

An OpenMP Approach to Modeling Dynamic Earthquake Rupture Along Geometrically Complex Faults on CMP Systems

Xingfu Wu

Department of Computer Science & Engineering

Texas A&M University

College Station, TX 77843

Email: wuxf@cse.tamu.edu

Benchun Duan

Department of Geology & Geophysics

Texas A&M University

College Station, TX 77843

Email: duan@tamu.edu

Valerie Taylor

Department of Computer Science & Engineering

Texas A&M University

College Station, TX 77843

Email: taylor@cse.tamu.edu

Abstract

Chip multiprocessors (CMP) are widely used for high performance computing and are being configured in a hierarchical manner to compose a CMP compute node in a parallel system. OpenMP parallel programming within such a CMP node can take advantage of the globally shared address space and on-chip high inter-core bandwidth and low inter-core latency. In this paper, we use OpenMP to parallelize a sequential earthquake simulation code for modeling spontaneous dynamic earthquake rupture along geometrically complex faults on two CMP systems, IBM POWER5+ system and SUN Opteron server. The experimental results indicate that the OpenMP implementation has the accurate output results and the good scalability on the two CMP systems. Further, we apply the optimization techniques such as large page and processor binding to the OpenMP implementation to achieve up to 7.05% performance improvement on the CMP systems without any code modification.

1. Introduction

There are no analytical solutions for a spontaneous dynamic earthquake rupture with a friction law such as slip-weakening friction operating on the fault. Thus, numerical methods are required to study spontaneous dynamic rupture processes on faults. The most widely used numerical codes in the field of earthquake dynamic source models are based on the finite difference method (FDM). But it is difficult for FDM to deal with complex fault geometry and complex geologic structures. Duan et al. [1, 2, 3] have been developing and using an explicit dynamic finite element method (EQdyna) to implement sequential simulations for modeling spontaneous earthquake rupture on geometrically complex faults, such as faults with bends, stepovers, or

branches. However, a sequential simulation takes time from more than 40 hours to several days for relatively small earthquake model datasets (i.e., several to ten million elements) on a SUN server with 4 dual-core AMD Opteron processors. It means waiting for several days to verify and validate a model. Therefore, it is necessary to parallelize the sequential earthquake simulation code in order to significantly shorten the simulation time by efficiently utilizing all processors within a CMP node. In this paper, we propose to use OpenMP to parallelize the EQdyna, and discuss the OpenMP implementation in detail.

Today, the trend in high performance computing systems has been shifting towards cluster systems with CMPs. Further, CMPs are usually configured hierarchically to form a compute node of parallel systems. For example, Hydra at Texas A&M University Supercomputer Facility [13] consists of nodes that have 8 DCMs (Dual-Chip Modules) with one dual-core POWER5+ processor per DCM. While CMP presents significant new opportunities such as on-chip high inter-core bandwidth and low latency, it also presents new challenges in the form of inter-core resource conflict and contention. In [8, 6], it is argued that the full benefit of these architectures will not be harnessed until the software industry and community fully embrace parallel programming. A challenge to be addressed is how well current shared-memory parallel programming paradigms, such as OpenMP, exploit the potential offered by such a CMP node for scientific applications.

OpenMP [9, 10, 7] is the most popular shared-memory parallel programming model. OpenMP is a set of compiler directives and callable runtime library routines that extend sequential programming languages such as Fortran, C and C++ to express shared memory parallelism. OpenMP provides a fork-join execution model in which an OpenMP program begins execution as a single process (master thread). The master thread executes sequentially until a parallelization directive is invoked. Then, the master thread

creates a team of threads and becomes the master of the team. The statements enclosed in the parallel region are executed in parallel by each thread until a work-sharing directive is invoked. The work-sharing directive such as “parallel do” or “parallel sections” distributes the workload among the threads. All threads need to synchronize at the end of the parallel region unless a “nowait” clause is specified. Upon completion of the parallel region, all threads synchronize and only the master thread continues execution. The advantage of OpenMP is that an existing sequential code can be easily parallelized by inserting OpenMP directives around time consuming loops which do not contain data dependencies, leaving the code unchanged. This is the most common and cost-effective way to generate a parallel program for utilizing the CMPs. Therefore, we use OpenMP to parallelize the EQdyna for exploring the parallelism of the code at node level by fully utilizing all processors.

