Configuration and Performance of a Beowulf Cluster for Large-Scale Scientific Simulations

To achieve optimal performance on a Beowulf cluster for large-scale scientific simulations, it's necessary to combine the right numerical method with its efficient implementation to exploit the cluster's critical high-performance components. This process is demonstrated using a simple but prototypical problem of solving a time-dependent partial differential equation.

Beowulf clusters in virtually every price range are readily available today for purchase in fully integrated form from a large variety of vendors. At the University of Maryland, Baltimore County (UMBC), my colleagues and I bought a medium-sized 64-processor cluster with high-performance interconnect and extended disk storage from IBM. The cluster has several critical components, and I demonstrate their roles using a prototype problem from the numerical solution of time-dependent partial differential equations (PDEs). I selected this problem to show how judiciously combining a numerical algorithm and its efficient implementation with the right hardware (in this case, the Beowulf cluster) can achieve parallel computing's two fundamental goals: to solve problems faster and to solve larger problems than we can on a serial computer.

System Configuration

Our cluster—called Kali after the multiarmed Indian mother goddess—is an IBM 1350 xSeries cluster with 32 dual-processor nodes, a high-performance Myrinet interconnect for parallel computations, and a 0.5-Tbyte central disk array (www.ibm.com/servers/eserver/clusters/). We run it with a version of the Linux operating system. We use only one possible commodity cluster configuration and describe its use in one type of application area; however, much more powerful systems share the same conceptual setup. You can find more information about Kali at www.math.umbc.edu/~gobbert/kali/.

All CPUs are Intel Xeon 2.0-GHz processors. Beowulf clusters became affordable in recent years because these CPUs are commodity products, mass-produced for the PC market. To obtain the best performance from their integration into a parallel computer, though, many other vital components must be specialized and are by no means "commodity" components (they're not cheap, either).

One example of vital specialized hardware is the Myrinet interconnect from Myricom (www.myricom.com). Its key features are much-reduced latency (time delay for a communication to start) compared to conventional Ethernet and a high volume of throughput, rated at 2 Gbits per second (Gbps). Our system includes a 32-port Myrinet switch composed of four blades with eight ports each. The ports on each blade and the four blades themselves are connected via crossbar links.

We use the Myrinet interconnect for communi-

1521-9615/05/$20.00 © 2005 IEEE
Copublished by the IEEE CS and the AIP

Matthias K. Gobbert
University of Maryland, Baltimore County

cations in a parallel code. All other network traffic—for example, file serving—travels over a secondary network of conventional 100-Mbits-per-second (Mbps) Ethernet. Figure 1 shows a schematic of the cluster nodes. The 32 dual-processor computational nodes available for parallel computations (shown inside the red dashed box) are labeled `node001`, `node002`, …, `node031`, and `storage1`. The secondary Ethernet network is shown in blue and purple and connects all the nodes. Although all computational nodes have two Intel CPUs, `storage1` is a special *storage node* because it connects to the main 0.5-Tbyte disk array. This node efficiently serves files to all other nodes using a special, faster 1-Gbps port (shown in purple in the figure) in the Ethernet switch. Additionally, reflecting the storage node's importance to the cluster operation, it has 4 Gbytes of memory, two 36.4-Gbyte hard drives (the second one mirroring the first for fail-safe redundancy), and two hot-swappable power supplies. The other 31 *compute nodes* proper have 1 Gbyte of memory and an 18.2-Gbyte local hard drive. This cluster constitutes a *distributed-memory* parallel computer—each node's memory can be accessed only by the CPUs on that node.

In principle, considerable disk space is clearly available across the compute nodes—more than 32 times 18.2 Gbytes, or 582.4 Gbytes, although parts of those disks contain operating system utilities, software locally installed on the nodes for better performance, and network and management utilities. Central disk storage—in the form of the 0.5-Tbyte redundant array of independent disks (RAID) connected to the storage node—allows more space for either applications driven by large input data or those with large output files, and provides user convenience by allowing offline post-processing in serial on the user node. This RAID comprises eight 73.4-Gbyte Small Computer System Interface (SCSI) hard disks, one of which is a hot-swappable spare. The files are stored in striped form—that is, each file is split into pieces that are distributed across the hard drives, speeding up disk access significantly by allowing reads to and writes from all drives to be performed in parallel. The use of checkbits, which allow for data recovery in case one of the seven remaining disks fails, increases the disk space required to store a file and thus reduces the capacity of the RAID actually available by the equivalent of roughly another hard drive. Thus, we actually have about 367 Gbytes of usable disk space on the RAID, after we also deduct a spool partition for the tape backup system. This setup is designed for a high degree of data safety—up to two hard drives can fail before you have to resort to the tape backup system. Although physically in its own case, the RAID is directly connected to the storage node's bus via a SCSI controller (shown in brown in Figure 1). Another vital benefit of a central storage device is that it lets us effectively manage tape backups. To accommodate the standard tape capacity of 100 Mbytes, the RAID's space is subdivided into a 100-Mbyte `/home` partition, which is backed up, and a 267-Mbyte `/scratch` partition, which is not.

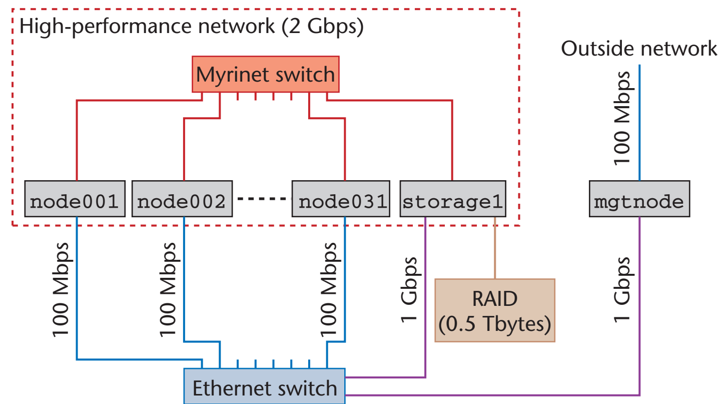


Figure 1. Schematic of the network connections and their connection speeds. The high-performance computational network is shown in red. The secondary Ethernet network, used to serve files from the RAID, is shown in blue and purple. The management network isn't shown.

The *management and user node*—`mgtnode` in Figure 1—is a dual-processor node with the same-sized memory and fail-safe dual hard drive and power supply setup as the storage node. However, the management and user node has a connection to the outside network and is connected to all other nodes only via the Ethernet switch. Like the storage node, `mgtnode` uses a 1-Gbps connection (shown in purple in Figure 1) to connect to the other nodes. In addition to performing system management, such as booting the entire system, `mgtnode` doubles as Kali's user node. Users edit and compile their programs, submit them to a scheduler, and postprocess results graphically, accessing the control RAID via the special, faster ports in the Ethernet switch, outside of the computational network.

