Parallel Performance Studies for an Elliptic Test Problem

Matthias K. Gobbert

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

gobbert@math.umbc.edu

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 the scheduling policy used. 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, 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 the cluster hpc in the UMBC High Performance Computing Facility and give guidance on the scheduling policy to be implemented. Specifically, they confirm that it is beneficial to use all four cores of the two dual-core processors on each node simultaneously, giving us in effect a computer that can run jobs efficiently with up to 128 parallel processes.

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 by the finite difference method results in a large, sparse, highly structured system of linear equations. The parallel implementation of the conjugate gradient method as appropriate iterative linear solver for this linear system involves necessarily communications both between all participating parallel processes and between pairs of processes in every iteration. Therefore, this method provides an excellent test problem for the overall, real-life performance of a parallel computer. The results are not just applicable to the conjugate gradient method, which is important in its own right as a representative of the class of Krylov subspace methods, but to all memory 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] 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. These past results for compute nodes with two (single-core) processors purchased in 2003 from IBM also show clearly that for best speedup over serial code only one parallel process should be run per node; in fact, using both processes of each node may not make the runs any faster. This is a well-known fact for memory bound code, that is, code that requires a lot of data to feed the processor, and gives rise to the observation there will not be any performance improvement when using both processors, see for instance the webpage of the state-of-the-art package PETSc [5], which implements methods of the same type as considered here. This note now considers these issues for the new distributed-memory cluster hpc purchased in 2008 also from IBM with InfiniBand interconnect and with each compute node having two dual-core processors (AMD Opteron 2.6 GHz with 1024 kB cache and 13 GB of memory per node) for a total of up to four parallel processes to be run simultaneously per node. This machine is part of the UMBC High Performance Computing Facility, see `www.umbc.edu/hpcf` for more information on the facility as well as the machine. Section 4 describes the parallel scalability results in detail and provides the underlying data for the following summary results.

Table 1 summarizes the key results of the present study by giving the wall clock time (total time to execute the code) in hours:minutes:seconds (HH:MM:SS) format. We consider the test problem on four progressively finer meshes, resulting in progressively larger systems of linear equations with system dimensions ranging from about 1 million up to 67 million equations. The parallel implementation of the conjugate gradient method is run on different numbers of nodes from 1 to 32 with different numbers of processes per node used. 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 76 seconds (1:16 minute) for the 1024×1024 mesh resulting in about 1 million equations. The lower-right entry of each sub-table lists the run using both cores of both dual-core processors on all 32 nodes for a total of 128 parallel processes working together to solve the problem, which takes about 3 seconds for this mesh. More

Table 1: Wall clock time in HH:MM:SS for the solution of problems on $N \times N$ meshes using 1, 2, 4, 8, 16, 32 compute nodes with 1, 2, and 4 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
1 process per node	00:01:16	00:00:45	00:00:21	00:00:10	00:00:05	00:00:02
2 processes per node	00:00:56	00:00:30	00:00:14	00:00:07	00:00:03	00:00:01
4 processes per node	00:00:27	00:00:14	00:00:07	00:00:03	00:00:02	00:00:03
(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
1 process per node	00:10:20	00:05:18	00:02:38	00:01:28	00:00:41	00:00:21
2 processes per node	00:07:02	00:04:01	00:02:06	00:00:59	00:00:32	00:00:15
4 processes per node	00:03:30	00:01:56	00:01:00	00:00:31	00:00:18	00:00:08
(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
1 process per node	01:51:29	00:52:04	00:27:51	00:13:07	00:06:41	00:03:23
2 processes per node	01:10:07	00:35:52	00:17:58	00:09:25	00:04:33	00:02:22
4 processes per node	00:32:37	00:16:45	00:08:55	00:04:33	00:02:32	00:01:28
(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
1 process per node	15:29:02	07:57:23	03:36:56	01:48:33	00:53:50	00:27:13
2 processes per node	09:55:05	04:30:34	02:26:41	01:12:59	00:37:11	00:18:15
4 processes per node	04:05:11	02:17:12	01:08:28	00:35:25	00:18:26	00:10:02

strikingly, one realizes the advantage of parallel computing for the large 8192×8192 mesh with over 67 million equations: The serial run of about $15\frac{1}{2}$ hours can be reduced to about 10 minutes using 128 parallel processes.

