Parallel FEM Simulation of Crack Propagation on the AC^3 Velocity Cluster*

George Coulouris[3], Gerd Heber[1], David Lifka[3], Keshav Pingali[2], David Schneider[3], Paul Stodghill[2], Paul Wawrzynek[1], John Zollweg[3]

[1]Cornell Fracture Group, Rhodes Hall, Cornell University, Ithaca, NY 14853
heber@tc.cornell.edu, wash@stout.cfg.cornell.edu
[2]CS Department, Upson Hall, Cornell University, Ithaca, NY 14853
{chew,pingali,stodghil,vavasis}@cs.cornell.edu
[3]Cornell Theory Center, Cornell University, Ithaca, NY 14853
{glc5,lifka,schneid,zollweg}@tc.cornell.edu

Abstract:

This paper describes our experiences porting the Crack Propagation on Teraflop Computers (CPTC) testbed for fracture simulation from the IBM SP-2 to the AC^3 Velocity, a cluster of SMP's running NT. There are several points that are made in this paper. First, the process of porting the 150K lines of code in our testbed from UNIX to NT proved to be much easier than we had initially expected (with a few notable exceptions). Second, the performance of our testbed on Velocity exceeds its performance on the IBM SP-2, in particular, we see better scalability on Velocity. Third, the performance results demonstrate the scalability of our software and the underlying hardware, and we expect to achieve our performance goals on future Velocity type machines.

1. Introduction

Understanding how fractures develop in materials is crucial to many disciplines, e.g., aeronautical engineering, material sciences, and geophysics. Fast and accurate computer simulation of crack propagation in realistic 3D structures would be a valuable tool for engineers and scientists exploring the fracture process in materials. In a previous paper [4], we introduce the Crack Propagation on Teraflop Computers (CPTC) testbed, an innovative system for high performance fracture simulation. Our performance goal is to accurately solve 1000 crack propagation steps involving up to 1,000,000 degrees of freedom in 1 hour. In order to achieve this goal we require two things,

- Scalable simulation software, and
- Hardware capable of sustained teraflop performance.

In [4] we have already shown that our application achieves scalable performance on the IBM SP-2. Thus, we have confidence that our software is scalable.

Several months ago, the Cornell Theory Center (CTC) decided to move all of its high performance computing users from its aging IBM SP-2 to the AC^3 Velocity, a brand new Intel Pentium-based cluster of SMP's running Microsoft NT. Initially, we were very skeptical that this new platform would be able to deliver the same level of scalability that we had observed on the IBM SP-2. Also, we were very concerned that porting our software from UNIX to NT would be a very painful and time-consuming process. This paper describes our experiences porting the CPTC testbed to the AC^3 Velocity and discusses the performance of the resulting system.

2. Testbed Overview

Within the scope of this paper, it is sufficient to think about crack propagation as a dynamic process of creating new surfaces within a solid. During the simulation, crack growth causes changes in the geometry and, sometimes, in the topology of the model. Roughly speaking, with the tools in place before the start of this project, a typical fracture analysis at a resolution of ~10^4 degrees of freedom, using boundary

* This work was supported by NSF grants CCR-9720211, EIA-9726388, ACI-9870687, and EIA-9972853

elements, would take about 100 hours on a state-of-the-art single processor workstation. The goal of this project is it to create a parallel environment which allows the same analysis to be done, using finite elements, in 1 hour at a resolution of ~10^6 degrees of freedom. In order to attain this level of performance, our system will have two features that are not found in current fracture analysis systems:

Parallelism – We need to exploit parallelism in order to increase the amount of memory in which to store our problems and to reduce the execution time of our simulations.

Adaptivity – Cracks are generally very small compared to the structure in which they are embedded, and their growth is very dynamic in nature. Because of this, it is impossible to know a priori how fine a mesh is required to accurately model crack growth. While it is possible to over-refine the mesh, this is undesirable, as it tends to dramatically increase the required computational resources. Our system must be adaptive, both in the sense that it handles changes to the geometry that occur during fracture simulation, and in the sense that it uses adaptive *p*- and *h*-refinement in order to reduce the error in the computed solution.

