the default behavior of concentrating up to 8 processes on one CPU is appropriate for a multi-user system, since it allows additional users access to the second CPU.

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, while 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. We use here the conclusions drawn already above from Table 4.1 that there is no disadvantage to using all cores on a node, that is, for $p \geq 16$ we use 16 processes per node, while we use one node only for $p < 16$ (with the remaining cores idle). Comparing adjacent columns in the raw timing data in Table 4.2 (a) confirms our previous observation that, with the exception of increasing the number of increasing the number of processes from 4 to 8, using twice as many processes speeds up the code by a factor of two approximately, at least for small values of p. However the efficiency and speedup are generally poor for most values of p. To quantify this more clearly, the speedup in Table 4.2 (b) is computed, which shows near-optimal speedup with $S_p \approx p$. This is is expressed as $E_p \approx 1$ in Table 4.2 (c). This table confirms the dramatic loss efficiency that we observed in Table 4.1 which occurs by increasing the number of processes from 4 to 8.

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	
2 processes per node	00:00:03	00:00:02	00:00:01	00:00:01	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	
8 processes per node	00:00:01	00:00:01	00:00:00	00:00:00	00:00:00	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	

(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	
2 processes per node	00:00:47	00:00:19	00:00:07	00:00:04	00:00:02	00:00:01	
4 processes per node	00:00:28	00:00:11	00:00:04	00:00:02	00:00:02	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	
16 processes per node	00:00:20	00:00:07	00:00:02	00:00:02	00:00:02	00:00:02	

(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	
2 processes per node	00:06:18	00:03:14	00:01:36	00:00:41	00:00:15	00:00:09	
4 processes per node	00:03:30	00:01:50	00:00:57	00:00:25	00:00:09	00:00:05	
8 processes per node	00:02:44	00:01:24	00:00:44	00:00:20	00:00:06	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	

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

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

Table 4.2: Intel compiler with Intel MPI performance on maya (2013) by number of processes used with 16 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 $p = 8$ 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	00:00:09	00:00:03	00:00:02	00:00:01	00:00:01	00:00:00	00:00:00	00:00:01	00:00:01	00:00:01
2048	00:01:34	00:00:47	00:00:28	00:00:21	00:00:20	00:00:07	00:00:02	00:00:02	00:00:02	00:00:02
4096	00:12:26	00:06:18	00:03:30	00:02:44	00:02:44	00:01:25	00:00:45	00:00:17	00:00:06	00:00:12
8192	01:41:07	00:51:00	00:28:29	00:21:28	00:20:57	00:10:42	00:05:38	00:02:48	00:01:37	00:00:44
16384	14:08:26	07:01:30	03:55:38	02:55:26	02:49:13	01:25:16	00:43:08	00:22:10	00:11:33	00:06:31

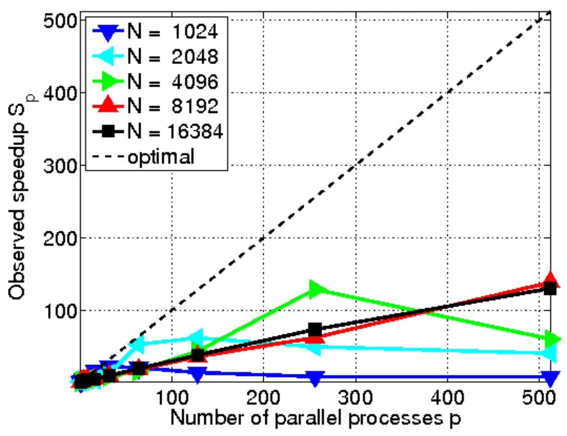
(b) Observed speedup S_p

N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$	$p = 256$	$p = 512$
1024	1.00	2.78	5.24	9.05	17.58	22.73	20.71	14.12	8.55	8.55
2048	1.00	2.02	3.41	4.57	4.79	12.68	52.91	61.96	49.83	40.77
4096	1.00	1.97	3.56	4.55	4.55	8.75	16.45	43.49	128.82	60.44
8192	1.00	1.98	3.55	4.71	4.83	9.45	17.96	36.06	62.40	138.59
16384	1.00	2.01	3.60	4.84	5.01	9.95	19.67	38.26	73.51	130.22

(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	1.00	1.39	1.31	1.13	1.10	0.71	0.32	0.11	0.03	0.02
2048	1.00	1.01	0.85	0.57	0.30	0.40	0.83	0.48	0.19	0.08
4096	1.00	0.99	0.89	0.57	0.28	0.27	0.26	0.34	0.50	0.12
8192	1.00	0.99	0.89	0.59	0.30	0.30	0.28	0.28	0.24	0.27
16384	1.00	1.01	0.90	0.60	0.31	0.31	0.31	0.30	0.29	0.25

(a) Observed speedup S_p



(b) Observed efficiency E_p

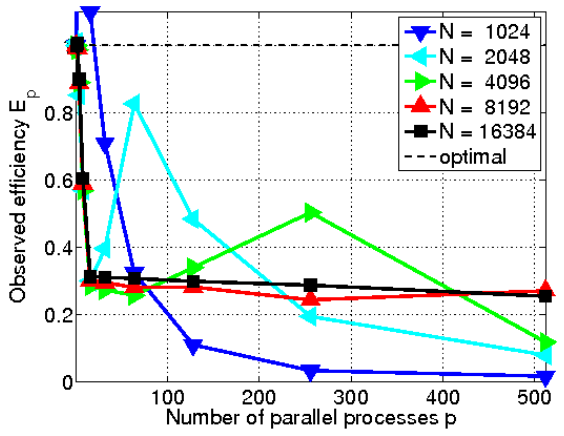


Figure 4.2: Intel compiler with Intel MPI performance on maya (2013) by number of processes used with 16 processes per node, except for $p = 1$ which uses 1 process per node, $p = 2$ which uses 2 processes per node, $p = 4$ which user 4 processes per node, and $p = 8$ which uses 8 processes per node.