The results in Table 1 are arranged to study two key questions: (i) whether the code scales optimally to all 32 nodes, which ascertains the quality of the InfiniBand interconnect network, and (ii) whether it is worth wile 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 or not).

(i) Reading along each row of Table 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 nodes in the final columns of table in all but the coarsest mesh considered.

(ii) To analyze the effect of running 1, 2, or 4 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 dual-processors nodes. These results confirm that it is not just effective to use both processors on each node, but also to use both cores of each dual-core processor simultaneously. Roughly, this shows that the architecture of the IBM nodes purchased in 2008 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 dual-core processors (as opposed to single-core processors). Moreover, these results will guide the scheduling policy implemented on the cluster: Namely, on the one hand, it is not disadvantages for the serial jobs to run several of them simultaneously on one node, and on the other hand, for jobs using several nodes, it is advantageous to make use of all cores on the nodes reserved by the scheduler.

To put the results of Table 1 in perspective, contrast them to those in Table 7 in the appendix for the cluster kali purchased in 2003 with a Myrinet interconnect network and two (single-core) processors per node: The speedup along rows of the table is excellent, as well, which confirms the low latency of the Myrinet interconnect. But comparing the data column-wise in each sub-table shows that it is of no advantage to use both processors on each node, as the architecture of each node cannot deal with the contention of both processors attempting to access the shared memory, thus creating a bottleneck for memory bound codes.

2 The Elliptic Test Problem

We consider the classical elliptic test problem of the Poisson equation with homogeneous Dirichlet boundary conditions (see, e.g., [6, 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 pieces as parallel processes are available and a portion distributed to each process. That is, each parallel process possesses its own portion of each vector, and normally no vector is ever assembled in full on any process. To understand what this means for parallel programming and the performance of the method, note that an inner product between two vectors distributed in this way is computed by first forming the local inner product between the local portions 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 portions, because they do not require any parallel communications. However, this assumes tacitly that the scalar factors that appear in the vector updates are available on all parallel processes. This is accomplished already as part of the computation of these factors by using a so-called Allreduce operation, that is, a reduce operation that also communicates the result to all processes. This is implemented in the MPI function `MPI_Allreduce`. Finally, the matrix-vector product $q = Ap$ also computes only the portion of the vector q that is local to each process. But since the matrix A has non-zero off-diagonal elements, each local portion needs values of p that are local to the two processes that hold the neighboring portions of p. The communications between parallel processes thus needed are so-called point-to-point communications, because not all processes participate in each of them, but rather only specific pairs of processes that exchange data needed for their local calculations. Observe now that it is only a few components of q that require data from p that is not local to the process. Therefore, it is possible and potentially very efficient to proceed to calculate those components that can be computed from local data only, while the communication with the other processes is taking place. This technique is known as interleaving calculations and communications and can be implemented using the non-blocking MPI communications commands `MPI_Isend` and `MPI_Irecv`.

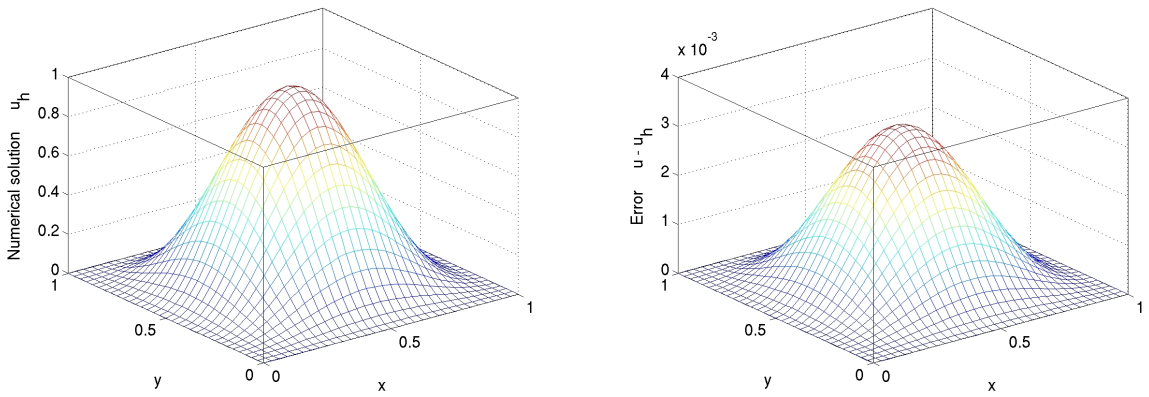
3 Convergence Study for the Model Problem