Our validation and evaluation experiments conducted for this work utilize two CMP systems with different number of cores per node. Pangu is a SUN Opteron server with 4 dual-core AMD Opteron processors. Hydra at Texas A&M University Supercomputer Facility [13] is an IBM POWER5+ cluster with 40 p5-575 nodes, and each node has 32 GB of memory and 8 DCMs (Dual-Chip Modules) with one dual-core POWER5+ processor per DCM. Further, each system has a different node memory hierarchy. We use two production datasets to validate and evaluate our OpenMP implementation of EQdyna. The experimental results indicate that the OpenMP implementation has the accurate output results and the good scalability on the two CMP systems. Further, we apply the optimization techniques such as using the large page and processor binding to our OpenMP implementation to achieve up to 6.04% performance improvement on Pangu and up to 7.05% performance improvement on Hydra without any code modification.

The remainder of this paper is organized as follows. Section 2 describes the sequential earthquake simulation code EQdyna and its control flow. Section 3 proposes our OpenMP implementation of EQdyna. Section 4 describes the architecture and memory hierarchy of two CMP systems used in our experiments. Section 5 evaluates and explores performance characteristics of our OpenMP implementation, and presents our optimization results using large page and processor binding. Section 6 concludes this paper.

2. A Sequential Earthquake Simulation

EQdyna is an explicit finite element dynamic code which has been under development since 2005. This code is intentionally developed to simulate spontaneous dynamic rupture propagation along geometrically complex faults and wave propagation in complex geologic structures [1, 2, 3]. The sequential version of EQdyna had been verified in a

community-wide code validation exercise on seven benchmark problems [4] by the end of 2008. A brief description of mathematical and physical aspects of EQdyna can be found in [1]. As an explicit finite element earthquake simulation code, EQdyna does not need to solve a coupled set of equations for solution because the coefficient matrix is diagonal. The central difference method used in the code is conditionally stable. Therefore, the time step used in simulations, which is limited by the minimum element size and wave speed in a model, must be small enough to ensure numerical stability.

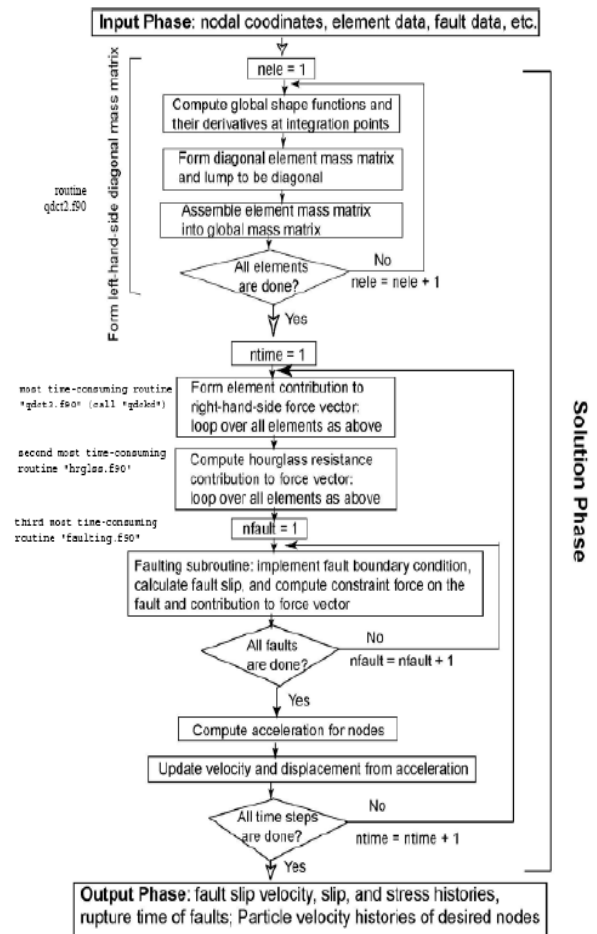


Figure 1. Control Flow of EQdyna

Figure 1 gives a control flow that displays the basic structure of EQdyna. There are three main phases in the program: Input phase, solution phase, and output phase. During the input phase, the geometrical, material and computational data in the model are read in. These data include 1) execution control parameters such as total simulation time, time step, stiffness damping coefficient, etc.; 2) nodal coordinates; 3) nodal boundary conditions; 4) initial conditions; 5) element data (topological data of