Reviewing the network design, the Myrinet interconnect carries the interprocessor communications within the parallel code, whereas the Ethernet switch lets any node connect directly to the RAID, including the management node. So, when a job runs in parallel on several computational

Alternative Setups

We did have alternative choices for the cluster's setup. On one hand, we could have dispensed with the `mgtnode` outside of the Myrinet universe and instead integrated its functions into a combined storage, management, and user node. This would let us access the redundant array of independent disks (RAID) directly from the user node. However, because this node is often busy with multiple users compiling code or postprocessing data, we'd usually need to exclude it from parallel computations, effectively reducing the cluster size to 31 computational nodes. Thus, it's advantageous to dedicate an additional node to the management and user tasks, giving us a true 64-processor cluster. Extending the Myrinet network to this node would be very costly, because we'd need a larger, 64-port Myrinet switch.

On the other hand, we could have connected the RAID to the `mgtnode` to give the user node direct access to the user data for postprocessing. This wouldn't change the network usage. However, it would have robbed us of one additional option we wanted for future research on code performance, because a second way exists in our setup for transferring data from a compute node to the RAID: We can dedicate the process running on `storage1` to be a so-called I/O process by making it the only process among one job's parallel processes that accesses the (directly connected) RAID and that communicates the data to and from the other processes working on the job via the Myrinet. This might conceivably lead to significant performance improvements for applications with large I/O. However, this approach is more cumbersome to program compared to making disk access available directly via the Ethernet, and it makes the simultaneous running of several jobs on the cluster more complicated to manage because only two jobs can use the storage node simultaneously (we want to run only as many processes on the storage node as there are CPUs available). Thus, our codes don't presently use this feature. Additionally, another way to speed up the I/O from individual computational nodes is to use their local hard drives; however, if, for instance, graphical postprocessing of data is intended, we'd need to collect the data at some point on the RAID anyhow, so we don't currently use the local hard drives for this purpose.

nodes, including, potentially, the storage node, the code's internal communications occur via the Myrinet, whereas the code accesses input and output files on the RAID via the Ethernet. After completing the job, the user performs the postprocessing in serial on the user node, outside the computational network. For alternative choices for the configuration of the cluster, see the "Alternative Setups" sidebar.

Physical Arrangement

Figure 2 shows a schematic of the two racks containing all the components. We received each rack essentially fully assembled and we only needed to connect the networks between them. The lettering H5 and H6 indicates their placement in the operations room of our Office of Information Technology, which houses Kali. All compute nodes are stacked in the H5 rack, with the Ethernet switch located centrally, so the compute nodes can easily connect to the Ethernet. Due to their low height of 1 U—a height unit equivalent to 1.75 inches or 44.5 millimeters—some space remains unused at the top of the 42-U-high rack. The remaining components are located in the H6 rack, with the Myrinet switch at the bottom and the storage node directly below the RAID. Because they hold more than one hard drive, both `storage1` and `mgtnode` are 2-U-high nodes. Still, significant space remains unused in the H6 rack.

Figure 3a shows the thin compute nodes `node001` through `node031` stacked up in rack H5, whereas Figure 3b confirms that several slots in rack H6 remain empty. The view of the back of rack H5 in Figure 3c shows nodes `node001` through `node020` located physically below the Ethernet switch. The gray cables around the outside are the Ethernet cables; black cables connect each node directly to its neighbor in the center. The latter are part of the third—so far, neglected—cluster network; this management network is responsible for booting the computational nodes in a daisy-chain fashion from the management node, starting at `node001` and moving up. The red fiber-optic cables in the picture are the Myrinet cables; they collect from each node downward into the floor. Figure 3d shows the back of rack H6, where the Myrinet cables emerge from the floor again and connect to the Myrinet switch. You can see the four horizontal blades in the switch, each containing eight ports. Above the Myrinet switch is the storage node and the RAID.

Prototype Problem

As I've mentioned, two fundamental reasons exist

for using parallel computing: First, using several processors in parallel to attack a problem should help obtain the solution more quickly. Second, and more fundamentally, distributing a problem onto several processors can help solve a problem that's too large for a serial machine. The following prototype problem will demonstrate both these advantages. The application concerns the flow of calcium ions in a single human heart cell.[1,2] The model consists of a system of reaction-diffusion equations with nonlinear reaction terms and a highly nonsmooth source term with a probabilistic component in the calcium equation. The simulation domain Ω represents one heart cell, which we can acceptably model as a brick $\Omega := (-X, X) \times (-Y, Y) \times (-Z, Z) \subset \mathbb{R}^3$ with one longer dimension $Z > X = Y$. Realistic numbers are $X = Y = 6.4$ and $Z = 32.0$, measured in micrometers.

To focus on parallel computing, let's consider a simpler prototype problem that neglects the reactions between the species and the calcium source but retains the realistic diffusive transport of the calcium ions. Whenever choices are required in the following, we'll let the application problem guide us. In effect, the study reported here is a thorough test of our method's core and its implementation.

Find $u(x, y, z, t)$ for all $(x, y, z) \in \Omega$ and $0 \le t \le T$ such that

$$\frac{\partial u}{\partial t} - \nabla \cdot (D \nabla u) = 0 \quad \text{in } \Omega \text{ for } 0 < t \le T, \qquad (1a)$$

$$\mathbf{n} \cdot (D \nabla u) = 0 \quad \text{on } \partial\Omega \text{ for } 0 < t \le T, \qquad (1b)$$

$$u = u_{\text{ini}}(x, y, z) \quad \text{in } \Omega \text{ at } t = 0, \qquad (1c)$$

where $\mathbf{n} = \mathbf{n}(x, y, z)$ denotes the unit outward normal vector at surface point (x, y, z) of the domain boundary $\partial\Omega$. Here, T denotes the final time for the simulations, and the diagonal matrix $D = \text{diag}(D_x, D_y, D_z)$ consists of the diffusion coefficients in the three coordinate directions. To model the diffusion behavior realistically, we pick the same values as in the application example $D_x = D_y = 0.15$ and $D_z = 0.30$ in micrometers squared over milliseconds. As the initial distribution, we pick the smooth function

$$u_{\text{ini}}(x, y, z) = \cos^2\left(\frac{\pi x}{2X}\right) \cos^2\left(\frac{\pi y}{2Y}\right) \cos^2\left(\frac{\pi z}{2Z}\right).$$

To get an intuitive feel for the solution behavior, observe that the PDE in Equation 1a has no source term and that we prescribe no-flow boundary conditions on the entire boundary in Equation 1b. Hence, the chemical will diffuse through the domain without escaping from it, starting from the nonuniform initial distribution (Equation 1c), until the chemical reaches a steady state, constant throughout the cell. Because the system conserves mass, we can analytically compute the constant steady-state solution as $u_{SS} \equiv 1/8$ for future reference.

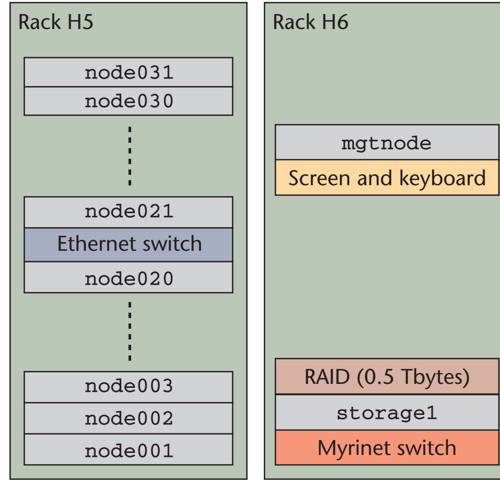


Figure 2. A schematic of the Kali components' physical arrangement in their two racks. H5 holds all the compute nodes and the Ethernet switch, whereas H6 holds the remaining components, including the Myrinet switch, the storage node, and the user node.

