CONF-9410291--1

LA-UR-94-4339

Los Alamos National Laboratory is operated by the University of California for the United States Department of Energy under contract W-7405-ENG-36.

TITLE: A Practical Approach to Portability and Performance Problems on Massively Parallel Supercomputers

AUTHOR(S): David M. Beazley
Peter S. Lomdahl

SUBMITTED TO: Proceedings of "Performance Tuning for Parallel Computing Systems", Chatham Cape Cod, MA, October 3-5, 1994

Los Alamos

Los Alamos National Laboratory
Los Alamos, New Mexico 87545

FORM NO. 836 R4
ST. NO. 2629 5/81

DISCLAIMER

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

A Practical Approach to Portability and Performance Problems on Massively Parallel Supercomputers

David M. Beazley
Department of Computer Science
University of Utah
Salt Lake City, UT 84112

Peter S. Lomdahl
Theoretical Division
Los Alamos National Laboratory
Los Alamos, NM 87545

December 8, 1994

Abstract

We present an overview of the tactics we have used to achieve a high-level of performance while improving portability for a large-scale molecular dynamics code SPaSM. SPaSM was originally implemented in ANSI C with message passing for the Connection Machine 5 (CM-5). In 1993, SPaSM was selected as one of the winners in the IEEE Gordon Bell Prize competition for sustaining 50 Gflops on the 1024 node CM-5 at Los Alamos National Laboratory [6]. Achieving this performance on the CM-5 required rewriting critical sections of code in CDPEAC assembler language. In addition, the code made extensive use of CM-5 parallel I/O and the CMMD message passing library. Given this highly specialized implementation, we describe how we have ported the code to the Cray T3D and high performance workstations. In addition we will describe how it has been possible to do this using a single version of source code that runs on all three platforms without sacraficing any performance. Sound too good to be true? We hope to demonstrate that one can realize both code performance and portability without relying on the latest and greatest prepackaged tool or parallelizing compiler.

1 Introduction

One of the promises of massively parallel supercomputing has been its ability to open up whole new areas of exciting computational challenges for scientists in a wide range of fields from physics to medicine. However, as research in the last decade has shown, tackling new problems on parallel machines also requires one to rethink old algorithms and learn new programming techniques to make use of the parallel programming environment. While parallel machines have been promoted as having very high performance, more often than not, getting high performance on a particular machine is much more difficult than most vendors would like you to believe. For this reason, many researchers have sounded the call to develop a standardized set of parallel programming tools and languages to make parallel programming easier [9].

Unfortunately it seems that many of the efforts to develop tools and languages have sacrificed code performance in favor of portability or ease of use. Often times, solutions are overengineered or not realistically applicable to the problem at hand. Given the fact that the main motivation for using a parallel machine in the first place is to get high performance for solving a

large-scale problem, compromising performance in favor of using a completely overengineered tool or language claiming to do everything seems unacceptable.

In this paper we present on overview of our real-life experiences developing an application code for the CM-5 and later porting it to the Cray T3D and workstations. During our code development, we found the state of parallel programming tools/language development to be disappointing at best and of little consequence whatsoever to final outcome of our project. Rather than relying on others to make the latest and greatest parallelizing compiler or tool, we adopted an aggressive approach of doing everything ourselves using whatever means are available on any given machine. As we will show, this approach has not only greatly simplified code development, but is has allowed us to achieve a high degree of code portability without sacraficing performance. It has also allowed us to keep up with the extremely rapid pace of computing technology since we have not had to rely on specialized tools to be developed for every machine we decide to use. While tool developers have the best of intentions, it is our belief that high code performance does not come for free. It never has in the past and it won't in the future (if you don't believe this, consider the difficulty of getting 30% of the peak performance solving a real problem on your desktop workstation). Furthermore, we contend that no compiler or tool is going to understand your problem as well as you do. With all of the disclaimers now out of the way, we hope that our approach will provide an alternative to application programmers who are disappointed with the current state of parallel programming tools or languages.

