BEOWULF CLUSTER PERFORMANCE FOR A DYNAMIC PROGRAMMING SOLUTION OF A TWO-CLASS QUEUEING NETWORK

JONATHAN R. SENNING

Abstract. We describe an experiment designed to determine if a dedicated parallel cluster is effective in solving certain queueing network problems using dynamic programming. We show that the same domain decomposition method that is commonly used for solving elliptic partial differential equations with finite differences is effective for this class of problems and yields roughly the same parallel efficiency. Finally, we suggest that a modest program of research in high performance computing at Gordon College is both desirable and achievable.

1. Introduction

Over the past year I have become convinced that now is the right time to explore high performance computing (HPC) at Gordon College. My own interest in this area was rekindled as I worked with my colleague Michael Veatch to solve queueing network problems, many of which are computationally and memory intensive. Several other colleagues in the natural sciences also have either specific projects or on-going research that can make use of parallel processing power. There are many research projects that are accessible to our students with programming experience, and exciting opportunities exist for computer science students to collaborate with students in other disciplines. This paper documents a very small first step toward a program of research in high performance computing that I hope will be established here.

2. Background and Testing Configuration

2.1. **Background.** In the late 1980s and early 1990s my research involved creating effective strategies to solve modeling problems on shared memory multiprocessor computers. These computers have relatively few processors and communication between them takes place via memory and other internal mechanisms. In contrast, distributed memory parallel computers can have any number of processors each with access to its own memory. Communication between processes running on different processors requires an external network. Today the lines between these types of parallel computers are blurred. Many CPUs have multiple *cores* (computational units that work independently of each other) so current parallel computers today are likely to be hybrids that have both distributed and shared memory.

Some types of problems can be parallelized so easily that they are called *embarrassingly parallel problems*. The work necessary to solve these problems can be divided into multiple tasks that can be independently carried out on different

Date: March 14, 2009.

processors without little to no communication required between them. In most cases, however, problems do not partition so nicely and some portions of the program that solves the problem must be executed sequentially. Any communication between processes will likely further delay the program.

The performance of a parallel algorithm is measured by its speedup and its efficiency. The *speedup* of an algorithm is the ratio of the time it takes to run on a single-processor machine to the time it takes to be completed by N processors working together:

$$\text{speedup} = \frac{\text{time for serial solution}}{\text{time for solution on } N \text{ processors}}.$$

Most embarrassingly parallel problems have near perfect speedups; that is a speedup of nearly N on N processors. Other problems typically have speedup curves that fall off from ideal indicating that the relative effectiveness of each additional processor drops as N increases. The*efficiency* of a parallel implementation is computed by

$$\text{efficiency} = \frac{\text{speedup achieved on } N \text{ processors}}{N}$$

and gives a measure of the performance of a parallel machine relative to the number of processors it has. Assuming that the cost of a parallel computer is proportional to the number of compute nodes it has, the efficiency can be used to assess the cost-to-benefit ratio of increasing the number of nodes.

2.2. **The Test Cluster.** In 2007 we installed eight new Hewlett-Packard workstations and a file server, all running Linux, into our departmental workstation laboratory. All that is necessary to combine these workstations into a parallel cluster is a software library (or *stack*) that allows for parallel processes to be started and stopped and to communicate with one another. The Message Passing Interface (MPI) standard was defined and implemented by various vendors so that software could be written for message passing systems in a platform-independent manner. While several commercial close-source MPI implementations exists, the academic research community seems to settled on two open-source versions: MPICH2[1] and OpenMPI[2] (the successor to LAM/MPI).

Although the entire computer science workstation cluster is configured so that it can be used as a parallel computer, the workstations comprising the cluster are also used for other work and this can make it difficult to gather accurate performance data. For this project we (temporarily) constructed a true "Beowulf" cluster consisting of a head node and four dedicated compute nodes from commodity hardware and running open-source software. The cluster configuration is shown Figure 1 and the hardware is listed in Table 1. The compute nodes do not require internal disks, monitors or keyboards. Two different network options, a fast 1000Mbps (1000 megabit per second or "gigabit") network and a slower 100Mbps network, were tested to see the effect of network latency on performance.