We can obtain the true solution for this linear constant-coefficient problem on a rectangular domain analytically, for instance, by separation of variables and Fourier analysis. We give this as

$$u(x, y, z, t) = \frac{1 + \cos(\lambda_x x)\exp(-D_x \lambda_x^2 t)}{2} \times \frac{1 + \cos(\lambda_y y)\exp(-D_y \lambda_y^2 t)}{2} \times \frac{1 + \cos(\lambda_z z)\exp(-D_z \lambda_z^2 t)}{2}, \qquad (2)$$

where $\lambda_x = \pi/X$, $\lambda_y = \pi/Y$, and $\lambda_z = \pi/Z$. Now, we can use this true solution to gauge a priori what value of the final time T is suitable for approaching the steady-state solution. We reach this steady state when all exponential function terms in Equation 2 become vanishingly small. Given the realistic val-

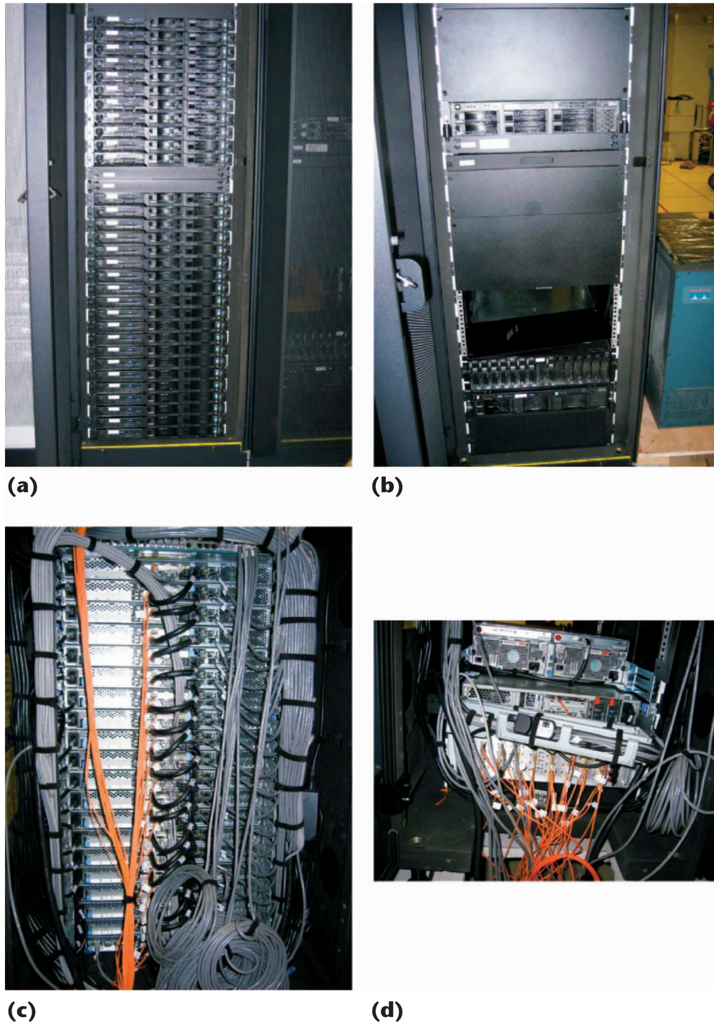


Figure 3. Photographs of the cluster. The (a) compute nodes in rack H5 and the (b) remaining components in rack H6 are connected by network cables in (c) the back of rack H5 (lower half pictured) and (d) the back of rack H6 (bottom part pictured). (Photographs courtesy of Randy Philipp, UMBC Office of Information Technology.)

ues chosen already for the diffusion coefficients and the domain size, the choice $T = 1{,}000.0$ milliseconds is the smallest suitable value because then the largest exponential term is $\exp(-D_z \lambda_z^2 t) \approx 0.055$. Although the application problem doesn't tend to a steady state, this time scale is still interesting to study because it reveals the underlying time scale associated with the diffusion in the system without source and reaction times.

Numerical Method

We've demonstrated in previous work that a method-of-lines approach using finite elements for the spatial discretization converges for the application problem,[3] and we continue using it here. General background on finite-element methods for time-dependent PDEs is available.[4,5] To use as little memory as possible, we restrict ourselves to nodal basis functions that are linear in each coordinate direction. Because the domain Ω is itself a brick, it makes sense to discretize it uniformly into smaller brick elements of volume $(\Delta x)(\Delta y)(\Delta z)$, where Δx, Δy, and Δz denote the mesh spacings in the three coordinate directions.

One challenge for numerically solving this application problem with the required accuracy results from the need for an extremely high grid resolution for the domain. Calcium ions enter the cell through calcium channels, modeled on this length scale as point sources at *calcium release units*. These CRUs are distributed throughout the cell at distances of $\Delta x_{\mathrm{CRU}} = \Delta y_{\mathrm{CRU}} = 0.8$ and $\Delta z_{\mathrm{CRU}} = 2.0$, measured in micrometers.[2] This gives a CRU lattice that's $16 \times 16 \times 32$ in our cell domain Ω. Using the rule of thumb that we wish to place at least eight mesh points between CRUs, we want to use at least 128 elements in the x and y directions and 256 in the z direction. However, to further guarantee that the mesh spacing $\Delta z := 2Z/N_z$ is approximately equal to $\Delta x = \Delta y$, we increase the resolution in the z direction and consider a mesh with $128 \times 128 \times 512$ elements for the application problem.

To understand this resolution's complexity, let's compute the number of degrees of freedom N, finite-element terminology for the number of unknowns the code must determine. In the method-of-lines approach, we're referring to the spatial discretization here because we must determine these unknowns at every time step. Denote by $N_x = N_y = 129$ and $N_z = 513$ the number of points on which the solution is based for a mesh with $128 \times 128 \times 512$ elements. This gives $N = (N_x)(N_y)(N_z) = 8{,}536{,}833$, or more thn 8.5 million unknowns. If the code stores each unknown as one double-precision number using 8 bytes of memory per number, it takes roughly 65 Mbytes to store the solution in memory. Table 1 shows these and other predictions for four possible mesh resolutions. We can see already that for the finest resolution of $256 \times 256 \times 1{,}024$, storing its solution with over 67.7 million unknowns requires roughly 517 Mbytes, a formidable number even on a workstation with, say, 1 Gbyte of memory.

A method-of-lines discretization of a PDE such as ours results in a stiff system of ordinary differential equations (ODEs).[6] To avoid the severe restriction on the time step Δt that would result from using an explicit time-stepping method in this case, we use the implicit Euler method, which—

Table 1. Predicted memory and disk space usage (results are rounded).