2 Molecular dynamics and the need for speed

Our research interests have been in the area of large-scale molecular dynamics in order to study the dynamical properties of materials such as fracture, dislocation dynamics, and ductile-brittle transition. The idea behind an MD simulation is really quite simple; one solves Newton's equations of motion $F = ma$ directly for a large collection of N atoms [1]. To simplify this general N-body problem for materials simulations, we assume each atom only interacts with other atoms that are nearby. Thus, a cutoff distance r_{max} is specified and any atoms that are further away from each other than r_{max} do not interact (this is valid in many materials due to screening effects). While a short-range molecular dynamics simulation greatly simplifies the problem, we are still faced with a considerable computational challenge due to relatively short length and time scales accessible. Realistic simulations must often involve a very large number of atoms and be run for a large number of timesteps. Atoms may also interact according to complicated many-body potentials. The details of our algorithm are not discussed here, but they can be found in [4, 5, 6].

One of the goals of our work has been to model macroscopic properties in materials. This problem is especially difficult considering the fact that a speck of dust can contain considerably more than a billion (10^9) atoms. Simulations with more than a billion atoms are still out of reach today, but there has been considerable interest in developing codes capable of simulating more than 100 million atoms on parallel machines[2, 3, 5, 6, 8]. To date, the largest test simulation that has been performed involved 600 million atoms which we recently performed on a 1024 node CM-5.

Figure 1 shows a snapshot from a recent fracture experiment involving a plate of 104,031,072 atoms (roughly $1620 \times 1620 \times 40$ atoms) While this may seem like alot of atoms, this plate would only be about 0.3 microns wide with a thickness of 0.008 microns (as a sidenote, it is interesting to point out that it is now possible to produce integrated circuits with these length scales so such simulations are nearly large enough to provide direct comparison with experiment). In the experiment, a notch is placed in the side of the plate and the plate pulled apart under a constant strain rate of

0.002. This is done by introducing an initial velocity profile and slowly expanding the boundaries in the x-direction. There are periodic boundary conditions in the x and z directions. The atoms interact according to a tabulated short-range pair-potential. As the simulation progresses, the plate fractures when the overall strain reaches approximately 4%.

Figure 1. Fracture experiment with 104,031,072 atoms

This simulation required more than 150 hours of CPU time on a 512 processor CM-5 and used 11.6 Gbytes of RAM. A total 8550 timesteps were performed with each timestep requiring approximately 58.7 seconds. Of that time, 21.3 seconds were spent handling the 14473 message passing operations performed by each node in a single step. In addition, the code performed 906 gigabytes of file I/O and presented massive visualization problems as each output file was more than 1.6 Gbytes in size. Visualization was performed by Mike Krogh of the Advanced Computing Laboratory using a 256 node CM-5.

Hopefully by now, we have convinced the reader that simulations with more than 100 million atoms present a formidable computational problem involving a significant amount of computation, communication, I/O, and visualization complications. Even more sobering is physical reality. A 100 million atom simulation is still a pretty small simulation when compared to real materials. Furthermore, 8550 timesteps only corresponded to approximately 85 picoseconds of real time. Lastly, this simulation used a relatively simple inter-atomic interaction. Simulating materials such as metals or silicon will require many-body potentials. We have implemented several of these potentials and to provide a point of comparison, performing a similar experiment with silicon would have required more than 1100 hours of CPU time on a 512 node CM-5. Considering the fact that we'd like to perform billion atom simulations over several nanoseconds, we will clearly need a rather substantial improvement in computing power. Of course it's becoming increasingly clear that we are not going to be able to perform an MD simulation of the 10^{25} atoms in a glass of beer anytime in the forseeable future[11].

3 Portability Issues

In developing our CM-5 code, code portability was a nonissue. We used any means possible to get high performance on the CM-5 including optimizations to improve communications overhead and CDPEAC assembler to use the vector units[5]. This was certainly justified considering the 30 million dollar cost of the CM-5 and our desire to see what kinds of problems could be performed on such a machine. Recently, we were faced with porting this code from the CM-5 to the T3D. This section describes our approach which focuses on three main areas; eliminating CPDEAC, fixing the message passing, and solving the I/O headache.

