Proceedings of FEDSM 2000:
ASME Fluids Engineering Division Summer Meeting
June 11-15,2000, Boston, MA

FEDSM2000-11223

PARALLELIZATION OF POTENTIAL FLOW SOLVER USING PC CLUSTERS

Prof. Blair.J.Perot Manjunatha .N. Ramanathpura

Department of Mechanical and Industrial Engineering, University of Massachusetts,
Amherst, MA - 01003

Abstract

An analysis of the performance of low cost PC clusters for scientific computations is presented. The solution of Laplace's equation is analyzed in detail on two small PC clusters, a shared memory machine, and a single processor machine for three different problem sizes. Detailed analyses of the communication costs indicate that PC clusters are suitable for large-scale scientific computations if communication can be overlapped with computation. However, small amounts of non-overlapped communication, such as the dot products in the Conjugate Gradient algorithm, can severely impact the parallel performance of otherwise highly parallel programs.

1.Introduction

Due to extremely large sales volumes, desktop processors such as the Pentium II offer very good performance for their price. Connecting a number of these machines together into a PC cluster can further enhance the power of these low cost processors. The cluster nodes are connected via a fast network allowing all the processors to operate on a single problem in unison. Large PC clusters, such as the ASCI Red machine with 9,072 processors [1], are among the fastest supercomputers available today reaching speeds in excess of 10^{12} floating point operations per second (teraflops). Smaller machines of the order of 32 to 128 processors are now common in the academic setting. They provide computational performance comparable to million dollar commercial supercomputers at about one-tenth the cost.

The PC cluster-computing model has the potential to radically alter how large-scale scientific computing is performed in the future. This paper evaluates the actual performance of a number of small PC clusters when numerically solving Laplace's equation using an unstructured mesh algorithm. Laplace's equation arises in numerous physical situations (potential fluid flow, conductive heat transfer, electric fields, stress analysis, etc), and when solved with an unstructured mesh displays most of the important numerical characteristics that are important to performance on PC clusters. Laplace's equation should be reasonably indicative of a wide variety of other large-scale scientific computing applications.

Processors can be connected together in two fundamentally different ways. The first paradigm is shared-memory machines where multiple CPUs exist on a single motherboard and communicate via the same memory. Dual and Quad processor machines of this type are common, and some configurations with up to 14 CPUs are available. However, all but the dual processor machines are very expensive per CPU. The other paradigm is to connect processors via a network such as Ethernet. Machines that are connected via a network can not easily access the memory of other machines and must communicate with each other via message passing over the network. PC clusters often take advantage of both paradigms, networking together a number of shared memory (dual or quad processor) machines.

The paper will evaluate the tradeoffs associated with various PC cluster configurations and discuss the algorithmic requirements required to parallelize an unstructured Laplace solver on such machines. We will attempt to define the limits of current PC cluster technology and identify the current bottlenecks associated with this type of supercomputer architecture.

2. Problem Description

The problem that was considered is the inviscid flow around circular cylinder. Figure 1 shows the relevant boundary conditions and final solution (streamlines) for this problem. The initial condition for this problem was a simple uniform flow.

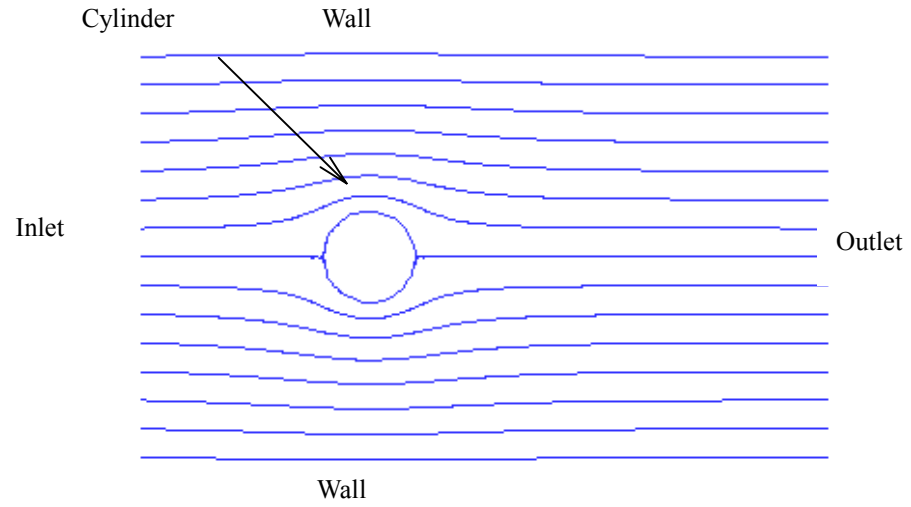

Figure 1. Test Problem: Flow (streamlines) around a circular cylinder.