Resolution	Degrees of freedom N	Predicted memory need for solution (Mbytes)	Predicted memory need for serial code (Mbytes)	Predicted disk space for 1,001 frames (Gbytes)
$32 \times 32 \times 128$	140,481	1	12	3
$64 \times 64 \times 256$	1,085,825	8	91	26
$128 \times 128 \times 512$	8,536,833	65	716	207
$256 \times 256 \times 1,024$	67,700,225	517	5,682	1,641

although only first-order accurate—will require the least memory among implicit methods. An implicit time-stepping method for our linear PDE involves the solution of a linear system of dimension N in every time step. Using a conventional direct solver, such as Gaussian elimination, requires us to store the system matrix of size $N \times N$. Even in sparse storage (meaning only nonzero entries are stored), this would be prohibitively expensive for our desired N values because 27 essentially nonzero diagonals in the system matrix exist from the finite-element discretization in three dimensions.[3] We've avoided this storage cost entirely by switching to an iterative solver for this linear system—the conjugate-gradient (CG) method is appropriate for this symmetric problem—and by using a matrix-free implementation of the matrix-vector product in the iterative method. Thus, our only memory requirements are approximately 10 auxiliary vectors of dimension N in addition to the solution at the current time step. (This isn't necessarily the smallest number of auxiliary vectors possible, but because the application problem will require a couple of extra vectors, we aren't yet optimizing the code in this respect.) With the number of all significantly sized variables established, we can compute the predicted memory needed for the entire code on a serial machine (see Table 1). We see immediately that $128 \times 128 \times 512$, requiring 716 Mbytes of memory, is the finest resolution that fits on a machine with 1 Gbyte of memory. We can see already that parallel computing will yield a benefit if we can compute the solution with a finer resolution than possible on a serial machine.

Using the implicit Euler method, the time step Δt isn't restricted due to stability, so we can use any (positive) value. We exploit this by implementing automatic time-stepping, which controls Δt such that the estimator for the local truncation error satisfies a chosen tolerance.[6] The automatic step-size control is the key to saving time for simulations of transient problems with large final times such as $T = 1,000.0$. We're also using the study in this article to test this part of the code's functionality.

Output Considerations

To look at the solution at numerous time steps, we solve the transient problem, first saving solution data for several chosen time steps to disk, then postprocessing the data. To realistically determine how often to save the solution to disk, consider that the application model[2] lets the CRUs open and close with frequency $\Delta t_{\mathrm{CRU}} = 1.0$ milliseconds. Thus, we want to save the solution to disk with the same frequency; for $0 \le t \le T = 1,000.0$, we need to save 1,001 frames. Currently, we use output in ASCII format at present for best compatibility with all versions of our postprocessing software, Matlab (www.mathworks.com). To accurately capture the double-precision variables, we use 26 bytes for each number. The final column of Table 1 shows how much estimated disk space is required to save this many frames. For a parabolic problem such as Equation 1, known to have a very smooth solution, we could save significant disk space by outputting the solution at, say, every other point. In the application problem however, the probabilistic component in the nonsmooth source term can create concentration increases in any part of the domain at any time, so we need to be prepared to save the entire solution for postprocessing.

For the resolution $128 \times 128 \times 512$, for example, we need roughly 207 Gbytes of disk space to save the results of one complete simulation. Although possible in principle in the `/scratch` partition of our RAID, it could cause problems in practice if other users already have data stored. But for the finest resolution $256 \times 256 \times 1,024$, we would need 1,641 Gbytes to save all 1,001 frames of the solution simultaneously to disk; this much disk space isn't available. To work around this, we need to postprocess the solution at every time step on the fly, meaning that the solution is postprocessed in all desired ways immediately after it's computed, and then it's deleted. This also implies that using ASCII storage isn't a seriously limiting factor at present because switching to binary stor-

age using 8 bytes per double-precision number would decrease the disk space requirement by a factor of only 3.25. Although significant, the savings associated with postprocessing on the fly are much greater. This shows that output considerations are an issue for the solution of time-dependent PDEs, and having significant central disk space is necessary to facilitate the convenient postprocessing of combined data from all processors on the user node.

Parallel Implementation

On a distributed-memory cluster, the choice of data structure is crucial because it can make the difference between a method that scales well to many parallel processes and one that doesn't. We opted for the simplest one-dimensional decomposition of the data to get the cleanest code possible with clear communication patterns. The domain Ω is divided into P nonoverlapping subdomains, one on each of the P parallel processes, numbered $0 \leq p < P$. We divide Ω in the long, or z, direction. This means that each process contains approximately N_z/P x-y planes of $(N_x)(N_y)$ points. Using a column-oriented data structure—such as the one Matlab uses—makes each x-y plane's nodal values contiguous. This and the resulting simplicity of Matlab's postprocessing interface drove our choice, at the expense of numerical efficiency, which might improve by, say, a red-black ordering of the points. To fully control the memory usage, we program in C, using the Intel compiler `icc.` We use the message-passing interface MPI,[7] currently the most popular library for parallel communications, because of its portability; specifically, we use the Myricom implementation of MPI based on MPI's popular MPICH implementation.[7]

The most significant amount of communication is required inside the iterative solver as part of the matrix-free matrix-vector product, where all processes $1 \leq p < P - 1$ must receive the values on the last x-y plane of process $p - 1$ and on the first x-y plane of process $p + 1$ at every iteration; processes $p = 0$ and $p = P - 1$ each require only one of these. Correspondingly, processes $1 \leq p < P - 1$ need to send their first x-y plane to process $p - 1$ and their last x-y plane to process $p + 1$. These communications between neighboring processes are implemented by *nonblocking* `MPI_Isend/MPI_Irecv` commands,[8,9] meaning that during the communication phase, the processes can simultaneously perform other calculations that don't depend on any of the values being received. The only other type of communication that occurs in the iterative method is that of scalar products between vectors. Because the results of the scalar products are needed on all processes, `MPI_Allreduce` commands are used.[8,9] Every iteration of the iterative solver has exactly two such communications, and our memory-optimal implementation of the CG method has been shown to scale very well for at least 32 processes.[10]

Aside from calling the iterative solver at every time step, the (not yet fully optimized) ODE method itself uses three matrix-vector multiplications per time step as well as a few `MPI_Allreduce` communications per time step—one each for the solution's norm and the estimator of the local truncation error and additional ones for computing diagnostic output such as min(u) and max(u), which we choose to calculate at every time step. There are an order of magnitude fewer ODE steps than CG iterations, which is why communications in the iterative solver are the most costly.

One adage of successful parallel programming regarding communication is "don't." That is, it's best to design algorithms that communicate as rarely as possible. In MPI, the programmer explicitly requests all communications using calls to MPI functions. Thus, one of MPI's best features is that it keeps the programmer aware of all communications, which often leads to a reflection on the choice of algorithm or to improvements in the code's communication structure.