3.1 Eliminating CDPEAC

Ironically, our choice to use CDPEAC on the CM-5 made our code more portable than if we had used CMF or C*. The approximately 1000 lines of CDPEAC assembler code were isolated to only a few code modules and the original C code was still available. Thus, removing the CDPEAC simply required adding a few conditional compilation directives and making a few small code modifications. This required less than an hour of effort and resulted in a code that was entirely written in ANSI C. ANSI C is supported on virtually every machine we are aware of so it is easily portable.

3.2 Fixing the message passing

In order to run on the T3D we needed to change all of our message passing from CMMD to a combination of PVM and Cray shared memory. Since CMMD has substantially more functionality than PVM, it was going to be moderately difficult to change all of the code to use a new library. Rather than taking this approach, we decided it would be easier to implement our own message passing wrapper library for the CMMD functions we used. The idea here is very simple; simply rename each CMMD call in the code (we replaced "CMMD" by the word "SPaSM" in the function names). The new "functions" are then implented using a combination of macros and C code. Thus on the T3D, we simply implemented the functionality of the CMMD functions listed in Table 1 using a combination of PVM and shared memory. On CM-5, this wrapper library is simply a set of macros.

CMMD_send_block	CMMD_receive_block	CMMD_send_noblock
CMMD_send_and_receive	CMMD_reduce_int	CMMD_reduce_double
CMMD_sync_with_nodes	CMMD_set_global_or	CMMD_get_global_or
CMMD_msg_pending	CMMD_all_msgs_done	

Table 1. Required CMMD functions.

This approach simplified the porting process considerably. The wrapper library could be developed and tested independently of the production code. When the library was complete it was linked in with the production code and we were able to run SPaSM on the T3D within a few minutes. The I/O capabilities were still horribly broken (see next section), but we were able to verify the correct operation of the code and run a few simple test cases. While writing our own message passing library may seem complicated, bringing the code up on the T3D required the effort of one person working for 3 days. This effort included the time to learn how to use PVM and the Cray T3D C compiler.

3.3 The I/O headache

Our CM-5 code made extensive use of the parallel I/O capabilities provided by the CMMD library. In CMMD, four basic I/O modes are possible. In CMMD_local mode, each node can manipulate files independently of the other nodes. In CMMD_independent mode each node opens the same file, but the nodes can read or write the file independently although some care must usually be made when writing. CMMD_sync_bc (synchronous broadcast) mode allows all nodes to manipulate files as if a single process were running. For example, if all nodes execute a `fprintf` statement in this mode, only one copy of the output will appear (as if printed by a single process). This mode is particularly useful for reading input parameters from files or standard input as data read is automatically broadcast to all of the nodes. Finally, CMMD_sync_seq (synchronous sequential) mode allows the nodes to write large amounts of data in parallel to devices such as the scalable disk array (SDA). In this mode, each node will write the contents of a local buffer to a single file in node order. Node 0 will write data, followed by node 1, node 2, etc... This is particularly useful for saving results and later restarting as this mode provides the highest possible bandwidth to parallel I/O devices.

Unfortunately, the T3D supports none of these modes. This leads to all sorts of bizarre I/O behavior without making some rather substantial code modifications. Early versions of our T3D code provided little or no I/O support (many features were simply removed to get the code

to compile at all). Any I/O that was provided was patched together using numerous conditional compilation directives and other hacks. Finding this situation unacceptable, we eventually decided to write our own I/O library much in the same spirit as our message passing wrapper library. Our I/O library copied the functionality of the CM-5 code. In order to provide support for different I/O modes, we had to implement wrappers around the UNIX file operations and CMMD extensions shown in Table 2.

fprintf	fscanf	read
write	fopen	open
fgets	CMMD_set_open_mode	CMMD_set_io_mode
CMMD_fset_io_mode		

Table 2. UNIX and CMMD file operations in I/O library.