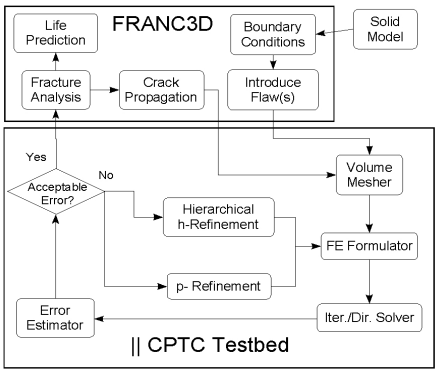

Figure 1. Simulation loop

Figure 1 gives an overview of a typical simulation. During pre-processing, a solid model is created, problem specific boundary conditions (displacements, tractions, etc.) are imposed, and flaws (cracks) are introduced. In the next step, a volume mesh is created, and (linear elasticity) equations for the displacements are formulated and solved. An error estimator determines whether the desired accuracy has been reached, or further iterations, after subsequent adaptation, are necessary. Finally, the results are fed back into a fracture analysis tool for post-processing and crack propagation.

Our code is almost exclusively written in C/C++ and uses MPI. We use a variety of third party packages including BLAS/LAPACK, PETSc [1,2], ParMETIS [6], PSPASES [6], and JANUS [8]. In all, about 150k lines of project and third-party software source code had to be ported in order to get the testbed running under NT.

3. Overview of AC^3 Velocity

The current installation of the AC^3 Velocity, designated V1, consists of 64 Dell Power Edge 6350 nodes. Each node is a 4 way SMP with Pentium III Xeons running at 500 MHz, with a 2MB L2 cache. All 4 processors on a node share 4 GB of RAM and 54 GB of disk space. All of the nodes are connected by Giganet interconnect, which has a fat tree topology and provides 1.25 Gb of bandwidth between nodes.

Velocity V1+ is currently being installed at the CTC and consists of 64 2-way Pentium III SMP's running at 733 MHz, with 256KB L2 cache. It also has a slightly faster Giganet interconnect. Velocity V2 will be an Itanium-based cluster.

At the time when the experiments discussed in Section 4 were performed, V1 was running Microsoft Windows NT 4.0 SP6a, and V1+ was running Microsoft Windows 2000 Advanced Server. We use the MPI implementation MPI/Pro 1.5 by MPI Software Technology, Inc. (MSTI) [17], which uses shared memory within nodes and the VI Architecture [9] between nodes.

A more detailed discussion of this machine's characteristics can be found in [3].

4. Porting

The full discussion of the issues encountered during the porting can be found in [3].

Our primary objective in porting to NT was to be able to take full advantage of the performance provided by the Velocity. However, while porting to NT, we had to maintain our ability to compile and run on our existing UNIX platforms. Maintaining two sets of codes, one for NT and one for UNIX was not acceptable, because keeping the two code sets synchronized would be very time consuming. This led us to our second important porting objective: maintaining a singe set of codes for both UNIX and NT.

Using most of the available development environments for NT (e.g., Microsoft Visual Studio) did not appear to be an option, as these tools do not understand or produce UNIX

Makefile's or Bourne shell scripts. Fortunately for us, there was at least one development environment that did work with UNIX Makefile's and shell scripts, namely, Cygwin [10]. The Cygwin environment consists of two basic things. First, it provides a library that sits between UNIX applications and the NT operating system and provides functions that implement UNIX system calls using their NT equivalent. The second thing that Cygwin provides is many of the important GNU development tools, such as gcc/g++/g77, GNU Make, the Bash shell, etc. Using Cygwin, we could compile and run our testbed under NT with very few modifications. The biggest change required was to add "$(EXE)" to the end of executable names in the Makefile's, and cause "EXE" to be defined as ".exe" under NT and "" under UNIX. In addition, we found that by taking a few small steps, such as rewriting Bourne scripts into the more portable Python [11], that we were able to run our testbed without having the full Cygwin environment present. This allowed us to compile our testbed on a desktop machine and then copy the executables to Velocity for execution.

Even though our testbed worked using the Cygwin compilers, we wished to try using several of the native compilers because we thought that they might provide better code optimizations. The compilers that we used during this port were,

- Microsoft Visual Studio C/C++ V6.0 [12]
- Compaq Visual Fortran 6.0 [13]
- Intel C/C++ 4.5 [14] and Fortran 4.5 [18]
- MinGW ports of the GNU compilers [15].

The first problem that we faced using the native compilers was that all of them, except MinGW, took a different set of command line options than their UNIX counterparts. For instance, the NT compilers did not recognize the standard UNIX options "-g" and "-O", but rather expect "/Z7" and "/Ox". Also, object files under NT have a ".obj" extension instead of a ".o" extension. In order to reduce our porting effort, we decided to "UNIX'ify" the native compilers. We did this quite simply by writing scripts that translate the standard UNIX compiler options into the corresponding native compiler options and that generate object files with the UNIX ".o" extension. For instance, we wrote a script, "vs_cc" for invoking the Visual Studio C compiler as if it were a UNIX compiler. Here is one example of how it might be invoked,

```
% vs_cc -v -g -O -c foo.c
+ cl /nologo /c /O1 '/DWINNT'
      '/DWIN32' '/D_WIN32'
      /Z7 /Tpfoo.cc /Fofoo.o
foo.cc
%
```

(The "-v" option tells the script to print the command line used to invoke the native compiler. The line starting with "+" is that command line). Having these scripts allowed us to use UNIX Makefile's directly and with very few medications.