elements and material parameters for elements); 6) fault data (fault node pairs, frictional coefficients, initial shear and normal stresses, critical slip distance D_0). In addition, a couple of calculations are also performed in this phase. First, global equation numbers and assembly mapping arrays are established after nodal and element data are input. Second, the material moduli matrix is computed and stored for each type of material.

The main body of the code is the solution phase. There are four main tasks in the solution phase: 1) forming the left-hand-side diagonal mass matrix; 2) forming element contribution to the right-hand-side force vector; 3) computing the hourglass resistance contribution to the force vector; 4) implementing the fault boundary and forming the fault boundary constraint force contribution to the force vector. The first task can be executed outside of the timestep loop. The other three tasks are within the time step loop. The first three tasks involve the iterations over all elements in the model. The tasks 2) and 3), performed by the functions *qdct3* and *hourglass* respectively, are most time-consuming, because they involve loops over all elements in the model at each timestep. They provide element contribution to nodal force at each node of an element. Element contributions to individual nodes are assembled through assembling arrays established in the input phase. The output phase outputs the results for the fault data, including time histories of fault slip velocity, slip, stresses and rupture time, and time histories of particle velocity at desired nodes off the fault.

3. OpenMP Implementation of EQdyna

OpenMP parallel programming within one CMP node can take advantage of the globally shared address space and on-chip high inter-core bandwidth and low inter-core latency. The use of globally addressable memory on the CMP node allows users to exploit parallelism by inserting OpenMP compiler directives where applicable into a sequential program to generate an OpenMP program. This is the most common and cost-effective way to generate a parallel program for utilizing the CMPs. Therefore, we use OpenMP to parallelize the EQdyna for exploring the parallelism of the code at node level by efficiently utilizing all processors.

According to Figure 1, Figure 2 presents high-level structure of our OpenMP implementation of EQdyna by minimizing the number of OpenMP parallel regions. Because there are some data dependencies between timesteps and the number of timesteps is usually much smaller than the number of nodes or elements (shown in Table 2 in Section 5.1), we focus on the parallelization inside each timestep. The functions *qdct3* and *hourglass* dominates the most of execution time for the sequential EQdyna (more than 96% for two datasets we used later in Table 2), so our OpenMP implementation focuses on the two functions which consist of very time-consuming loops

with the number of iterations that equals the number of elements for the datasets. We find that there is no data dependency between the two functions *qdct3* and *hourglass*.