The potential flow solver is an in-house code which uses a finite volume discretization, unstructured meshes, and a Jacobi Preconditioned Conjugate Gradient (CG) solver [2]. The CG solver is an iterative method, and convergence was stopped when the relative solution error was less than 0.01%. This level of convergence took on the order of 200 to 650 iterations depending on the number of unknowns (mesh size) of the problem. Three different mesh sizes were tested: a small mesh with 16,410 unknowns, a medium size mesh with 34,448 unknowns, and a large problem size of 114,128 unknowns.

The problem was tested on four different hardware configurations as shown in Figure 2. Configuration A is just a standard single CPU machine. Configuration B is a Dual processor machine. The two CPUs in configuration B can communicate data via their common memory. The second two

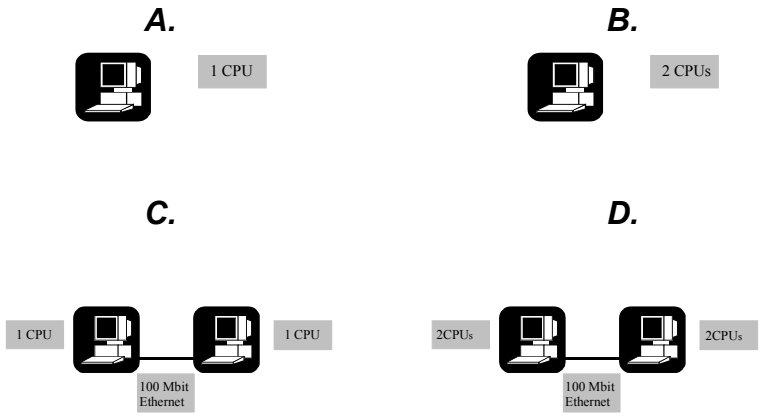

Figure 2. Cluster Configurations

configurations are simple PC clusters. Configuration C is constructed from two single processor machines connected via 100 Mbit Ethernet and Configuration D is two dual processor machines connected via fast Ethernet. Configuration D utilizes both communication paradigms of shared memory and message passing. All the machines involved in these tests contain 400 MHz Intel Pentium II processors running Windows NT. The message-passing interface (MPI) is Wmpi version 1.2, a public domain implementation of MPI from Coimbra, Portugal [3]. MPI is a library specification for message passing between the cluster of machines, proposed as a standard by a broadly based committee of vendors, implementers, and users.

Effective partitioning is an important technique for achieving better performance of parallel algorithms since it minimizes the amount of data that must be passed between the processors. Figure 3 shows an example of the simple mesh partitioning used for this work. Partitions were constructed by constructing contiguous strips where each strip contains the same number of

unknowns. Figure 3 shows the partitioning used for hardware Configuration D. Note that the second partition from the left is slightly wider since it contains most of the empty cylinder.

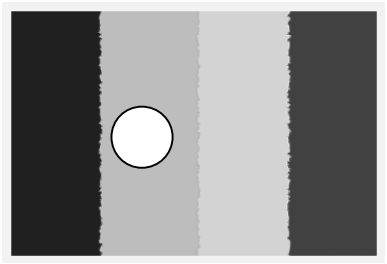

Figure 3. An example of mesh partitioning.

3. General Performance Results

Timings for the various hardware configurations and problem sizes are shown in Table 1. In most cases the problem was actually solved many times in order to get stable timing results. In addition, since the different problem sizes require different numbers of iterations for convergence, we have reported the more relevant values of the time per iteration (in milliseconds), and the efficiency of the parallel implementation compared to the single processor configuration (A).