Providing fully functional I/O support required a somewhat more complicated programming effort than before. The final library consisted of more than 1000 lines of ANSI C code. In order to mimic CMMD parallel I/O operations, the library maintains a list of open files and the I/O "mode" of each file. According to the file mode, nodes may coordinate in message-passing operations in order to properly read or write files. For example, file pointers and other information will be passed around to ensure that files are written properly. Similarly, when reading from standard input, node 0 will actually perform the read but will broadcast the data to all of the other nodes, mimicing the behavior of the CM-5. This all occurs behind the scenes so the actual source code does not have to worry about the details of how each I/O mode is implemented on a particular machine. While the I/O library is fairly complex, the entire library was implemented in less than week and provided a working solution to one of the major obstacles preventing real production work on the T3D.

3.4 General comments and results

This approach of writing our own message passing and I/O wrappers has dramatically improved code portability across many different platforms. Shortly after porting the code to the T3D, we were able to bring the code up on a single processor workstation in a simulated message-passing environment (the code thinks it's doing message passing, but the wrapper library simply copies buffers around). In addition, we were able to run the code on a multiprocessor SPARCcenter 2000 running Solaris using a shared-memory multithreaded approach.

It is extremely important to emphasize that we use only one version of source code that compiles on all platforms. The only difference between machines is the wrapper library used. When running on the CM-5, we use the CM-5 wrapper library. On the T3D we use the T3D wrapper. This has greatly simplified code maintenance and debugging since new code modules can be developed on a single processor workstation and later run on the CM-5/T3D without modification. This has proved to be especially useful considering the reliability and uptime of most massively parallel machines. As further proof of the effectiveness of this approach, a new code module for modeling silicon was developed entirely on a CM-5E over a period of several months. This code compiled and produced the correct results on the T3D without a single code modification.

Lastly, we should point out that this approach has resulted in no performance penalty. In fact, CM-5 performance improved slightly due to better code organization. The CDPEAC kernels are also still available due to the modular design. On other machines, by writing our own wrapper libraries we are able to optimize communications and I/O without affecting the main source code. Recently, our original PVM based message passing library was replaced by a new library that used

the Cray shared-memory library entirely. This resulted in a a factor four performance increase in communication speed.

4 Performance Issues

While portability is nice, we have not lost sight of our real goal of getting the highest performance possible. We have taken a rather aggressive approach to performance optimization on most machines. We are skeptical that any compiler is going to magically be able to map our problem onto the machine in such a way that we get high performance. Furthermore, we don't want to always be waiting for better tools or compilers in order to run fast. While it may seem crazy to some people, we argue that the best way to get high performance is to understand the underlying architecture of the machine. In particular, an understanding of superscalar RISC microprocessors will be extrememly useful as this type of processor is used in most parallel machines today.

4.1 A self evaluation

Prior to the installation of the T3D at Los Alamos, we knew that getting high performance would depend on how effectively we could use the DEC Alpha in the T3D processing nodes. With this in mind, we set out on a mission to answer the question "how well does our C code use the SPARC processor on the CM-5?" We assumed if we couldn't use the SPARC very well we probably wouldn't be able to do very well on the Alpha either. To our knowledge, this question had never be addressed in any great detail on the CM-5 since most efforts were focused on using the vector units. Our goal was to see if we could understand our code's behavior before running on the T3D. Most of this work has been described in [7] so many of the details will be ommitted here.

Since our application was dominated by the force computation, we extracted this part of the code so we could analyze it in detail. To analyze the code, we dumped the assembler output from the compiler and studied it with a SPARC achitecture manual in hand [10]. By running a small test problem, we were able to determine a dynamic instruction profile as shown in Table 3 and the timing breakdown in Table 4. In addition, we developed a simple cache simulator of the 64K direct-mapped cache that produced the results in Table 5 for the same test problem.

Instruction type	Cycles	%
LOAD	33703588	32.0%
FP (floating point)	27908776	26.5%
ALU (integer)	18874774	17.9%
STORE	14585631	13.8%
CONTROL (branches)	7079666	6.7%
NOP (no operation)	3234324	3.1%
Total	105386759	

Table 3. Dynamic distribution of instructions executed.