In addition, we had to make small changes to source files in order for them to compile with the native compilers. These changes fall into two basic groups. First, changes had to be made to account for the fact that NT and UNIX are different operating systems. These changes included, adding the "b" qualifier to the mode argument of "fopen" to open files as binaries to prevent CR-NL translations, and changing codes that invoke system calls for high precision timing.

The second set of changes that had to be made was to account for limitations of the compilers that we used. These include the following,

- Many NT C/C++ compilers do not appear to scope identifiers declared in the "for" statement initializer according to the ANSI C++ standard. We worked around this by adding the following macro to a header file that was included in all source files:

```
#define for \
    if (false) {} else for
```

- The STL that comes with MS VS 6.0 is not standard conforming. This required us to rework our code in a few places.
- The MS VC++ compiler generated an internal error for one of our source files. We were able to eliminate the error by tweaking the code slightly.

After approximately 2 man-months of work, we are able to compile and run most of our testbed using the Cygwin and native compilers. There are a few loose ends that remain, however, that prevent us from having everything compiling and running at the present time. They are as follows:

- The MS VS C++ compiler does not correctly parse certain complicated template syntax, such as static template class member initializers ("template <int N>

`class { static const int x = N; }`") and member templates of template classes ("`template <...> class { template <...> function_decl; }`"). These constructs are used in a number of places in our code. To get the performance numbers in Section 5, we temporarily patched our code so that it would compile, but these changes were too fragile to be a long term solution.

- The Intel C++ compiler generated an internal error on several of our source files when exceptions and optimizations were both enabled. We have not yet found a way around this problem.
- We are currently unable to get one of the third party packages, PSPASES [6], to work under NT, because,
  - PSPASES is written partly in Fortran 90, and GNU G77 does not handle F90 syntax.
  - PSPASES fails when compiled with Compact Fortran compiler. This compiler uses a non-standard calling convention, and we believe that we are not correctly accounting for this.
  - The Intel C/C++ and Fortran compilers are able to compile and link our testbed and PSPASES, but, as mentioned above, the C/C++ compiler generated an internal error when exceptions and optimizations are both enabled.

5. Equation Solving Performance Results

In this section, we present performance results for the testbed's Equation Solving module. We restrict ourselves to this one module for two reasons. First, several of the modules in our testbed are still under active development and have not yet been ported to Velocity. Second, for most problems, well over 80% of the testbed's total execution time is spent in the Equation Solver, so its performance is clearly the most important.

In Table 1, we show the elapse execution time for the solver module for a medium sized problem for both, the CTC SP-2 and AC^3 Velocity. We used the Conjugate Gradient solution method and the Global Extraction Element-By-Element (GEBE) preconditioner [4,5]. The SP-2 configuration included POWER2 Super Chips (P2SC) @ 120 MHz (thin nodes), 256 MB RAM per node, TB3 switching fabric (150 MB/s peak hardware bandwidth), running AIX 4.2.1. On Velocity, the code was compiled with either Microsoft's Visual C++ 6.0 SP3, or MinGW GCC 2.95.2. On the SP-2, the code was compiled with GCC 2.95.2. On the SP-2, each node contains one processor. On Velocity, there are 4 processors in each node which allowed us to run a given processor number in different node configurations. In the table, this denoted by the different "Procs/Node", 1, 2, and 4.

Tee/QMG, 6,022 Vertices, 22,746 Tetrahedra, 110,118 dof					
Procs	Procs/ Node	SP-2	V1 (VC++)	V1 (MinGW)	V1+ (MinGW)
4	1	904.75	767.18	767.18	696.17
	2		841.10	841.10	887.14
	4		988.90	988.90	
8	1	474.65	379.05	550.39	377.81
	2		433.62	604.45	437.06
	4		535.72	701.75	
16	1	278.08	192.66	285.60	184.98
	2		226.14	306.35	220.12
	4		266.86	336.71	
32	1	158.16	97.57	152.51	107.95
	2		115.26	168.55	131.08
	4		152.08	189.32	
64	1		62.35		
	2		59.75	96.52	83.02

Table 1: Elapse Solver Time (secs.)