Simulation Results and Numerical Performance

We can solve Equation 1 on meshes with the four resolutions that Table 1 lists. Figure 4 shows slice plots of the solution for the resolution $32 \times 32 \times 128$ at different times; the domain's long dimension is oriented from left to right in the plots. Besides slices at the bottom and the back of the domain, which act as visual guides to the domain shape, we select slices at $x = 0$, $y = 0$, and $z = 0$ through the center of the domain and two additional ones at $z = -16$ and $z = +16$ in the long dimension. The first plot shows the initial solution's symmetry and covers the full color range from dark blue for $u = 0$ on the domain boundary to red for $u = 1$ in the center. Throughout the subsequent plots, the solution diffuses rapidly through the domain. By $t = 100$, the solution starts approaching its steady-state value of $u_{SS} \equiv 1/8$—visible in light blue in the central part of the domain up to and including the boundaries in x and y on the slice $z = 0$. We can see this process continue by $t = 200$ and note that the solution at the boundaries of $z = \pm Z$ also starts reaching the

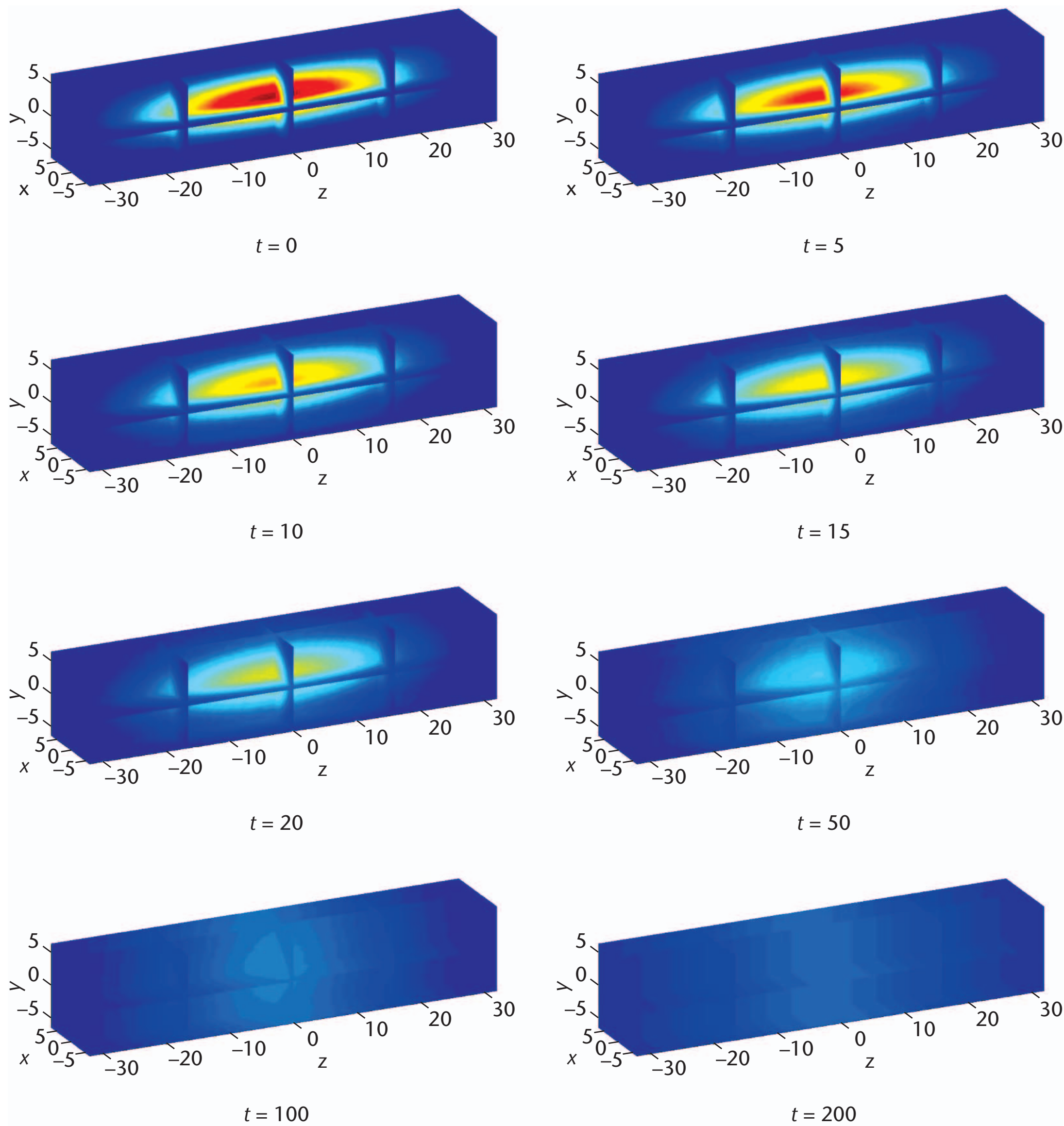


Figure 4. Slice plots of the numerical solution at specified times for resolution $32 \times 32 \times 128$. Starting from a peak at the center at $t = 0$, it approaches its constant steady state by $t = 200$.

steady-state value. For a color comparison, contrast the solution near the corners of the domain at $t = 200$ here and near the corners at $t = 0$.

To judge the numerical method's performance, we collect several diagnostic and performance variables for the finest resolution of $256 \times 256 \times 1{,}024$ (see Figure 5). Figure 5a plots the spatial minimum and maximum solution values $\min(u)$ and $\max(u)$ over the domain Ω at all times t. We see that as t grows, the minimum and maximum converge to each other and bracket the steady-state value of $u_{SS} \equiv 1/8$. To see this behavior, we had to compute to a final time on the order of $T = 1{,}000$. For this smooth problem, all coarser resolutions also show this effect correctly.

We measure the PDE error for a finite-element method in the spatial L^2-norm of the difference between the true solution u and the finite-element solution u_h at every time t, which the finite-element theory[4,5] predicts to be on the order of h^2 for $h := \max\{\Delta x, \Delta y, \Delta z\}$. Figure 5b plots this semi-discretization error against t. The error is largest during the transient phase up to approximately $t = 100$, after which it decreases as the solution changes less rapidly. As expected, the coarser spatial discretizations show progressively larger PDE errors.