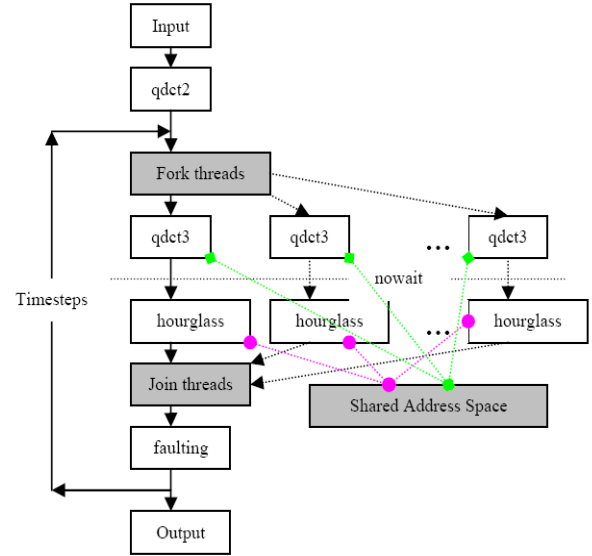


Figure 2. OpenMP parallelization of EQdyna

For the sake of simplicity, Figure 2 only shows our OpenMP implementation for the two time-consuming loops in the functions *qdct3* and *hourglass*. The OpenMP program proceeds in the fork-join execution model shown in Figure 2. First, it processes Input and *qdct2*, then enters the timestep loop. We insert a parallelization directive (`!$omp parallel`) just before the function *qdct3*. Secondly, when the parallelization directive is invoked, the master thread (with solid line) forks several new threads (with dash lines). A worksharing directive (`!$omp do`) is added inside the function *qdct3* so that the workload in *qdct3* is divided equally among the threads. Thirdly, the threads process their own workload in parallel, and share the same address space (green lines) for easily referencing data that other threads have updated. Because of no data dependency between the functions *qdct3* and *hourglass*, the worksharing directive with the `nowait` clause is added in *qdct3*. This is very beneficial because the threads that process *qdct3* continue immediately to *hourglass* without waiting for all threads to finish *qdct3* so that it can reduce the amount of time that threads are idle. After all threads finish *hourglass*, they are joined to the master thread. Then the program processes the function *faulting*, and so on.

Of course, we also use OpenMP to parallelize other loops in the earthquake simulation code written in Fortran 90. For instance, we find that large array operations like $brhs = brhs/alhs$ (where the arrays *brhs* and *alhs* with the array size of more than the number of nodes) also are time-consuming. So we use the following statements to parallelize the large array operation (where *neq* is larger than the number of nodes):

```

!$omp parallel do default(shared) private(i)
do i = 1,neq
  brhs(i) = brhs(i)/alhs(i)
enddo
!$omp end parallel do

```

Based on our experience in the OpenMP implementation, it is important to avoid various false sharings. For instance, most data is shared by default, and some data is made private explicitly in our OpenMP implementation. However, one local logical variable *zerodl* was not made private, this caused that the OpenMP program was executed much slower than its sequential counterpart because multiple OpenMP threads updated the shared data (to true or false) simultaneously and very frequently to result in the unsatisfied conditions in some if-statements. After the

logical variable *zerodl* was made private, the OpenMP program is executed very fast.

We found that parallelizing the function *faulting* caused the incorrect results, mainly because results were written out at a given time interval in the function. This function takes relatively little time in the sequential run, so we keep the function unchanged. We also tried different OpenMP implementations of the EQdyna, especially parallelizing the timestep loop, however, because the number of timesteps is much smaller than the number of nodes or elements (shown in Table 2) and there are some data dependencies between timesteps, the parallelizing the timestep loop was not an efficient OpenMP implementation. Based on our experimental results, the OpenMP implementation proposed in this paper is the most efficient.

Table 1. Specifications of two CMP systems

| Configurations | Hydra | Pangu |
|----------------|----------------|---------------------------|
| Total Nodes | 40 | 1 |
| Cores/chip | 2 | 2 |
| Cores / Node | 16 | 8 |
| CPU type | 1.9GHz POWER5+ | 2.6 GHz dual-core Opteron |
| Memory/Node | 32GB | 48GB |
| L1 Cache/CPU | 64/32 KB | 64/64 KB |
| L2 Cache/chip | 1.92MB | 1MB |
| L3 Cache/chip | 36MB | NA |

Table 2. Two Datasets of EQdyna

| Datasets | Total Nodes | Total Elements | Element Sizes | Time Steps |
|----------|-------------|----------------|---------------|------------|
| D1 | 5,017,500 | 4,917,488 | 150 m | 2,858 |
| D2 | 10,625,471 | 10,447,920 | 100 m | 7,500 |

4. Execution Testbeds

Details about the two CMP systems used for our experiments are given in Table 1. These systems differ in the following main features: number of processors per node, configurations of node memory hierarchy, CPU speed, multi-core processors, operating systems, and communication networks.