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 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 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 2 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 wall clock time in HH:MM:SS and in seconds, and the predicted and observed memory usage in MB



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

Figure 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 2: Convergence study listing the mesh resolution N, the number of degrees of freedom (DOF), the norm of the finite difference error $\|u - u_h\|_{L^\infty(\Omega)}$, the number of conjugate gradient iterations to convergence, the time in HH:MM:SS and in seconds, and the predicted and observed memory usage in MB for a one-processor 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	00:00:00	0.02	< 1	N/A
64	4,096	8.0356e–4	95	00:00:00	0.03	< 1	N/A
128	16,384	2.0081e–4	191	00:00:00	0.11	< 1	N/A
256	65,536	5.0191e–5	385	00:00:01	1.07	2	N/A
512	262,144	1.2543e–5	781	00:00:09	8.78	8	13
1024	1,048,576	3.1327e–6	1579	00:01:16	75.57	32	37
2048	4,194,304	7.8097e–7	3191	00:10:20	620.13	128	133
4096	16,777,216	1.9356e–7	6447	01:51:29	6688.58	512	517
8192	67,108,864	4.6817e–8	13028	15:29:02	55742.27	2048	2179

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 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 by monitoring the Linux command `top` on the compute node being used. For runs that take under one second, it was not possible to observe the memory usage in this way and this is indicated by N/A in the last column.

The norms of the finite difference errors in Table 2 decrease by a factor of 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 theory. This also confirms that the tolerance of the iterative linear solver is tight enough to ensure a sufficiently accurate solution of the linear system also for the finest meshes considered. The increasing numbers of iterations needed to achieve the convergence of the linear solver highlights the fundamental computational challenge with methods of this type: 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 finer meshes do show some significant run times (over one hour) and memory usage (more than 1 GB); while not overwhelming by today's standards, parallel computing clearly does offer opportunities here 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 2 agree with past results for this problem, repeated for convenience in Table 6 in the appendix as well as compared to results obtained by the software package MATLAB; 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 hpc

The run times for the finer meshes observed for serial runs in Table 2 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. would be too long infeasible in practice. 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 the number 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 3 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 3 distribute these processes as widely as possible over the available nodes, that is, each process is run on a different node up to the available number of 32 nodes. In other words, up to $p = 32$, three of the four cores available on each node are idling, and only one core performs calculations. For $p = 64$ and $p = 128$, this cannot be accommodated on 32 nodes, thus 2 processes run on each node for $p = 64$ and 4 processes per node for $p = 128$. Comparing adjacent columns in the raw timing data in Table 3 (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 3 (b) is computed, which shows near-optimal speedup with $S_p \approx p$ for all cases up to $p = 32$, which is expressed in terms of efficiency $E_p \approx 1$ in Table 3 (c). The customary visualizations of speedup and efficiency are presented in Figure 2 (a) and (b), respectively. Figure 2 (a) shows very clearly the excellent speedup up to $p = 32$ parallel processes. The efficiency plotted in Figure 2 (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 the precise data in Table 3 (b) and (c) available and not just their graphical representation in Figure 3. The conclusions discussed so far apply to up to $p = 32$ parallel processes. In each case, only 1 parallel process is run on each node, with the other three cores available to handle all other operating system or other duties. For $p = 64$ and $p = 128$, 2 or 4 processes share each node necessarily, as only 32 nodes are available, thus one expects slightly degraded performance as we go from $p = 32$ to $p = 64$ and $p = 128$. This is born out by all data in Table 3 as well as clearly visible in Figure 3 for $p > 32$. However, the times in Table 3 (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 and Figure 4, and we use 4 processes per node in Table 5 and Figure 5, wherever possible. That is, $p = 128$ requires 4 processes per node also in Table 4 and Figure 4, as only 32 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 5 and Figure 5 runs only this two-process job on its node. The results in the efficiency plots of Figures 4 (b) and 5 (b) show clearly that there is a significant loss of efficiency when going from $p = 1$ (always on a dedicated node) to $p = 2$ (with both processes on one node). But the efficiency plots also show clearly that the performance continues to improve with larger p all the way up to $p = 128$ without any additional loss of efficiency. The detailed timing data in Tables 4 (a) and 5 (b) confirm this, as the timings continue to halve for doubling the number of processes, except from $p = 1$ to 2. In fact, it is remarkable in Table 5 that even using all four cores per node still gives excellent improvement of performance. This leads us to conclude that there is no disadvantage to using all cores per node, as opposed to 'reserving' one core to handle the operating system tasks or similar.