Configuration	**A** (1)	**B** (2)	**C** (1/1)	**D** (2/2)
Big Problem				
Time/iteration (10^{-3})	88.503	51.684	45.603	26.6030
Efficiency	1.0	0.8562	0.9704	0.83171
Medium Problem				
Time/iteration (10^{-3})	25.405	15.207	14.364	9.4034
Efficiency	1.0	0.8353	0.8843	0.6754
Small Problem				
Time/iteration (10^{-3})	11.034	5.623	7.242	6.522
Efficiency	1.0	0.982	0.762	0.423

Table 1. Time per iterations and parallel efficiency for each hardware configuration and problem size.

Efficiency is calculated by using the below formula

$$\eta = \frac{t_1}{n \times t_2}$$

Where t_1 is the time taken to solve the problem on single processor machine, t_2 is the time taken by each process, for the same problem, on cluster of machines, and n is the number of processors in the cluster. This is effectively a measure of the total time taken by all CPU's to solve a certain problem verses the total time it takes a single CPU to solve the same problem.

The large problem shows a nearly linear increase in performance as the number of CPU's is increased. Interestingly, Configuration C with 2 CPUs communicating over the network, is more efficient than Configuration B where two CPU's communicate via shared memory. This does *not* indicate that the network communication is faster than the shared memory communication. It is an artifact of the background operating system processes running on these machines. In Configurations A and C background processes are performed on a second CPU which is not being used for the test problem. However, Configurations B and D use all the available processors and so the background processes must compete with the test problem for CPU time. These background processes do not use much CPU time, but the context switching on each CPU does reduce the available CPU time and also the cache hit rates. The roughly 15% efficiency loss in configurations B and D is almost entirely due to background process contention for the CPUs.

The medium and small problems show a significant decrease in efficiency compared to the larger problem. The decrease is particularly pronounced for Configuration D. This is because for smaller problem sizes the communication times begin to become more important. Communication time is almost independent of the size of problem when the code is reasonably well partitioned and the sizes of data exchanges over the

network are small. The time taken to exchange a small data item is referred to as the communication latency. On our system the latency is roughly 0.7 milliseconds which is equivalent to roughly 200,000 real number adds by the CPU. So even a few communication operations can soon impact a computationally intensive algorithm. The impact of communication time is analyzed in detail in the next section

4. Detailed Performance Results

The iterative solution of Laplace's equation via the Jacobi preconditioned CG method requires one matrix multiply, two dot products, and three vector add/multiplies (vector1 = vector1 + scalar*vector2). Only the matrix multiply and dot products require communication between processors. The communication in the matrix multiply can be overlapped with the matrix multiply calculation. In this way, the processor remains busy while the data travels over the network, and only becomes idle if the data has not arrived by the time the calculation is complete. The dot products require far less data to be communicated but can not be overlapped with any calculation. The full dot product is accomplished by performing partial dot products on each CPU and the adding the result from each CPU together. The partial dot products must be completed before the summation over the network can occur, and the resulting total dot product is required before the CG method can proceed.

Table 2 shows the detailed time distribution for different configurations and different size of problems. Time is given in milliseconds per iteration. Communication and computation times are separately shown in order to give clear idea of tradeoffs between different configurations. The communication time shown for the matrix operation is the time it takes to initiate the communication and then possibly wait for the result. It is the CPU time used by matrix communication. The actual time to send a message is typically much longer but not of direct interest as long as the CPU is busy during that time.

Table 2. reveals a number of very interesting observations. First, the time spent in the matrix communication is relatively small for all the configurations tested. There is sufficient computational load to hide the communication time if the matrix multiply is appropriately programmed. The matrix communication time is also roughly constant. The variations are thought to be due to statistical variation or slight differences in the computational load on the CPU's. Tests indicate that with the current hardware configurations that the matrix communication time can be hidden by computation as long as the number of unknowns per CPU is of the order of 1000 or more. 1000 unknowns per CPU is quite small for computationally intensive scientific problems requiring parallel hardware.