Hydra at Texas A&M University Supercomputer Facility [13] is an IBM POWER5+ cluster with 40 p5-575 nodes, and each node has 32 GB of memory and 8 DCMs (Dual-Chip Modules) with one dual-core POWER5+ processor per DCM. Hydra has the default page size of 4KB and IBM AIX 5.3, and it supports user-level large page size of 64KB using the *ldedit* or *ld* commands [5]. The SMT (Simultaneous Multi-Threading) mode is not enabled for regular use. IBM AIX provides the command *bindprocessor* to bind a process to a physical processor, and provides the

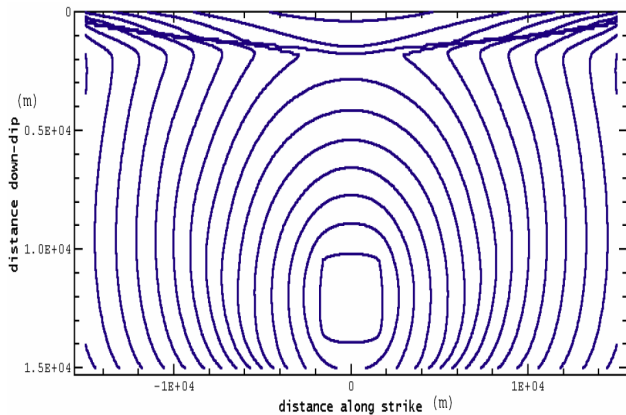
environment variable *XLSMPOPTS* to bind a thread to a physical processor. For example, *XLSMPOPTS= startproc=0:stride=2* means binding threads to different processors on different chips with one thread per chip (Note that each chip has two processor cores on Hydra).

A SUN Opteron server Pangu from Department of Geology & Geophysics at TAMU has 4 dual-core AMD Opteron processors and 48 GB of memory. Pangu has the default page size of 4KB and SUN Solaris operating system, and it supports user-level large page sizes of 2MB or 4MB using the compiler option *-xpagesize=2M* or *4M* [11]. SUN Solaris dynamically schedules OpenMP threads to physical processors, and provides the command *pbind* to bind a thread/process to a physical processor. On Pangu, we use the command *pbind* to develop a batch tool to automatically bind multiple threads to different processors in order to reduce the system overhead caused by the Solaris dynamic scheduling.

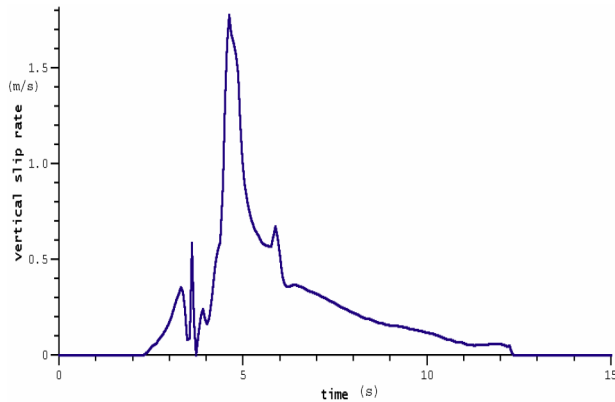
5. Experimental Results and Performance Analysis

5.1 Benchmark Problems

We work on a benchmark problem TPV10 of the SCEC (Southern California Earthquake Center) code validation exercises [4, 12] to test our OpenMP implementation of EQdyna. The benchmark solves dynamic rupture propagation along a 60 dipping normal fault (30 km x 15 km) and wave propagation in a homogeneous three-dimensional half space. Initial stress on the fault linearly increases with depth. Two datasets are generated for our tests shown in Table 2. In dataset 1 (D1), we use an element size of 150 m (i.e., the edge length of brick elements near the fault before being sheared to conform the dipping fault geometry) to create finite element mesh, with a termination time of 10 seconds for the simulation. In dataset 2 (D2), we use an element size of 100 m and a termination time of 15 seconds, which are parameters chosen by the SCEC exercise. The model sizes of the two datasets are listed in Table 2.



(a) Rupture time contours on the fault plane



(b) Vertical slip velocity at the fault station

Figure 3. Results of dynamic rupture obtained by the dataset D2. (a) Rupture time contours on the fault plane. The spacing of two adjacent contour lines is 0.5 second.