The head node was configured with Caos NSA Linux[3] and Perseus[4], a cluster provisioning toolkit that manages the cluster. The software was installed and configured on the head node in less than one hour and the compute nodes require

[1] *http://www.mcs.anl.gov/research/projects/mpich2*
[2] *http://www.open-mpi.org*
[3] *http://www.caoslinux.org*
[4] *http://www.perceus.org/portal/project/perceus*

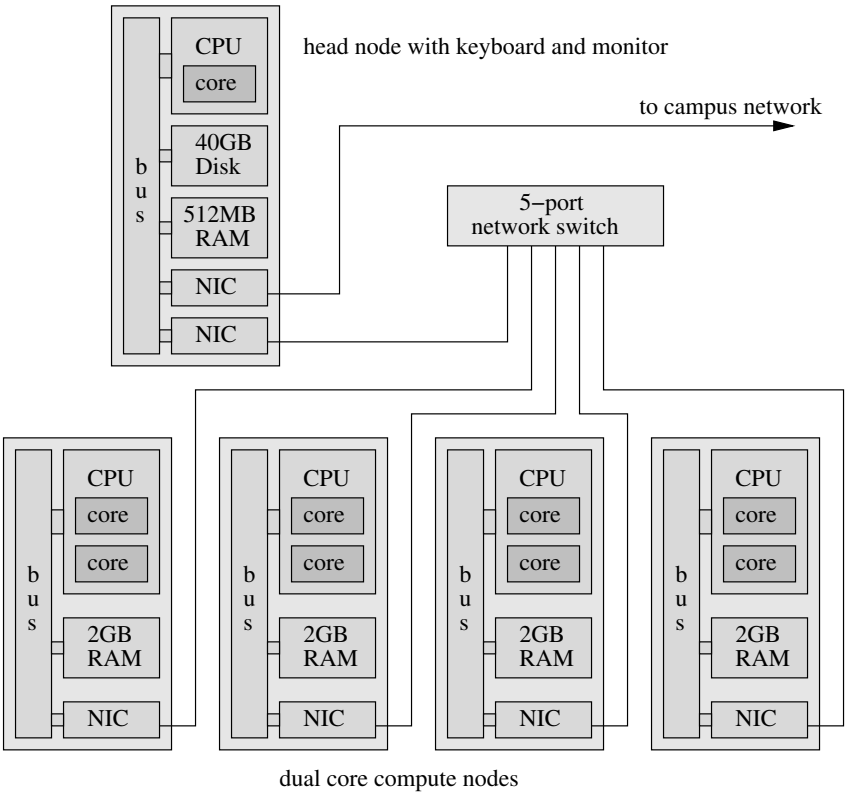

FIGURE 1. Four-node Beowulf Cluster

Head Node (1)	1GHz Intel Pentium 4 Processor 512MB RAM 40GB disk integrated 100Mbps Ethernet Controller (NIC) PCI 100Mbps Ethernet Controller (NIC)
Compute Nodes (4)	1.87GHz Intel Dual Core Processor 2GB RAM integrated 1000Mbps Ethernet Controller (NIC)
Network Switch	Option 1: 100Mbps Linksys 5-port unmanaged switch Option 2: 1000Mbps Netgear 5-port unmanaged switch

TABLE 1. Test Cluster Hardware

no installation at all; they merely need to be booted using PXE to request their operating system from the head node.