	Num cycles	Time (sec)	%
Executing useful instructions	105386759	3.19	36%
Memory access stalls	109000000	3.30	37%
Floating point stalls	47000000	1.42	16%
Unknown stalls	33000000	1.00	11%
Total		8.91	

Table 4. Time distribution of code activity

	Total	Hits	%	Miss	%
Instruction Fetch	104907504	104851000	(99.9%)	56504	(0.1%)
Data Fetch	33280936	33123800	(99.5%)	157136	(0.5%)
Stores	14552064	14536988	(99.9%)	15076	(0.1%)

Table 5. Simulated cache performance.

By performing this analysis we find that our C code is spending nearly 65% of its time stalling the processor. Most of these stalls are created by memory accesses which result in particularly bad performance on the SPARC since every memory access stalls the pipeline by 1-2 cycles (regardless of whether its a cache miss or hit). Fortunately our cache simulator indicates a low cache miss rate. This is also suggested by Table 4 as nearly 90% of the time can be accounted for (executing instructions, known memory stalls and stalls on the floating point unit).

4.2 Performance strategies

The fact that our code stalls the processor 65% of the time on the SPARC suggests several serious performance problems. The most significant of these problems is the fact that every double-precision load causes a 2 cycle stall on the pipeline [10]. This suggests that one should try to reduce memory operations as much as possible. While this may sound obvious, it is surprising to learn how many unnecessary memory operations are performed in your code. For example, in one section of code, a feature to improve modularity resulted in a series of unncessary floating point and memory operations that were executed in about 80% of the iterations in a particular loop. Working around this problem resulted in a 58% overall code speedup. Furthermore, any variables referenced by pointers in C will never be stored into registers and reused. Thus, speedups can be obtained by copying commonly used variables into local variables first (which will place values into registers). The performance strategies we have used are listed below :

- Reduce memory operations.
- Reorder floating point.
- Inlining.
- Increased use of local variables.
- Loop unrolling.

These optimizations are widely known and are probably familiar to most users. One could argue that these optimizations should be performed by the compiler. Yet, by applying these tactics directly to the code, we get huge speedups even when compiling with full compiler optimization.

It seems clear that you will almost always be able to beat the compiler at these things if you are clever. It is important to note that all of these optimizations can be made to the C code (no assembler code required).

4.3 Performance results

By applying the above strategies to our CM-5 code, we achieve a 119% speedup. Even more remarkable is the fact that the C code now only runs 2.2 times slower than the CDPEAC code even though the peak performance of the VUs is more than 20 times higher than that of the SPARC. When the same code is run on the CM-5E which uses a 40 Mhz SuperSPARC, we get a 95% speedup and the C code actually runs 5% faster than the CDPEAC code. This rather startling fact certainly leads us to wonder whether using the VUs was really worth the effort required. One of the difficulties in using the VUs is that vectorizing an inherently unstructured calculation is always going to result in extra work being performed. Clearly the difficulty in effectively using the VUs cannot be understated.

On the T3D, the same performance strategies result in a 183% speedup. As a result, the code sustains calculation rates between 27-41 Mflops/node which represents 18-27% of the peak performance. This is better performance than most other T3D applications that we are currently aware of.

As proof that the same optimizations can result in speedups on other RISC machines, Table 6 shows the performance speedups on a variety of platforms. In all cases, code was compiled with full compiler optimization. We see large speedups in all cases.

System	N	Unmodified	Optimized	Speedup
32 Node CM-5 (33 Mhz SPARC)	1024000	42.63	19.54	119%
32 Node CM-5 (CDPEAC)	1024000		8.87	
32 Node CM-5E (40 Mhz SuperSPARC)	1024000	11.37	5.83	95%
32 Node CM-5E (CDPEAC)	1024000		6.11	
32 Node T3D (150 Mhz DEC Alpha)	1024000	8.57	3.03	183%
HP-735 (99 Mhz HP-PA 7100a)	32000	4.62	1.61	187%
IBM Power2 (66 Mhz Rios 2)	32000	4.98	1.95	155%

Table 6. Performance of production code on a test problem.