5 Comparisons and Conclusions

Table 5.1 contains a summary of the results obtained on the cluster maya as well as a comparison to previous HPCF clusters. The table reports results for the historical mesh resolution of $N = 4096$, which was the largest resolution that could be solved on kali in 2003 (using the extended memory of 4 GB on the storage node). Also, to maintain backward comparisons, this table is restricted to 32 nodes, since the old clusters kali and hpc had fewer nodes. The first row of the table contains the results for cluster kali. This cluster was a 33-node distributed-memory cluster with 32 compute nodes including a storage node (with extended memory of 4 GB), containing the 0.5 TB central storage, each with two (single-core) Intel Xeon 2.0 GHz processors and 1 GB of memory, connected by a Myrinet interconnect, plus 1 combined user/management node. Note that for the case of all cores on 1 node, that is, for the case of both (single-core) CPUs used simultaneously, the performance was worse than for 1 CPU and hence the results were not recorded at the time. The second row of the table contains results for the cluster hpc which was a 35-node distributed-memory cluster with 33 compute nodes plus 1 development and 1 combined user/management node, each equipped with two dual-core AMD Opteron processors and at least 13 GB of memory, connected by a DDR InfiniBand network and with an Infiniband-accessible 14 TB parallel file system. The third row contains results for the cluster tara which was an 86-node distributed-memory cluster with two quad-core Intel Nehalem processors and 24 GB per node, a QDR InfiniBand interconnect, and 160 TB central storage. This cluster is now part of the cluster maya as maya (2009), and its QDR InfiniBand network extends to the newest portion maya (2013). The fourth row of the table contains results for the DDR InfiniBand connected portion maya (2010), and the fifth row contains results for the QDR InfiniBand connected portion maya (2013).

On the cluster kali, we observed a factor of approximately 30 speedup by increasing the number of nodes from 1 to 32. However by using both cores on each node we only see a factor of approximately 25 speedup. We do not observe the expected 64 factor speedup since each CPU on the node has such a low memory bandwidth that for a memory-bound algorithm it is actually faster to leave the second CPU idling rather than to use both [1]. Note that there are four cores on each node of cluster hpc compared to just two on the cluster kali, since the CPUs are dual-core. We observe approximately fourfold speedup that we would expect by running it on four cores rather than one. By running on 32 nodes with one core per node we observe the expected speedup of approximately 32; more in detail, the speedup is slightly better than optimal, which is explained by the smaller portions of the subdivided problem on each node fitting better into the cache of the processors. We see this for the first time here, but it is a typical effect in strong performance studies, in which a problem that already fits on one node is divided into smaller and smaller pieces as the number of nodes grows. Finally, by using all cores on 32 nodes we observe a speedup of 76.01, less than the optimal speedup of 128 [4]. On the cluster tara we observe a less than optimal speedup of approximately 5 by running on all 8 cores rather than on one, caused by the cores of a CPU competing for memory access. By running on 32 nodes with one core per node we observe a speedup of approximately 30. Finally, by using all 8 cores on 32 nodes we observe a speedup of 208, less than the optimal speedup of 256 [7]. On maya (2010) we observe that by running on all 8 cores on a single node rather than one core there is a speedup of approximately 3 rather than the optimal speedup of 8. By running on 32 nodes with one core per node we observe a speedup of approximately 30. But when combining the use of all cores with the use of 32 nodes, the DDR InfiniBand shows its limitation by reducing the speedup to 51, short of the optimal speedup of 256. On maya (2013) we observe that by running on all 16 cores on a single node rather than on one core there is a speedup of approximately 5 rather than the expected speedup of 16. We observe a greater than optimal speedup of 49.07 by running on 32 nodes with one process per node; this is caused by the relatively small problem fitting into cache after dividing it onto 32 nodes, together with the quality of the QDR InfiniBand interconnect. When combining the use of all 16 cores with the 32 nodes we observe a speedup of 62.17 rather than the optimal speedup of 512, indicating that even the QDR InfiniBand is becoming saturated.

Table 5.1 allows us to draw several key conclusions that affect the choice of scheduling rules on maya. The high-

Table 5.1: Runtimes (speedup) for $N = 4096$ on the clusters kali, hpc, tara, maya (2010), and maya (2013).

Cluster (year)	serial (1 core) time	1 node all cores time (speedup)	32 node 1 core per node time (speedup)	32 node all cores time (speedup)
kali (2003) [1]	02:00:49		00:04:05 (29.59)	00:04:49 (25.08)
hpc (2008) [4]	01:51:29	00:32:37 (3.42)	00:03:23 (32.95)	00:01:28 (76.01)
tara (2009) [7]	00:31:16	00:06:39 (4.70)	00:01:05 (28.86)	00:00:09 (208.44)
maya (2010)	00:17:00	00:05:48 (2.93)	00:00:34 (30.00)	00:00:20 (51.00)
maya (2013)	00:12:26	00:02:44 (4.55)	00:00:15 (49.07)	00:00:12 (62.17)

performance interconnect supports parallel scalability optimally, with the QDR InfiniBand outperforming the DDR InfiniBand in some cases. Even though speedup is less than optimal when using all cores in a node, it is certainly still faster to use all cores, as opposed to idling some, on the modern multi-core nodes. Finally, it is obvious and expected that the newer nodes are faster per core as well as per node, however, for most serial production code, that uses only 1 core, using one of the 2010 nodes with 2.8 GHz is a good default, and its DDR InfiniBand interconnect is no disadvantage for serial jobs.

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] 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.

[8] 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.

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