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, or 4 processes per node. This is answered by the data organized in the form of Table 1 in the Introduction. It is these results, which are the same raw timing data as in Tables 3, 4, and 5, that make it clear that using 4 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 2.

Table 3: Performance by number of processes used with 1 process per node, except for $p = 64$ which uses 2 processes per node and $p = 128$ which uses 4 processes per node.

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

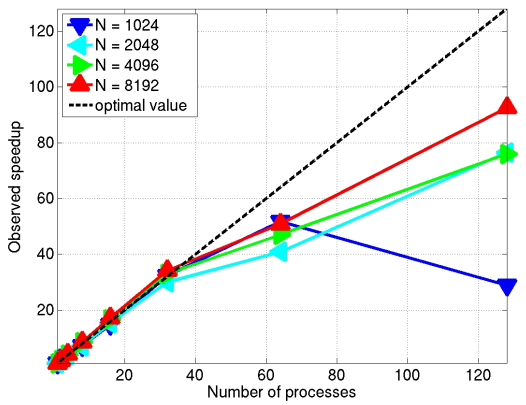
N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$
1024	00:01:16	00:00:45	00:00:21	00:00:10	00:00:05	00:00:02	00:00:01	00:00:03
2048	00:10:20	00:05:18	00:02:38	00:01:28	00:00:41	00:00:21	00:00:15	00:00:08
4096	01:51:29	00:52:04	00:27:51	00:13:07	00:06:41	00:03:23	00:02:22	00:01:28
8192	15:29:02	07:57:23	03:36:56	01:48:33	00:53:50	00:27:13	00:18:15	00:10:02

(b) Observed speedup S_p

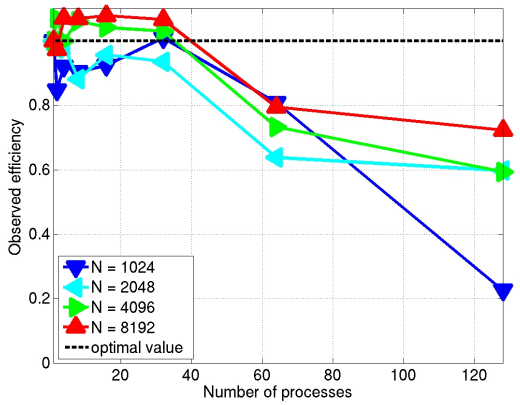
N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$
1024	1.0000	1.6944	3.6845	7.2316	14.7886	32.2949	51.7603	28.8435
2048	1.0000	1.9514	3.9346	7.0549	15.3119	29.9580	40.8518	76.4649
4096	1.0000	2.1412	4.0031	8.4985	16.6743	32.9763	46.9506	75.9031
8192	1.0000	1.9461	4.2825	8.5592	17.2577	34.1263	50.8941	92.5582

(c) Observed efficiency E_p

N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$
1024	1.0000	0.8472	0.9211	0.9039	0.9243	1.0092	0.8088	0.2253
2048	1.0000	0.9757	0.9836	0.8819	0.9570	0.9362	0.6383	0.5974
4096	1.0000	1.0706	1.0008	1.0623	1.0421	1.0305	0.7336	0.5930
8192	1.0000	0.9730	1.0706	1.0699	1.0786	1.0664	0.7952	0.7231



(a) Observed speedup S_p



(b) Observed efficiency E_p

Figure 2: Performance by number of processes used with 1 process per node, except for $p = 64$ which uses 2 processes per node and $p = 128$ which uses 4 processes per node.

Table 4: Performance by number of processes used with 2 processes per node, except for $p = 1$ which uses 1 process per node and $p = 128$ which uses 4 processes per node.