In addition to the two different network speeds, the fact that the compute nodes have dual core CPUs allows for additional configuration possibilities. In OpenMPI each node has one or more *slots* associated with it. Slots roughly correspond to processes so if a node has two slots then the MPI system will initially allocate two processes to the node. Normally the number of slots is set to correspond to the

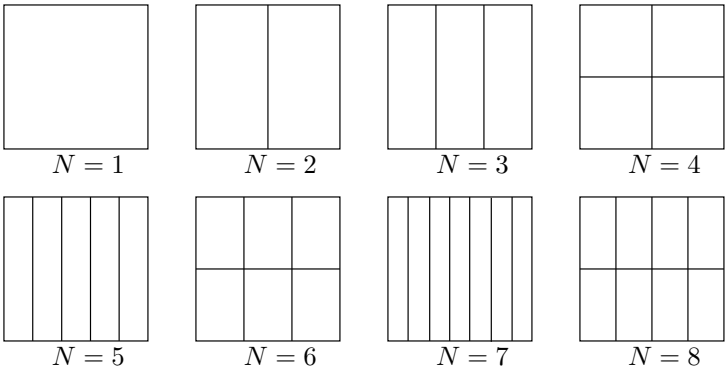

FIGURE 2. Domain Decompositions for one through eight processes

number of cores a CPU has but there may be situations when this is suboptimal. The most likely of these occur when the processes running in the multiple cores contend with each other for access to memory.

To run a program in parallel with OpenMPI the number of processes desired must be specified. The processes are assigned to slots in a round-robin fashion. For example, if there are four nodes each having one slot then the first process is assigned to the first node, the second to the second node, etc. The fourth process is assigned to the fourth node but to assign the fifth process the allocation scheme cycles back to the first node. If the same four nodes each have two slots, then the first two processes are assigned to the first node, the next two are assigned to the second node, etc. Using either scheme, the test cluster is fully utilized when eight processes are assigned. When only four processes are assigned, however, the first scheme uses one CPU core in each of the four compute nodes while the second scheme uses both cores in each of two compute nodes.

2.3. **The Test Programs.** Two parallel MPI-based programs were written in Fall 2008. The first program solves a boundary value problem with a two-variable elliptic partial differential equation using finite differences. The second program is a parallel version of a dynamic programming solution to a two station series line queueing network.

Both of these programs are iterative. During each iteration all the values on the interior of a two-dimensional array are updated using neighboring values. A common way to parallelize such problems is through *domain decomposition*. The entire array (the domain) is partitioned it into multiple smaller arrays (the subdomains) that overlap along their boundaries. The MPI standard defines routines to partition a rectangular array into a Cartesian array of subdomains. Using the defaults provided by OpenMPI gives the various partitionings shown in Figure 2. OpenMPI seems to prefer to create partitions along both grid dimensions when possible, but fall back to partitioning only the first dimension when it is not.

An embarrassingly parallel problem was also written to provide a comparison with these programs. We chose the midpoint rule approximation of a definite integral on an interval, a problem frequently used for this purpose.

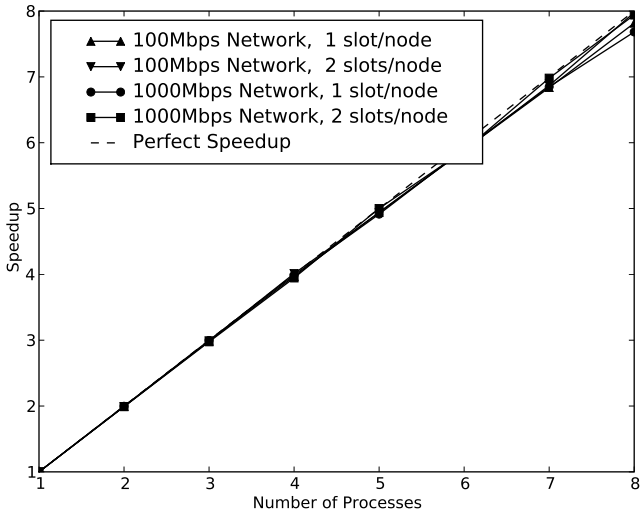

FIGURE 3. Midpoint rule results

3. Numerical experiments

Each of the three different programs described above were tested on each of four different configurations of the parallel cluster: slow network and 1 slot per node, slow network and 2 slots per node, fast network and 1 slot per node, and fast network and 2 slots per node.