Configuration	**Total Time**	**Communication**		**Computation**	
		Matrix	**Dot Product**	**Matrix**	**Other CG**
For big problem					
(B) 2	51.684	0.336	0.299	32.202	18.838
(C) 1/1	45.812	0.414	0.751	27.005	14.584
(D) 2/2	26.587	0.374	2.996	14.268	8.961
For medium problem					
(B) 2	15.208	0.237	0.227	10.481	4.263
(C) 1/1	14.364	0.472	1.172	8.094	4.626
(D) 2/2	9.403	0.333	2.954	3.69	2.426
For Small problem					
(B) 2	5.623	0.146	0.179	3.052	1.905
(C) 1/1	7.242	0.601	0.771	3.236	2.624
(D) 2/2	6.522	0.216	1.156	1.338	3.792

Table 2. Detailed timings for different configurations

The communication time required by the dot product is much more detrimental to the parallel performance. It can be seen that the dot product communication increases with the number of CPUs involved, and with the number of communications that most occur over the network. The time taken for the dot product communication is not small and becomes a significant fraction of the total solution time for the small problem (up to 15%). It is this one operation which is impacting the parallel efficiency of the small problem in Table 1. For Configuration D, the dot product communication step adds a total of eight real numbers. The relative cost for these eight adds is extremely high.

In order to demonstrate the impact the dot product communication has on the parallel efficiency of the code, the times for the medium size problem have been recalculated assuming that the dot product communication time was zero. The results are presented in Table 3. and should be compared with the original results in Table 1. It is clear than dot product communication is the only major impediment to the parallel efficiency of the code on PC clusters.

configuration	A	B	C	D
Time/iteration	25.405	14.981	13.192	6.449
Speedup	1.0	0.848	0.963	0.985

Table 3. Performance of the code for the medium size problem if dot product communication is assumed to be negligible.

There are a number of ways to reduce the dot product communication time. Developing a CG method where the dot product communication can be overlapped with computation would be effective. Parallel CG methods [4] have been devised which have a single effective dot product (rather than two) and an additional vector add/multiply. Chebychev iteration [5] is similar to CG and requires no dot products. The difficulty of Chebychev iteration is that external eigenvalues of the matrix must be accurately known. Other iterative methods, such as multigrid [6], also do not require dot products. Finally, faster communication hardware could be used. Network hardware with latencies at least 10 times smaller than 100 Mbit Ethernet are currently available [7], though much more expensive.

5. Threads vs. MPI

On shared memory machines or clusters of shared memory machines it is possible to perform communication among the processors on the same machine using operating system

Threads. Threads can allow a program to take advantage of multiple CPUs while allowing the CPUs to see the same memory locations. Allowing the CPUs to see the same memory locations allows the processors to communication via memory. This should be more efficient than passing messages between the CPUs. Configurations B shown in Figure 1. was tested using a Thread based communication paradigm rather than MPI to determine the performance tradeoffs associated with parallelizing the code using Threads. It was found that the Thread based code performed less than 10% faster. Table 4 shows the time taken by Thread communication and MPI for the small problem on Configuration B.

Communication Method	Total time taken
Threads	10.001
MPI	11.034

Table 4. Comparison of Threads vs MPI (Time in milliseconds per iteration) for Configuration B.

Since Threads can not be used to communicate between machines (over the network), MPI is still required for PC clusters. The complexity of combining threads and MPI to perform communication is probably not worth a 10% speed increase. Thread communication might become more attractive if shared memory emulation, such as Brazos from