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

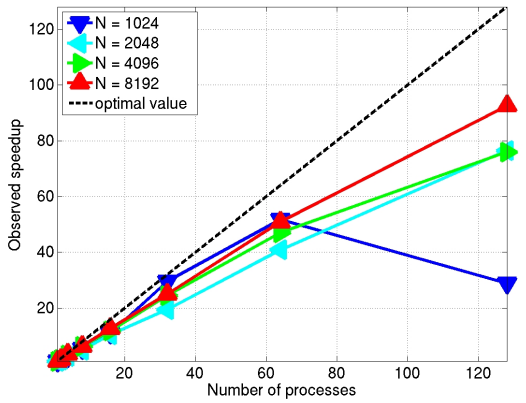
N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$
1024	00:01:16	00:00:56	00:00:30	00:00:14	00:00:07	00:00:03	00:00:01	00:00:03
2048	00:10:20	00:07:02	00:04:01	00:02:06	00:00:59	00:00:32	00:00:15	00:00:08
4096	01:51:29	01:10:07	00:35:52	00:17:58	00:09:25	00:04:33	00:02:22	00:01:28
8192	15:29:02	09:55:05	04:30:34	02:26:41	01:12:59	00:37:11	00:18:15	00:10:02

(b) Observed speedup S_p

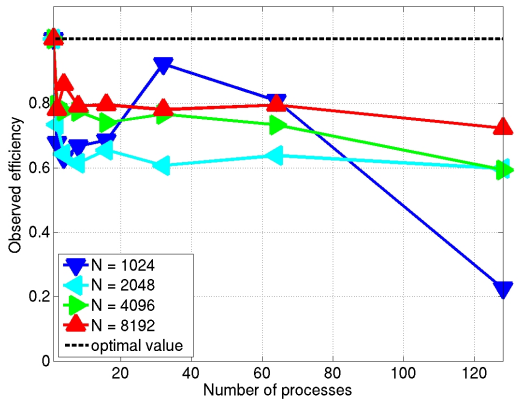
N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$
1024	1.0000	1.3565	2.5249	5.3406	10.9522	29.5195	51.7603	28.8435
2048	1.0000	1.4692	2.5741	4.9197	10.4876	19.4276	40.8518	76.4649
4096	1.0000	1.5897	3.1079	6.2063	11.8474	24.5389	46.9506	75.9031
8192	1.0000	1.5612	3.4337	6.3339	12.7281	24.9881	50.8941	92.5582

(c) Observed efficiency E_p

N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$
1024	1.0000	0.6782	0.6312	0.6676	0.6845	0.9225	0.8088	0.2253
2048	1.0000	0.7346	0.6435	0.6150	0.6555	0.6071	0.6383	0.5974
4096	1.0000	0.7949	0.7770	0.7758	0.7405	0.7668	0.7336	0.5930
8192	1.0000	0.7806	0.8584	0.7917	0.7955	0.7809	0.7952	0.7231



(a) Observed speedup S_p



(b) Observed efficiency E_p

Figure 3: Performance by number of processes used with 2 processes per node, except for $p = 1$ which uses 1 process per node and $p = 128$ which uses 4 processes per node.

Table 5: Performance by number of processes used with 4 processes per node, except for $p = 1$ which uses 1 process per node and $p = 2$ which uses 2 processes per node.

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

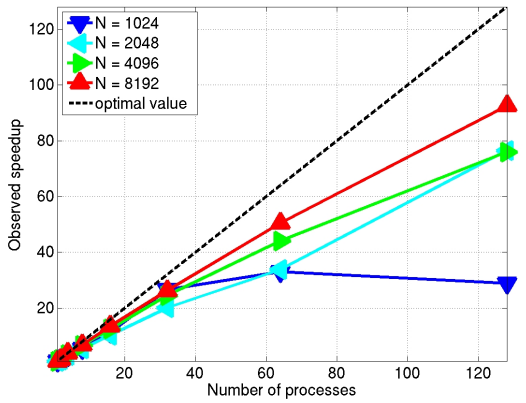
N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$
1024	00:01:16	00:00:56	00:00:27	00:00:14	00:00:07	00:00:03	00:00:02	00:00:03
2048	00:10:20	00:07:02	00:03:30	00:01:56	00:01:00	00:00:31	00:00:18	00:00:08
4096	01:51:29	01:10:07	00:32:37	00:16:45	00:08:55	00:04:33	00:02:32	00:01:28
8192	15:29:02	09:55:05	04:05:11	02:17:12	01:08:28	00:35:25	00:18:26	00:10:02