3.1. **Midpoint rule for integration.** As a reference and example of speedup achievable with a (nearly) perfectly parallel program we use a program to estimate π by evaluating the integral

$$\int_0^1 \frac{4}{1+x^2}\,dx$$

using the midpoint sum rule with $200,000,000$ intervals. Each process computes the value of the sum for a range of intervals and then these sums are collected into one sum that represents the value of the integral. All the work except for the collection of sums (called a *gather* operation) can be done completely in parallel, making this an embarrassingly parallel problem.

Timing and performance data for the two different network configurations and two different slot configurations of the parallel cluster are reported in Table 2. The table shows, for differing number of processes, the program's running time in seconds, the speedup achieved, and the parallel efficiency. The speedup data is plotted in Figure 3.

This problem is *computation bound* (as opposed to *memory access bound* or *communication bound*) so it is not be surprising that nearly perfect speedup was achieved for all four different machine configurations.

Number of Processes	100Mbps Network					
	1 slot per node			2 slots per node		
	Seconds	Speedup	Efficiency	Seconds	Speedup	Efficiency
1	4.123	1.00	100%	4.144	1.00	100%
2	2.070	1.99	100%	2.077	2.00	100%
3	1.381	2.99	100%	1.386	2.99	100%
4	1.039	3.97	99%	1.033	4.01	100%
5	0.834	4.94	99%	0.840	4.94	99%
6	0.700	5.89	98%	0.704	5.88	98%
7	0.603	6.84	98%	0.603	6.87	98%
8	0.528	7.82	98%	0.520	7.98	100%
Number of Processes	1000Mbps Network					
	1 slot per node			2 slots per node		
	Seconds	Speedup	Efficiency	Seconds	Speedup	Efficiency
1	4.137	1.00	100%	4.139	1.00	100%
2	2.072	2.00	100%	2.080	1.99	99%
3	1.380	3.00	100%	1.391	2.98	99%
4	1.037	3.99	100%	1.049	3.95	99%
5	0.842	4.92	98%	0.828	5.00	100%
6	0.701	5.90	98%	0.704	5.88	98%
7	0.604	6.85	98%	0.593	6.98	100%
8	0.539	7.68	96%	0.521	7.94	99%

TABLE 2. Midpoint rule for definite integral

3.2. **Finite differences for Helmholtz equation.** The second program solves the elliptic partial differential equation

$$\bigtriangledown^2 u - 0.04u = 0$$

on the unit square using the method of finite differences. The domain is discretized with a grid with N subintervals in each direction and the solution of the equation is approximated at each of the grid points; in the test case $N = 499$, resulting in a 500×500 grid.

The iterative Red-Black SOR method was used to carry out the solution. Each process independently updates values in its subdomain but then the values along the subdomain boundaries must be communicated to processes working on the adjacent subdomains. This causes a synchronization point during each iteration and requires an amount of communication (with a corresponding delay) that increases as the number of subdomains increases.

Table 3 lists the timing and speedup data for this problem and Figure 4 compares the speedup data to the perfect speedup curve.

These results are significantly different than those for the last problem. Notice how the increase in network speed from 100Mbps to 1000Mbps dramatically improves the speedup and efficiency. This confirms that communication speed is a bottleneck in the parallel solution of this type of problem and efforts to increase communication speed may be fruitful.

At least three interesting features can be seen Figure 4. First, notice that when two processes are used on the fast network cluster we see that the 1 slot per node

Number of Processes	100Mbps Network					
	1 slot per node			2 slots per node		
	Seconds	Speedup	Efficiency	Seconds	Speedup	Efficiency
1	7.069	1.00	100%	7.040	1.00	100%
2	6.181	1.14	57%	6.332	1.11	56%
3	5.352	1.32	44%	6.033	1.17	39%
4	6.031	1.17	29%	4.840	1.45	36%
5	4.821	1.47	29%	4.619	1.52	30%
6	5.873	1.20	20%	4.052	1.74	29%
7	5.921	1.19	17%	4.351	1.62	23%
8	5.178	1.37	17%	4.000	1.76	22%
Number of Processes	1000Mbps Network					
	1 slot per node			2 slots per node		
	Seconds	Speedup	Efficiency	Seconds	Speedup	Efficiency
1	7.070	1.00	100%	7.002	1.00	100%
2	3.839	1.84	92%	6.024	1.16	58%
3	2.552	2.77	92%	3.818	1.83	61%
4	2.699	2.62	65%	2.999	2.33	58%
5	2.050	3.45	69%	2.093	3.35	67%
6	2.544	2.78	46%	1.990	3.52	59%
7	1.985	3.56	51%	1.740	4.02	57%
8	2.232	3.17	40%	1.792	3.91	49%