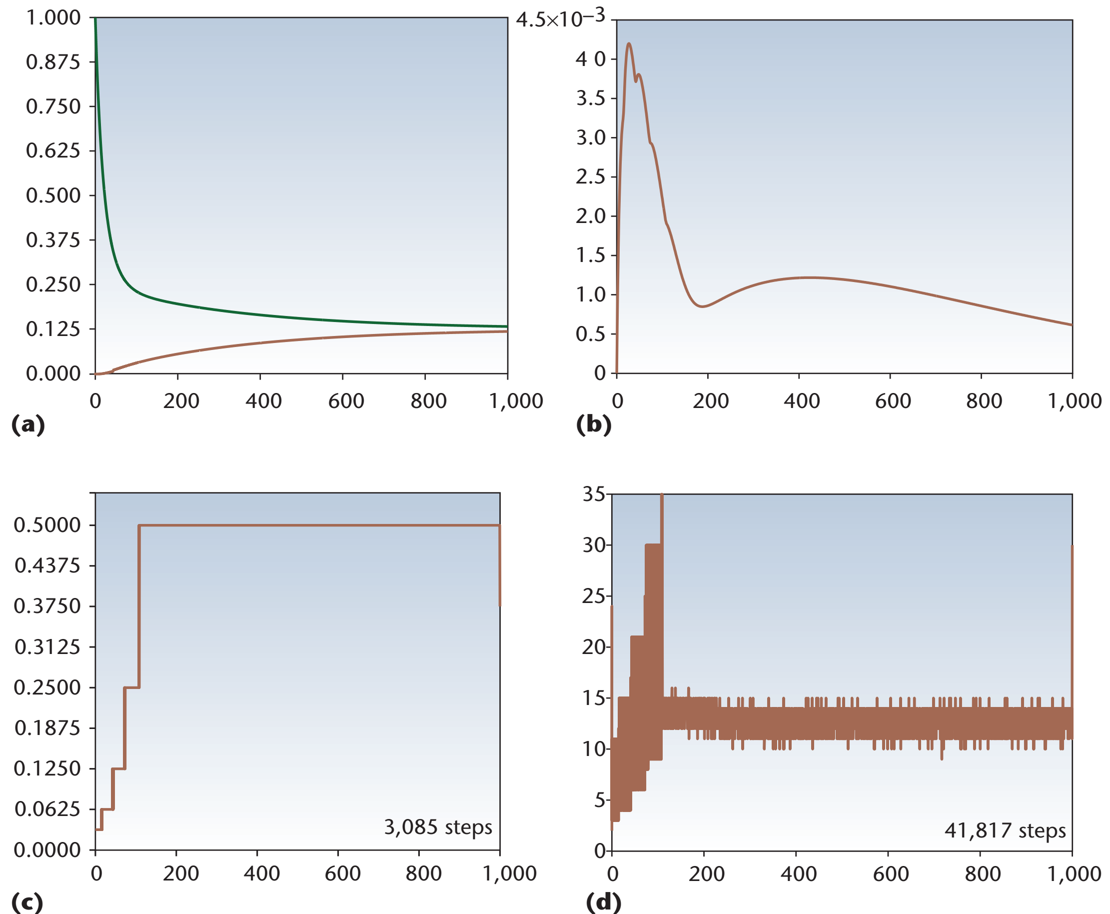


Figure 5. The diagnostic and performance output plotted versus time t for the finest resolution $256 \times 256 \times 1,024$. They demonstrate that the code computes the solution accurately and that the numerical method is efficient: (a) minimum and maximum of the solution u over domain Ω; (b) L^2-norm of the true error $u - u_h$; (c) variable time step Δt used; and (d) number of conjugate-gradient (CG) iterations required.

Figure 5c plots the local time step Δt used to compute the numerical solution at every time t; recall that the code uses variable time-stepping with automatic control of the estimated local truncation error, hence Δt isn't constant. We used a tolerance of 10^{-6} for this ODE error. The time step starts out as $\Delta t = 0.03125$, after which it doubles progressively to the maximum permitted value of $\Delta t = 0.5$. Using automatic step-size control gives us a major gain in efficiency without a loss of accuracy. This is why we use an implicit method for this problem, despite the more complicated coding compared to an explicit method; Δt is not restricted in magnitude, and the automatic step-size control can pick a large value for it. A total of 3,085 time steps are taken, over one-third of those incurred by $t = 100$. Somewhat surprisingly, the number of time steps is practically the same when using the coarser resolutions.

Recall that at every time step, we use the CG method to solve a linear system of equations. We use a tolerance of 10^{-3}; tighter tolerances didn't improve the PDE error in Figure 5b appreciably. Figure 5d reports the number of iterations this method took at every time t. It uses fewer iterations initially when the time step Δt is small and more around $t = 100$, where the solution is still changing somewhat but Δt has already increased. Then, the iteration numbers settle down to roughly 14 for the remaining time steps. Nearly half of the total 41,817 iterations are incurred by $t = 100$; for the coarser resolutions, fewer CG iterations are needed at each time step. The number of CG iterations remains reasonably small, even for the finest resolution $256 \times 256 \times 1,024$, which justifies using an unpreconditioned iterative method. This is good because it's difficult to devise a preconditioner that doesn't require additional

Table 2. Observed memory usage per process (in Mbytes).

Resolution	$P = 1$	$P = 2$	$P = 4$	$P = 8$	$P = 16$	$P = 32$
$32 \times 32 \times 128$	14	28	28	28	28	22
$64 \times 64 \times 256$	97	70	46	34	28	26
$128 \times 128 \times 512$	721	379	200	111	68	45
$256 \times 256 \times 1{,}024$	n/a	n/a	n/a	731	377	202

communications (which would make the preconditioning as costly as additional CG iterations themselves) and that can be implemented in matrix-free form (because we can't afford any additional storage).

Parallel Performance

Before discussing Kali's performance, I must first present some details regarding the parallel runs. We ran our code for the four different resolutions listed in Table 1 and with $P = 1, 2, 4, 8, 16$, and 32 parallel processes. The code run with 1 process is actually the parallel code run on 1 process, but, because all communication commands are conditional on $P > 1$, this code is equivalent to a serial code of the same algorithm. In all cases, a run with P processes uses only one CPU on each of P nodes used. Thus, each process has each node's entire 1 Gbyte of memory available.

Memory Performance

In Table 1, we analytically predicted the memory required for serial code. Using the Unix command `top`, we observe now how much memory the code actually uses per process. We understand that this might not be the most accurate way to make these observations, but, if they confirm our analytic predictions, we feel confident about their validity, whereas a significant disagreement can point to bugs (for example, memory leaks, if the observed memory usage keeps increasing over time). Table 2 collects these observations for all studies performed. For $P = 1$, we see that the observed values in Table 2 are just slightly higher than the analytic predictions in Table 1. This confirms that our predictions are reasonable and that the code doesn't use any surprising additional memory. Jumping from the 1-process cases to the cases of the finest resolution $256 \times 256 \times 1{,}024$, we notice, as expected, that the finest resolution can't run on one node. In fact, we need to use at least $P = 8$ nodes to accommodate the problem. Recall that only one parallel process runs on each dual-processor node—otherwise, this amount of memory per process couldn't even be accommodated for $P = 8$ because both CPUs on a node share the same 1 Gbyte of memory. The value of 731 Mbytes itself appears so far to agree with the predicted memory requirement of 5,682 Mbytes divided by $P = 8$. Comparing the memory usage per process from one P value to the next for the resolution $256 \times 256 \times 1{,}024$, we notice that memory per process is not halved exactly, which requires an explanation.

Also surprising is the observed memory usage for the resolution $32 \times 32 \times 128$ for more than one process—because the solution and all auxiliary vectors are now split across two processes, we'd expect the code to use less memory per process, but we observe the opposite. I believe the explanation is that the MPI libraries are actually only loaded at runtime for $P > 1$ and that they take up a major chunk of memory for such a coarse resolution. Looking at the $P = 2$ case, we hypothesize that this chunk should be 28 Mbytes minus half of the predicted serial memory of 12 Mbytes—that is, 22 Mbytes. I can now in turn try to predict the memory per process required for the $P = 2$ run of the resolution $64 \times 64 \times 256$. To the baseline 22 Mbytes, add half of the predicted serial memory of 91 Mbytes from Table 1 to get a prediction of 67 Mbytes, which is close to the observed 70 Mbytes. This formula—a baseline of 22 Mbytes plus the predicted serial memory from Table 1 divided by P—allows surprisingly accurate predictions for all other resolutions and cases of P. It also explains the behavior observed in Table 2 for the finest resolution $256 \times 256 \times 1{,}024$.