(b) Observed speedup S_p

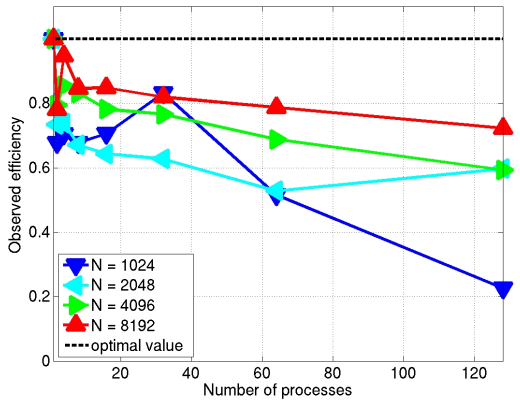
N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$
1024	1.0000	1.3565	2.8187	5.4250	11.2960	26.6092	33.0000	28.8435
2048	1.0000	1.4692	2.9565	5.3682	10.2909	20.0754	33.8314	76.4649
4096	1.0000	1.5897	3.4174	6.6576	12.5123	24.5380	44.0560	75.9031
8192	1.0000	1.5612	3.7892	6.7716	13.5682	26.2266	50.4222	92.5582

(c) Observed efficiency E_p

N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$
1024	1.0000	0.6782	0.7047	0.6781	0.7060	0.8315	0.5156	0.2253
2048	1.0000	0.7346	0.7391	0.6710	0.6432	0.6274	0.5286	0.5974
4096	1.0000	0.7949	0.8544	0.8322	0.7820	0.7668	0.6884	0.5930
8192	1.0000	0.7806	0.9473	0.8465	0.8480	0.8196	0.7878	0.7231



(a) Observed speedup S_p



(b) Observed efficiency E_p

Figure 4: Performance by number of processes used with 4 processes per node, except for $p = 1$ which uses 1 process per node and $p = 2$ which uses 2 processes per node.

A Performance Studies on kali

This appendix summarizes results of analogous studies to the previous sections performed on the cluster kali purchased in 2003. This cluster had originally 32 nodes, each with two (single-core) processors (Intel Xeon 2.0 GHz with 512 kB cache) and 1 GB of memory, connected by a Myrinet interconnect network; only 27 of the 32 nodes are connected by the Myrinet network at present (2008), hence only 16 nodes are available for parallel performance study, when considering only powers of 2 for convenience.

Table 6 collects the results of a convergence study for the finite difference approximation to the partial differential equation as well as for the conjugate gradient method as iterative solver of the resulting system of linear equations. The convergence results confirm the correctness of the analogous studies in Table 2. Comparing the serial raw timings, we see that the new processors (using only one core) are roughly 20% faster than the old processors.

Table 7 is a summary table of raw timing results analogous to Table 1 in the Introduction. Reading the data row-wise, we observe again excellent speedup, which confirms that the Myrinet network works very well. However, comparing the data for each mesh resolution column-wise, we notice clearly that using both processors per node is not an advantage over only using one processors with the second one idling. This is an observation that used to be accepted fact and is the basis for the statement that "standard dual processor PC's will not provide better performance when the second processor is used" [5, FAQ "What kind of parallel computers or clusters are needed to use PETSc?"]. Notice from Table 6 that the problem with $N = 2048$ requires over 2 GB of memory. Therefore, it is not possible to run this job on 2 nodes, as both nodes together only have 2 GB of memory. By contrast, the cases of 1 node are actually run on the storage node of kali, which has 4 GB of memory and which can thus accommodate the problem, either with 1 process or with 2 processes splitting the job.

Table 8 and Figure 5 summarize and visualize the underlying performance results for the case of running only 1 process on each node up to the available $p = 16$ nodes, analogous to Table 3 and Figure 2. We observe again excellent speedup up to the point of $p = 32$, where 2 processes necessarily share each node on the 16 available nodes. Notice how the drop-off in performance more dramatic for all cases of N from $p = 16$ to $p = 32$ here, more so than from $p = 32$ to 64 in Figure 2.