TABLE 3. Finite differences for Helmholtz Equation

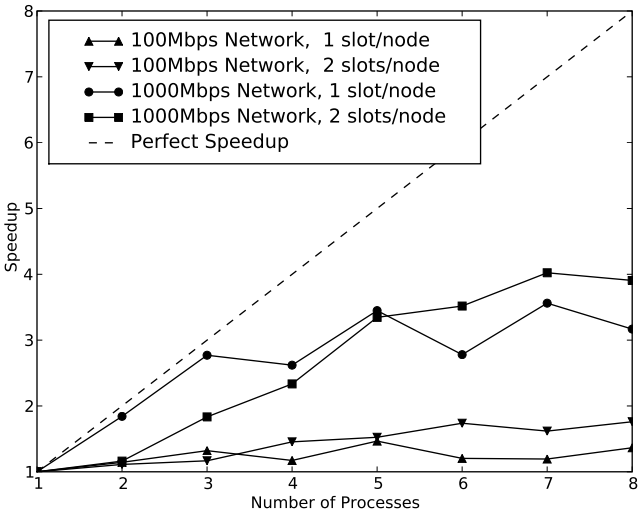

FIGURE 4. Finite differences results

configuration dramatically outperforms the 2 slot per node configuration. Recall that two different nodes work together in the first of these configurations while only

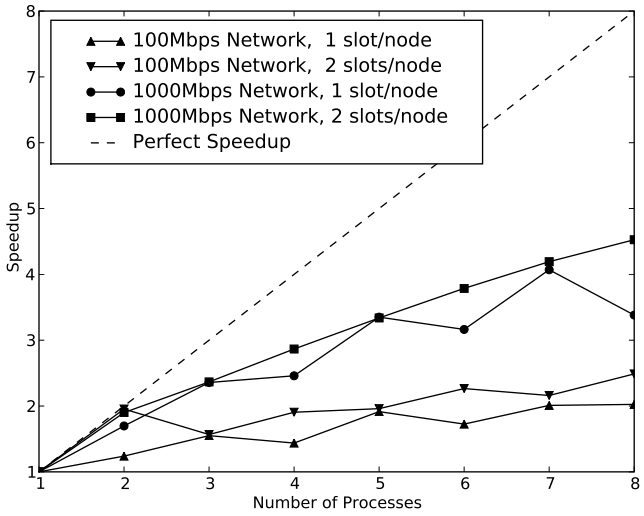

FIGURE 5. Series line dynamic program results

one node is used in the second. In both cases exactly two CPU cores are involved. This suggests that memory contention is a significant issue when both cores are working in a node.

Second, notice that the 1 slot per node, fast network configuration achieves nearly perfect speedup for 1, 2, and 3 processes but that the speedup falls off dramatically when for 4 processes. This effect is probably due to OpenMPI's default partitioning scheme. Observing Figure 2 we see that only one axis of the domain grid is partitioned when $N = 1, 2, 3$ but that both axes are partitioned when $N = 4$. The same multi-directional partitioning occurs for $N = 6$ and $N = 8$ and is likely the reason for the jagged, sawtooth effect in the fast network speedup curves. The 2 slot per mode configuration seems to be much less sensitive to the partitioning scheme.

Finally we note that, for both the slow and fast network configurations, using only 1 slot per node is more effective than using two slots per node when the number of nodes is small. As the number of nodes increases, however, and especially when both cores in the nodes are utilized, we find that the 2 slot per node configuration performs best.