5 Conclusions

While tool and language developers have the the best of intentions, we hope that we have demonstrated that both portability and performance are possible by simply taking a direct approach to the problem. Probably the best approach is to keep your program development simple and straightforward. By programming in ANSI C, we have been able to focus our entire effort on effectively using the machine rather than always trying to figure how to use a new set of compiler directives or tools for every new machine that comes along. By using wrapper libraries, we have been able to eliminate hardware dependencies from the main source code resulting in simplified code maintenance and debugging since only one version of source code is used for all machines. This simplified approach actually makes performance tuning easier because we do not need to worry about the extra layers of abstraction that a parallelizing compiler or tool would add. Instead, we have one simple task–making the RISC processor run as fast as possible. As it turns out, tactics for making

code run fast on one RISC architecture seem to be quite effective at producing speedups on other RISC architectures so this effort isn't wasted.

Lastly, we'd like to close with a philosophical note. As scientists working on a problem, our approach has given us almost complete control over all aspects of our problem. With the complexity of modern machines, this is extremely important because we are able to understand virtually all aspects of our code from the algorithms used to their mapping onto the underlying hardware. Without this knowledge, we do not see how we could believe any answers generated (it's not clear whether anyone should believe any results generated on a parallel machine in the first place considering their "proven" reliability). How could an experimental chemist or physicist believe the outcome of an experiment if they didn't understand all aspects of the laboratory techniques used to generate the data? Why should computational science be any different? We firmly believe that this is important and hope that users and software developers realize that a "black-box" approach is not necessarily the best or only way to use a parallel machine.

Acknowledgements

We'd like to acknowledge the many people who have provided assistance with this work. Adam Greenberg, Mark Bromley, Mike Drumheller, Denny Dahl, and Burl Hall of Thinking Machines Corporation provided major assistance with our questions about the CM-5 architecture and making performance measurements on the CM-5E. Wayne Vieira of Cray Research has provided valuable assistance with the T3D. We would also like to acknowledge the Advanced Computing Laboratory for its generous support and Dave Rich for his ongoing assistance. We acknowledge Mike Krogh for his visualization work and our collaborators in molecular dynamics research, Niels Grønbech-Jensen, Pablo Tamayo, Brad Holian, and Timothy Germann. This work was performed under the auspices of the U.S. Department of Energy.

References

[1] *Computer Simulations of Liquids*, M. P. Allen and D. J. Tildesley. Clarendon Press, Oxford (1987).

[2] R. C. Giles and P. Tamayo, *Proc. of SHPCC'92*, IEEE Computer Society (1992), p. 240.

[3] S. Plimpton, *Fast Parallel Algorithms for Short-Range Molecular Dynamics*, Sandia National Laboratory Report, SAND91-1144, UC-705 (1993).

[4] D. M. Beazley and P. S. Lomdahl, *Message-Passing Multi-Cell Molecular Dynamics on the Connection Machine 5*, Parall. Comp. 20 (1994) p. 173-195.

[5] D. M. Beazley, P. S. Lomdahl, P. Tamayo, and N. Grønbech-Jensen, *A High Performance Communications and Memory Caching Scheme for Molecular Dynamics on the CM-5.* Proceedings of the 8th International Parallel Processing Symposium (IPPS'94), IEEE Computer Society (1994), p. 800-809.

[6] P. S. Lomdahl, P. Tamayo, N. Grønbech-Jensen, and D. M. Beazley, *Proc. of Supercomputing 93*, IEEE Computer Society (1993), p. 520-527.

[7] D. M. Beazley and P. S. Lomdahl, *A Practical Analysis of Code Performance on High Performance Computing Architectures*, Los Alamos National Laboratory Report (preprint). (1994).

[8] Y. Deng, R. McCoy, R. Marr, and R. Peierls, *Molecular dynamics on distributed-memory MIMD computers with load-balancing*, Applied Math Letters (to appear).(1994).

[9] C.R. Cook, C.M. Pancake, and R. Walpole, *Proc. of Supercomputing 94*, IEEE Computer Society (1994), p. 126-133.

[10] SPARC RISC User's Guide, Cypress Semiconductor, San Jose. 1990.

[11] Special thanks to Dietrich Stauffer for putting things into perspective.