(b) Vertical slip velocity at the fault station with 0 km along both down-dip and along-strike distances

5.2 Experimental Results

To validate the OpenMP Implementation of the sequential code EQdyna, we use the above two datasets executed on these platforms such as the SUN server and TAMU Hydra in the following sections. We found that our OpenMP implementation generates the accurate output results. Figure 3 shows (a) rupture time contour on the fault plane and (b) vertical slip velocity at the fault station with 0 km of both down-dip distance and along-strike distance obtained by the dataset D2. These results have been verified within the SCEC code validation community [4, 12].

5.3 Performance Analysis and Optimization

In this section, we analyze the performance of our OpenMP implementation on the two CMP systems, and use large page and processor binding to further optimize the code.

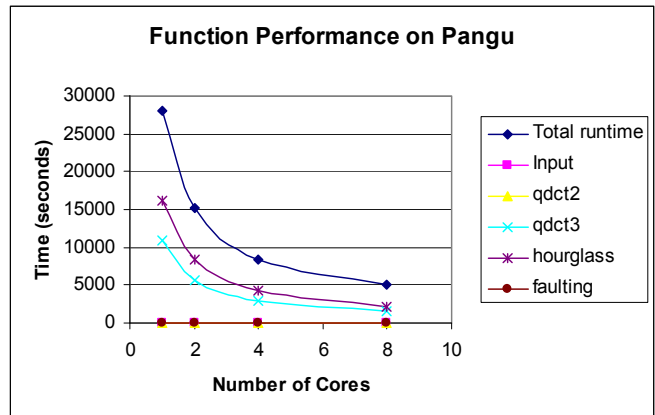


Figure 4. Function-level performance of our OpenMP implementation on Pangu

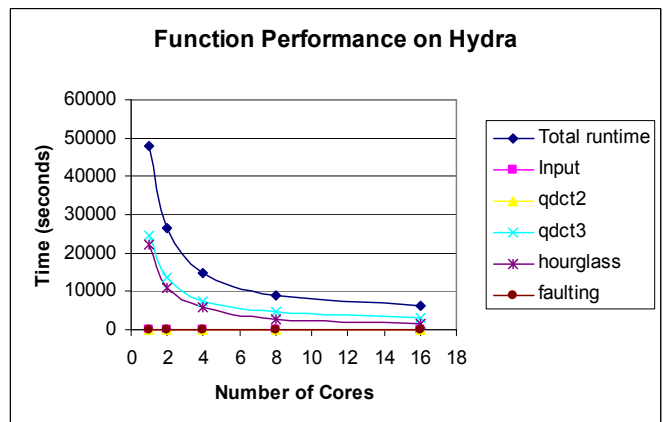


Figure 5. Function-level performance of our OpenMP implementation on Hydra

Figures 4 and 5 show the function-level performance of our OpenMP implementation of EQdyna on Pangu and Hydra. It is interesting to see that *qdct3* is the most dominated function on Hydra and *hourglass* is the most dominated function on Pangu because of different architectures. Comparing to the sequential performance, these figures indicate that our OpenMP implementation achieves significant performance improvement on the two CMP systems. It addresses that not only the OpenMP parallel programming is the most common and cost-effective way to generate a parallel program for utilizing the CMPs, but also OpenMP parallel programming within one CMP node can take advantage of the globally shared address space and on-chip high inter-core bandwidth and low inter-core latency.

Further, Figure 6 presents the scalability of our OpenMP implementation on the two CMP systems. The OpenMP implementation achieves the speedup of 1.84 on 2 cores, 3.35 on 4 cores, and 5.60 on 8 cores on Pangu, and 1.80 on 2 cores, 3.25 on 4 cores, and 5.29 on 8 cores on Hydra. Although the OpenMP implementation has the good scalability, the efficiency (ratio of the speedup to the number of cores) is decreased with increasing the number of cores.