Table 9 and Figure 6 summarize and visualize the performance results for the case of running 2 processes on each node, except $p = 1$ with 1 process. In particular the efficiency plot in Figure 6 (b) very clearly demonstrate that the performance degradation occurs from $p = 1$ to $p = 2$, that is, it is associated with using both processors per node instead of one process only.

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 Reseach 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] Satish Balay, Kris Buschelman, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. Portable, extensible toolkit for scientific computation (PETSc). `www.mcs.anl.gov/petsc`. Version 2.3.3, released May 23, 2007.

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

Table 6: Convergence study on kali 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 time in HH:MM:SS and in seconds, and the predicted and observed memory usage in MB for a one-processor 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	00:00:00	0.01	< 1	N/A
64	4,096	8.0356e–4	95	00:00:00	0.02	< 1	N/A
128	16,384	2.0081e–4	191	00:00:00	0.13	< 1	N/A
256	65,536	5.0191e–5	385	00:00:02	1.89	2	N/A
512	262,144	1.2543e–5	781	00:00:15	14.69	8	10
1024	1,048,576	3.1327e–6	1579	00:02:04	123.79	32	35
2048	4,194,304	7.8098e–7	3191	00:15:39	939.34	128	129
4096	16,777,216	1.9371e–7	6447	02:15:44	8144.13	512	514
8192	67,108,864	4.7355e–8	13028	19:46:16	71176.47	2048	2051

Table 7: Performance on kali. Wall clock time in HH:MM:SS for the solution of problems on $N \times N$ meshes using 1, 2, 4, 8, 16 compute nodes with 1 and 2 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
1 process per node	00:01:56	00:01:11	00:00:31	00:00:16	00:00:09
2 processes per node	00:01:56	00:01:07	00:00:29	00:00:19	00:00:07
(b) Mesh resolution $N \times N = 2048 \times 2048$, system dimension 4,194,304					
	1 node	2 nodes	4 nodes	8 nodes	16 nodes
1 process per node	00:15:43	00:08:07	00:04:03	00:02:02	00:01:05
2 processes per node	00:15:38	00:08:16	00:04:27	00:02:31	00:01:17
(c) Mesh resolution $N \times N = 4096 \times 4096$, system dimension 16,777,216					
	1 node	2 nodes	4 nodes	8 nodes	16 nodes
1 process per node	02:15:58	01:07:28	00:33:55	00:16:59	00:08:47
2 processes per node	02:05:20	01:04:18	00:32:42	00:18:21	00:09:44
(d) Mesh resolution $N \times N = 8192 \times 8192$, system dimension 67,108,864					
	1 node	2 nodes	4 nodes	8 nodes	16 nodes
1 process per node	19:46:16	N/A	04:50:22	02:24:10	01:13:58
2 processes per node	19:06:55	N/A	04:18:20	02:12:10	01:18:31

Table 8: Performance on kali by number of processes used with 1 process per node, except for $p = 32$ which uses 2 processes per node.

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

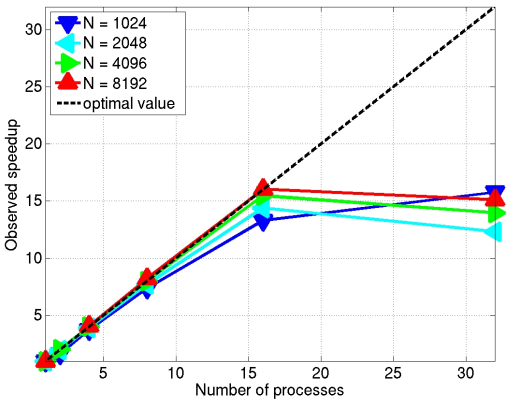
N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$
1024	00:01:56	00:01:11	00:00:31	00:00:16	00:00:09	00:00:07
2048	00:15:43	00:08:07	00:04:03	00:02:02	00:01:05	00:01:17
4096	02:15:58	01:07:28	00:33:55	00:16:59	00:08:47	00:09:44
8192	19:46:16	N/A	04:50:22	02:24:10	01:13:58	01:18:31