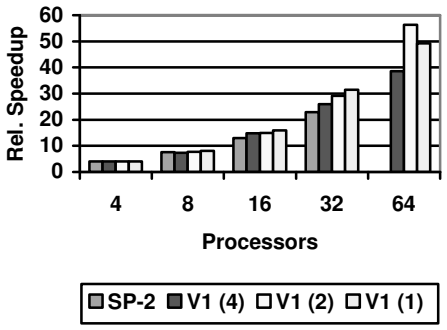

Figure 2: Relative Speedup – SP-2 vs. V1 (VC++)

Figure 2 shows the relative speedup of the solver on the SP-2 and on V1 using the VC++ compiler.

Our data shows several things,

- Velocity delivers better scalability than the SP-2 for our code. This is probably because of the Giganet's lower latency.
- For large processor numbers, the Velocity delivers better absolute performance than the SP-2.

- The V1+ delivers consistently better performance than the V1.
- Performance decreases as the number of processors per node increases. This can be attributed to the fact that all processors on a node share the same memory bus.
- The code compiled with GCC is significantly slower than that of VC++. One explanation might be that VC++ has better optimizations and code generation than GCC. Another reason might be that we are not using the most aggressive optimizations flags with GCC. We expect to resolve this question shortly.

Further performance results and analysis can be found in [3].

6. Conclusions

We learned two things as a result of this effort.

First, porting the 150K lines of code in our testbed from UNIX to NT was not nearly as difficult as we had initially expected, with a few notable exceptions. These exceptions were the non-GNU compilers that we used, with which we continue to have a great deal of difficulty. The GNU compilers under NT worked as well as their UNIX counterparts. Also, we learned the following important lessons for maintaining a single set of sources for both UNIX and NT,

- Use Cygwin, or a similar system as your development environment under NT.
- Where appropriate, rewrite Bourne shell scripts in a more portable scripting language, like Perl or Python.
- Use scripts to translate the UNIX-style compiler flags to the actual flags used by the native compilers.

The second important thing that we learned was that our testbed performed better on Velocity than on the IBM SP-2. In particular, we see better scalability on Velocity. Our performance results demonstrate the scalability of our software and the underlying hardware, and we expect to achieve our performance goals, namely solving 1000 crack propagation steps involving up to 1,000,000 degrees of freedom in 1 hour, on a future Velocity type machine.

References

1. Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object-oriented numerical software libaries. In E. Arge, A.M. Bruaset, and H.P. Langtangen, editors, *Modern Software Tools in Scientific Computing*. Birkhauser Press, 1997.
2. http://www.mcs.anl.gov/petsc/index.html.
3. George Coulouris, Gerd Heber, David Lifka, Keshav Pingali, David Schneider, Paul Stodghill, Paul Wawrzynek, John Zollweg. *Is Wintel ready for HPC? Experiences with Fracture Mechanics Simulation on an NT Cluster.* Technical Report TR2000-1795, Department of Computer Science, Cornell University. April, 2000.
4. Bruce Carter, Chuin-Shan Chen, L. Paul Chew, Nikos Chrisochoides, Guang R. Gao, Gerd Heber, Antony R. Ingraffea, Roland Krause, Chris Myers, Demian Nave, Keshav Pingali, Paul Stodghill, Stephen Vavasis, and Paul A. Wawrzynek. "Parallel FEM Simulation of Crack Propagation -- Challenges, Status, and Perspectives." *Irregular '00*.
5. Hladik, M.B. Reed, and G. Swoboda: *Robust preconditioners for linear elasticity FEM analyses*. International Journal for Numerical Methods in Engineering, Vol. 40, 2109-2127 (1997).
6. http://www-users.cs.umn.edu/~karypis/metis/
7. http://www-users.cs.umn.edu/~mjoshi/pspases/
8. J. Gerlach and M. Sato: *Generic Programming for Parallel Mesh Problems*. In Proceedings of Third International Symposium of Computing in Object-Oriented Parallel Environments ISCOOPE'99.
9. R. Dimitrov and A. Skjellum: An Efficient MPI Implementation for Virtual Interface (VI) Architecture-Enabled Cluster Computing. MPI Software Technology, Inc.
10. http://sourceware.cygnus.com/cygwin/
11. http://www.python.org/
12. http://msdn.microsoft.com/vstudio/
13. http://www.compaq.com/fortran/
14. http://www.intel.com/vtune/compilers/cpp/
15. http://www.mingw.org/
16. http://www.giganet.com/
17. http://www.mpi-softte ch.com/
18. http://www.intel.com/vtune/compilers/fortran/