Although appearing rather pedantic, this exercise demonstrates that we can make sense of the memory usage that the operating system command `top` reports. It also shows the benefit of finding a way to compare predicted and observed memory usage carefully as a tool to debug and to optimize memory usage in a parallel code.

Speed Performance

Finally, after we've confirmed that the solution is correct (we wouldn't want a code that gives incorrect results, no matter how fast it is), that the numerical method behaves as desired (we don't want a bad numerical method), and that we can

Scaled Speedup

Our computational experiments solve problems of fixed size to observe speedup of the parallel code. An alternative is the observed *scaled speedup*, a concept in which the problem size increases (for example, doubles) with each increase (doubling) of the number of processes P. This measure keeps the calculation time and memory usage per process as high as possible on each process, which blunts the effect of the communication cost increasing with increasing P. This measure, therefore, nicely combines a demonstration of solving larger problems faster with increasing P. For our algorithm, though, this concept is difficult to apply. We could keep the memory usage per process constant as P is doubled by doubling N_z, which would have resulted in the degrees of freedom N doubling for fixed N_x and N_y. However, the complexity of a transient run of our algorithm also involves the number of conjugate-gradient (CG) iterations, which would have increased with N. Thus, the algorithm's complexity per process more than doubles with doubling P, which is inconsistent with the definition of scaled speedup. Thus, we restricted ourselves to the conventional definition of speedup; in this sense, our speedup is a tougher measure of performance because the decreasing calculation complexity per process works against us as P increases along with the increasing communication cost.

indeed solve problems of significant size, we're ready to enjoy the final benefit of parallel performance: the speedup. We time the method by obtaining the wall-clock time from `MPI_Wtime` before the start and after the end of the ODE method and computing their difference. We must use a measure such as wall-clock time for parallel code—as opposed to CPU time, for example—because communications are inherent to parallel computing and thus must be taken into account. Recall that the idea of parallel computing was to distribute the calculation work into P parallel processes and to obtain the result P times as fast. However, although dividing the calculations into P processes means that the calculation cost gets better as P increases, a truly parallel code requires communications among those P processes, whose cost gets worse as P increases. This increasing communication cost in the face of decreasing calculation cost per process quickly becomes a challenge, an effect compounded by the fact that calculations on today's workstations are several orders of magnitude faster than communications. Because of these issues, the only honest way to measure timing for a parallel code is to include both calculations and communications, which we can do by recording wall-clock time, for instance. This is a tough but realistic measure of performance because it also includes various unavoidable operating system and network slowdowns associated with real-life system operation (for an alternative measure of speedup, see the "Scaled Speedup" sidebar).

Table 3 reports the wall-clock times observed for the code in hours:minutes:seconds. Comparing the results for the $P = 1$ times for the different resolutions shows how rapidly the problem's complexity and resulting times increase when refining the numerical mesh. By using more processes, however, we can reduce the times dramatically. To put the times for the finest resolution in perspective, we can solve a problem with no fewer than 67.7 million degrees of freedom from $t = 0$ to $T = 1{,}000.0$ in less than an overnight run when using 32 nodes.

The times in Table 3 are approximately halved in nearly all cases when we double the number of parallel processes. This property begins to break down for the largest number of processes, $P = 32$. We can visualize the effect of decreasing times with increasing P by plotting *observed speedup* S_P against P. Here, we define $S_P := T_1/T_P$ for a fixed problem size as the fraction of observed time T_1 on 1 process over observed time T_P on P processes. In the optimal case, in which $T_P = T_1/P$, the speedup will then be $S_P = P$.

Figure 6a shows the *observed speedup* for all four resolutions used. The dashed line is the optimal speedup $S_P = P$. By definition, speedup plots start at the value $S_P = 1$ for $P = 1$. As the communication time increases relative to the calculation time, the speedup curves eventually fall below the optimal value. To get the comparable visual effect for the finest resolution $256 \times 256 \times 1{,}024$, we modify its definition of speedup to $S_P := 8T_8/T_P$; thus, it starts at the optimal value for $P = 8$, the smallest P available. The four lines are remarkably close to the optimal value. At $P = 16$, we have $S_P \approx 15$ for all resolutions, which is still very close to the optimal value of 16. By $P = 32$, the different resolutions start showing a range of values from $S_P \approx 28$ for the coarsest to $S_P \approx 31$ for the finest resolution. Typically, speedup is better the larger the problem's complexity. This phenomenon results from the fact that, in this case, the calculation time remains a larger percentage of the com-

Table 3. Observed wall-clock time (in hours:minutes:seconds).

Resolution	$P = 1$	$P = 2$	$P = 4$	$P = 8$	$P = 16$	$P = 32$
$32 \times 32 \times 128$	00:16:38	00:08:26	00:04:13	00:02:08	00:01:04	00:00:36
$64 \times 64 \times 256$	02:19:23	01:10:40	00:36:24	00:18:08	00:09:33	00:05:04
$128 \times 128 \times 512$	23:39:26	11:56:59	06:04:43	02:58:58	01:32:11	00:48:57
$256 \times 256 \times 1{,}024$	n/a	n/a	n/a	35:31:26	18:08:23	09:11:23

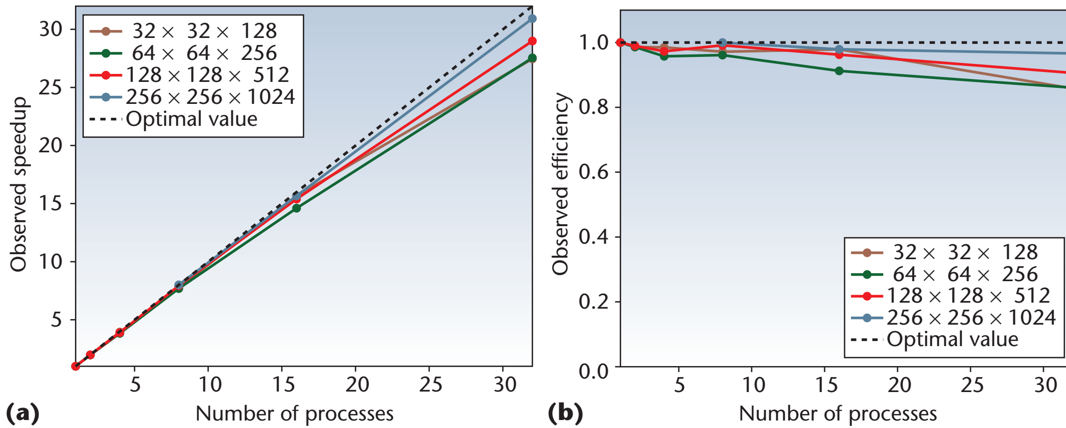


Figure 6. Speedup and efficiency. We can graphically represent our implementation's parallel performance with two measures: (a) observed speedup and (b) observed efficiency.

bined calculation and communication time. One way to gauge the complexity per process of a CPU-intensive job is to look at the memory usage per process. As Table 2 indicates, the memory usage per process falls quite dramatically for the larger P values, letting us conclude that not much calculation work is left to do per process, at least for the coarser resolutions.