(b) Observed speedup S_p

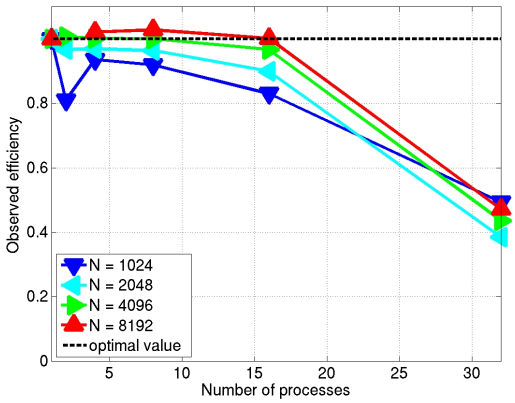
N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$
1024	1.0000	1.6202	3.7447	7.3537	13.2874	15.7708
2048	1.0000	1.9336	3.8792	7.7109	14.3927	12.3165
4096	1.0000	2.0154	4.0096	8.0054	15.4753	13.9698
8192	1.0000	N/A	4.0854	8.2284	16.0377	15.1087

(c) Observed efficiency E_p

N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$
1024	1.0000	0.8101	0.9362	0.9192	0.8305	0.4928
2048	1.0000	0.9668	0.9698	0.9639	0.8995	0.3849
4096	1.0000	1.0077	1.0024	1.0007	0.9672	0.4366
8192	1.0000	N/A	1.0213	1.0285	1.0024	0.4721



(a) Observed speedup S_p



(b) Observed efficiency E_p

Figure 5: Performance on kali by number of processes used with 1 process per node, except for $p = 32$ which uses 2 processes per node.

Table 9: Performance on kali by number of processes used with 2 processes per node, except for $p = 1$ which uses 1 process per node.

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

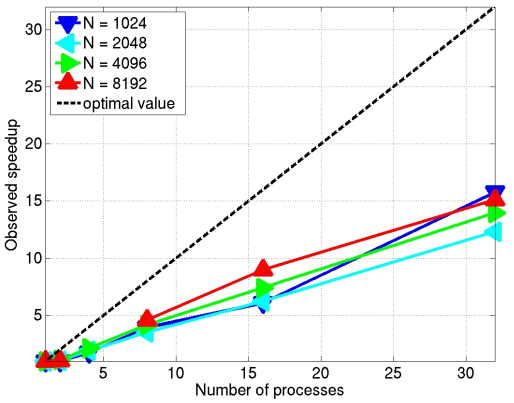
N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$
1024	00:01:56	00:01:56	00:01:07	00:00:29	00:00:19	00:00:07
2048	00:15:43	00:15:38	00:08:16	00:04:27	00:02:31	00:01:17
4096	02:15:58	02:05:20	01:04:18	00:32:42	00:18:21	00:09:44
8192	19:46:16	19:06:55	N/A	04:18:20	02:12:10	01:18:31

(b) Observed speedup S_p

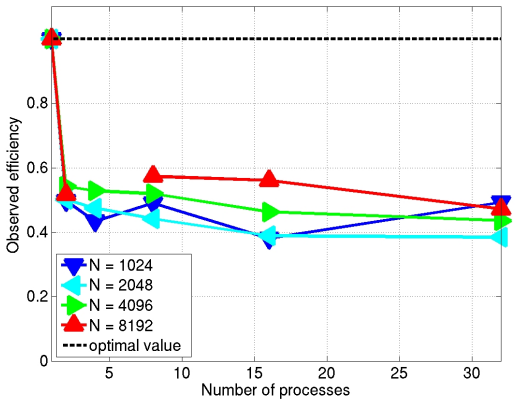
N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$
1024	1.0000	0.9963	1.7350	3.9226	6.0938	15.7708
2048	1.0000	1.0054	1.8989	3.5345	6.2241	12.3165
4096	1.0000	1.0848	2.1145	4.1588	7.4125	13.9698
8192	1.0000	1.0343	N/A	4.5919	8.9758	15.1087

(c) Observed efficiency E_p

N	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$
1024	1.0000	0.4981	0.4337	0.4903	0.3809	0.4928
2048	1.0000	0.5027	0.4747	0.4418	0.3890	0.3849
4096	1.0000	0.5424	0.5286	0.5198	0.4633	0.4366
8192	1.0000	0.5172	N/A	0.5740	0.5610	0.4721



(a) Observed speedup S_p



(b) Observed efficiency E_p

Figure 6: Performance on kali by number of processes used with 2 processes per node, except for $p = 1$ which uses 1 process per node.