3.3. **Dynamic programming solution of series line.** The third program implements a dynamic programming solution to a series line network queueing problem with two stations. During each iteration the values in each subdomain are updated using adjacent values and then values along the subdomain boundaries are exchanged. Table 4 shows the timing, speedup, and efficiency information for a problem with queue truncations of 300 resulting in a 301×301 grid.

While the speedup curves shown in Figure 5 are similar to those for finite differences, there are some interesting differences. First, when only two processes were run on the 2 slot per node configuration the program achieved nearly perfect

Number of Processes	100Mbps Network					
	1 slot per node			2 slots per node		
	Seconds	Speedup	Efficiency	Seconds	Speedup	Efficiency
1	26.85	1.00	100%	26.89	1.00	100%
2	21.67	1.24	62%	13.76	1.95	98%
3	17.32	1.55	52%	17.16	1.57	52%
4	18.68	1.44	36%	14.11	1.91	48%
5	14.02	1.92	38%	13.72	1.96	39%
6	15.56	1.73	29%	11.87	2.27	38%
7	13.36	2.01	29%	12.45	2.16	31%
8	13.26	2.02	25%	10.81	2.49	31%
Number of Processes	1000Mbps Network					
	1 slot per node			2 slots per node		
	Seconds	Speedup	Efficiency	Seconds	Speedup	Efficiency
1	26.83	1.00	100%	26.93	1.00	100%
2	15.81	1.70	85%	14.17	1.90	95%
3	11.38	2.36	79%	11.38	2.37	79%
4	10.91	2.46	61%	9.40	2.87	72%
5	8.01	3.35	67%	8.07	3.34	67%
6	8.48	3.16	53%	7.11	3.79	63%
7	6.59	4.07	58%	6.42	4.19	60%
8	7.93	3.38	42%	5.95	4.53	57%

TABLE 4. Series line dynamic program

speedup. We would not expect this behavior if memory access was a limiting factor so it appears that memory contention is less of an issue for this problem than for the finite differences problem.

Communication between processes still represents a bottleneck. When the number of processes is increased to 3 we observe a rapid drop in performance on the slower network 2 slot per node configuration while the drop on the fast network configuration is much less pronounced.

In the 1 slot per node configuration we once again observe the sawtooth appearance in the speedup curve that we saw in the finite differences case but the effect is almost nonexistent in the 2 slot per node case, especially for the fast network configuration.

4. Conclusions

The objectives of this research were twofold: (1) demonstrate that effective MPI-based parallel solutions exist for dynamic programming problems like those encountered in queueing network research, and (2) carry out a modest project to inaugurate a program of research in high performance computing at Gordon College.

A dynamic programming solution to a simple network problem was implemented using MPI and run on a four node, eight core parallel cluster. Its efficiency compared favorably to that of a parallel finite difference solution to an elliptic partial differential equation problem.

The usefulness of the dynamic programming approach to queueing network problems is limited by the *curse of dimensionality* as both computation time and memory requirements increase exponentially with the number and size of the network queues. When implemented on distributed memory architecture parallel solutions to these problems not only reduce the computation time also allow individual nodes to work with a portion required data. Although it does not scale well as the number of network queues is increased, such a parallel implementation does allow for longer queues or perhaps an additional queue.

The results here demonstrate that the speed of the network connecting the compute nodes directly affects the efficiency when solving dynamic programming problems. Gigabit Ethernet provides a relatively inexpensive medium. Its effectiveness is often enhanced by equipping the compute nodes with a second network interface. This allows for computation related communication to proceed independently of other network traffic (control, disk access, etc.).

Much of the research presented here could easily be done by undergraduates. Apart from knowledge of dynamic programming, carrying out this project required a basic understanding of parallel programming, knowledge of C++ and OpenMPI, and access to a small dedicated parallel cluster. There are multiple textbooks and freely available manuals and tutorials that students can use to learn how to use the necessary tools.

E-mail address: jonathan.senning@gordon.edu

Department of Mathematics and Computer Science, Gordon College, 255 Grapevine Road, Wenham, MA 01984