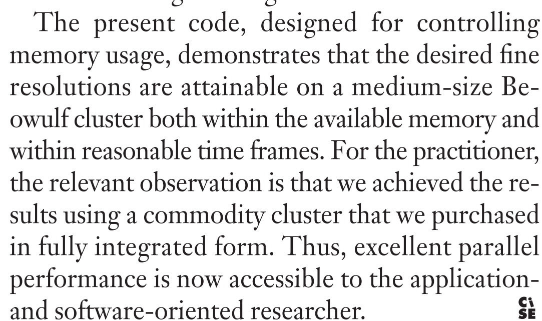
Figure 6b shows another quantity whose value can characterize the scalability of parallel code. The *observed efficiency* E_P is the ratio of speedup S_P over P. Hence, a value of $E_P = 1$ or 100 percent is optimal. The efficiency plot is often useful in bringing out certain features that are easily overlooked in the speedup plot. For instance, we see in Figure 6b that efficiency drops off by $P = 4$ for several cases, more rapidly than is visible in the speedup plot. But it doesn't drop off any further for larger values of P and is still roughly 95 percent for most resolutions at $P = 16$. At $P = 32$, we see again a range of values for the different resolutions, from approximately 87 percent for the coarsest to about 97 percent for the finest resolution. These speedup and efficiency results are excellent for a tightly coupled algorithm on a distributed-memory cluster such as this one, and illustrates the power of Kali's high-performance interconnect.

You might think at this point that we chose to use only one CPU per node in the parallel study for memory reasons alone. This isn't the case. Rather, the weak point on clusters using today's commodity CPUs is the small cache size (512 Kbytes for our Intel Xeon chips) that can quickly overload the local bus of a node for algorithms such as ours, particularly if both CPUs are being used simultaneously. The issue often manifests itself for the first time when a 2-process parallel run, using both of a node's CPUs, takes much more than half the time as a 1-process run in a parallel performance study. In reality, the problem isn't trouble with the parallel code or the hardware special to a cluster. Rather, the problem comes from the fact that algorithms such as ours incur a significant number of cache misses, and the local 32-bit bus can't serve the data fast enough from memory.

A software solution would be to redesign the algorithm to use a different ordering of the points that makes the accessed data more contiguous in memory.[11] One hardware solution would be to go to single-processor nodes, but this isn't cost-effective due to the steep cost increase that results from doubling the size of the network switches and the number of other expensive components. An-

other hardware solution, by contrast, would be to use computer chips with significantly larger cache size—available these days in several Mbytes—possibly in combination with a faster and wider 64-bit bus. But using such specialized chips would negate the fundamental cost advantage of using commodity 32-bit chips mass produced for the PC market. Moreover, for other types of algorithms that use less memory and have a less tightly coupled data structure (for example, those using explicit time-stepping for hyperbolic transport equations),[12] this type of slowdown hasn't prevented good performance results in practice. Thus, most users continue to buy Beowulf clusters such as ours with dual-processor nodes using commodity CPUs because this approach gives the best return on investment in a production environment, where throughput of as many jobs from multiple users is the ultimate goal.

Building on the present results, my colleagues and I are in the process of extending the method to the application problem. This involves solving several coupled reaction-diffusion equations similar to those in Equation 1, with additional nonlinear reaction and source terms. We're presently considering leveraging available software, for instance, by using a more general parallel computing library such as PETSc (www.mcs.anl.gov/petsc/) that in turn would call our matrix-free routines to affect the memory savings. In the long run, our goal is to simulate high-resolution PDEs, such as those used for this application problem, on commodity clusters that are affordable to typical researchers or research groups in science and engineering fields.



The present code, designed for controlling memory usage, demonstrates that the desired fine resolutions are attainable on a medium-size Beowulf cluster both within the available memory and within reasonable time frames. For the practitioner, the relevant observation is that we achieved the results using a commodity cluster that we purchased in fully integrated form. Thus, excellent parallel performance is now accessible to the application- and software-oriented researcher.

Acknowledgments

The US National Science Foundation's SCREMS grant DMS–0215373; principal investigators Jonathan Bell, Florian Potra, Madhu Nayakkankuppam, and myself; and additional support from the University of Maryland, Baltimore County, partially supported the purchase of the Beowulf cluster Kali. I thank the UMBC Office of Information Technology for Kali's setup and administration. I also thank the Institute for Mathematics and its Applications (IMA) at the University of Minnesota for its hospitality during Fall 2004. The IMA is supported by NSF funds. Finally, I thank Madhu Nayakkankuppam and Robin Blasberg for their invaluable feedback on a draft of this article.

References

1. L.T. Izu et al., "Large Currents Generate Cardiac Ca^{2+} Sparks," *Biophysical J.*, vol. 80, Jan. 2001, pp. 88–102.
2. L.T. Izu, W.G. Wier, and C.W. Balke, "Evolution of Cardiac Calcium Waves from Stochastic Calcium Sparks," *Biophysical J.*, vol. 80, Jan. 2001, pp. 103–120.
3. A.L. Hanhart, M.K. Gobbert, and L.T. Izu, "A Memory-Efficient Finite Element Method for Systems of Reaction-Diffusion Equations with Non-Smooth Forcing," *J. Computational and Applied Mathematics*, vol. 169, no. 2, 2004, pp. 431–458.
4. A. Quarteroni and A. Valli, "Numerical Approximation of Partial Differential Equations," *Springer Series Computational Mathematics*, vol. 23, Springer-Verlag, 1994.
5. V. Thomée, "Galerkin Finite Element Methods for Parabolic Problems," *Springer Series Computational Mathematics*, vol. 25, Springer-Verlag, 1997.
6. L.F. Shampine and M.W. Reichelt, "The Matlab ODE Suite," *SIAM J. Scientific Computing*, vol. 18, no. 1, 1997, pp. 1–22.
7. Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard," Argonne Nat'l Laboratory; www.mcs.anl.gov/mpi.
8. P.S. Pacheco, *Parallel Programming with MPI*, Morgan Kaufmann, 1997.
9. W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, 2nd ed., MIT Press, 1999.
10. K.P. Allen and M.K. Gobbert, "Coarse-Grained Parallel Matrix-Free Solution of a Three-Dimensional Elliptic Prototype Problem," *Proc. Int'l Conf. Computational Science and Its Applications* (ICCSA 03), LNCS 2668, V. Kumar et al., eds., Springer-Verlag, 2003, pp. 290–299.
11. Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed., SIAM, 2003.
12. S.G. Webster, "Stability and Convergence of a Spectral Galerkin Method for the Linear Boltzmann Equation," doctoral thesis, Dept. of Mathematics and Statistics, Univ. of Maryland, Baltimore County, 2004.

Matthias K. Gobbert is an associate professor of mathematics at the University of Maryland, Baltimore County (UMBC). His research interests revolve around the numerical solution of time-dependent partial differential equations, and he enjoys working with researchers in science and engineering whose computationally significant problems often involve systems of differential equations, high-dimensional domains, the necessity for fine resolution, and other challenges. Gobbert received a PhD in mathematics from Arizona State University. He is a member of SIAM, the American Mathematical Society, and the Electrochemical Society. Contact him at gobbert@math.umbc.edu; www.math.umbc.edu/~gobbert/.