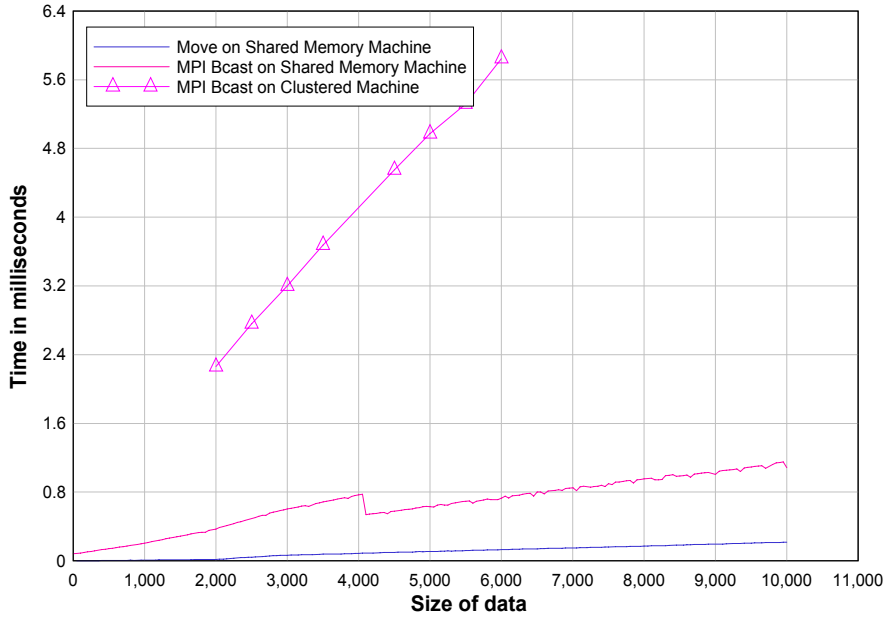

Figure 4. MPI Communication vs. Move

Rice University [8] was in place. Shared memory emulation is allows the operating system to address memory on other machines just as it addresses its own. While an attractive idea, shared memory emulation is probably not developed sufficiently for production scientific computations at this time.

6. Comparison of MPI Communication vs. Move

Figure 4. shows the time taken for moving data within the shared memory machine to that of MPI communication for both a shared memory machine as well as a cluster machine. We notice that a move (write to memory) takes almost negligible time when compared to the MPI Broadcast on a shared memory machine. This is why in the Table 4 we see a gain of roughly 10% using the Thread communication. In addition, the time taken by MPI communication on a clustered machine over the network is almost 8 times more than that of MPI on a shared memory machine. This explains the significant drop in efficiency of clustered machines over stand alone shared memory machines.

7. Conclusion

Solutions of Laplace's equation with different numbers of unknowns have been computed on four different hardware configurations. Two of the hardware configurations (C and D), require communication over a network and are representative of small PC clusters. One of the hardware configurations (A) is representative of shared memory multiprocessor machines. It has been found that PC clusters connected via 100 Mbit Ethernet are a viable hardware alternative for large scale parallel computing. The important caveat being that for smaller problem sizes, all communication has to be overlapped with useful computation. Even small communication operations, such as the dot product communication, which only adds four real numbers, can have a severe impact on the performance of the overall algorithm, if that communication is not overlapped with calculation. While Conjugate Gradient and related Krylov subspace methods are normally considered to be highly parallel this is not the case when applied to PC cluster architectures due to the presence of dot products.

At this time, scientific computations where all communication can be overlapped with computation are well suited to PC cluster architectures. However, overlapping of computation and calculation is probably not possible to accomplish with an optimizing compiler, and is not always possible in many supposedly parallel algorithms, such as standard CG. Code modification and possibly algorithm changes are required to effectively use PC clusters. Faster networking can obviate the necessity for this type of code modification and algorithm changes. However, we believe that network speeds will not be increasing significantly faster than CPU speeds in the near future, and so the basic imbalance will probably remain for some time unless expensive networking equipment is used.

References

1. James L. Tomkins, The ASCI Red Super computer, www.llnl.gov/asci/sc96fliers/snl/ASCIred.html.

2. William H. Press, Brian P. Flannery, Saul A. Teukolsky & William T. Vettering, Numerical Recipes, the Art of Scientific Computing, Cambridge University Press.

3. Win32 Message Passing Interfaces (WMPI1.2) dsg.dei.uc.pt/wmpi/intro.html.

4. A.Chronopoulos & C. Gear, s-step iterative methods for symmetric linear systems, J. Comput. Appl. Math., 25 (19), pp. 153-168

5. S. Ashby, T. Manteuffel, & J. Otto, A comparision of adaptive Chebyshev and least squares polynomial preconditioning for Hermitian positive definite linear systems, SIAM J. Sci. Statist. Comput., 13 (1992), pp. 1 - 29

6. Seokkwan Yoon & Dochan Kwak ,Multigrid Convergence of an LU Scheme, Frontiers of Computational Fluid Dynamics,Wiley publications.

7. Giganet Inc., www.giganet.com/

8. Brazos Parallel Programming Environment, www-brazos.rice.edu/brazos/