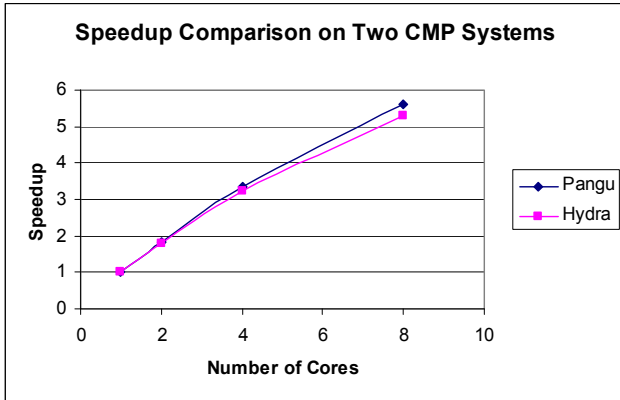


Figure 6. Scalability of our OpenMP implementation

Table 3 shows that the ratio of the time for the functions *qdct3* and *hourglass* to the total execution time on the two CMP systems is significantly decreased from 97.33% to 72.62% with increasing the number of cores. It indicates that the sequential portion of the OpenMP program increases its impact while the time for the parallel portion is reduced with increasing the number of cores. Therefore, to achieve the better performance, we need to parallelize the code as much as possible such as parallelizing the large array operation $brhs = brhs/alhs$ as we described before.

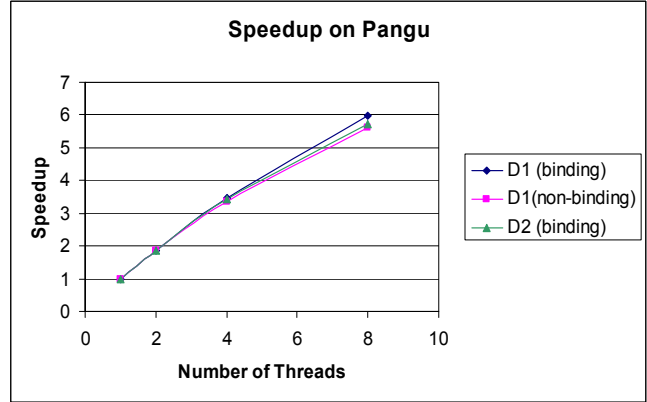


Figure 7. Scalability Comparison on Pangu

Figure 7 presents the scalability of the OpenMP implementation on Pangu. When the problem size increases from the dataset D1 to D2, using processor binding can achieve the better performance. “non-binding” means our original OpenMP implementation. Because OpenMP threads are dynamically scheduled to physical processors, this does increase system overhead. The goal of processor binding is to reduce the conflicts of chip resources on the CMP system. In our previous work [14], we found that processor binding resulted in up to 7.16% performance improvements for MPI scientific applications. Here, we use the command *pbind* to implement a batch process to bind the threads to different physical processors in order to reduce the resource contentions and system overhead from the dynamic scheduler on Pangu.

Table 3. Ratio of the time for *qdct3* and *hourglass* to the total execution time on Hydra and Pangu

| #Threads | 1 | 2 | 4 | 8 | 16 |
|----------|--------|--------|--------|--------|--------|
| Hydra | 97.33% | 93.53% | 88.23% | 81.00% | 72.62% |
| Pangu | 96.86% | 93.50% | 85.20% | 76.81% | -- |

Table 4. Performance improvement percentages for the dataset D1 on Pangu

| #Threads | 1 | 2 | 4 | 8 |
|----------|--------|-------|-------|-------|
| Binding | 0.50% | 1.25% | 2.33% | 6.04% |
| 4MB | 0.14% | 1.65% | 2.06% | 3.31% |
| 2MB | -0.14% | 1.39% | 3.95% | 2.77% |

Table 5. Performance improvement percentages for the dataset D1 on Hydra

| #Threads | 1 | 2 | 4 | 8 | 16 |
|--------------|-------|-------|-------|-------|-------|
| 64KB | 4.28% | 3.46% | 6.98% | 4.66% | 6.17% |
| 64KB+binding | -- | 4.63% | 7.05% | 5.56% | -- |

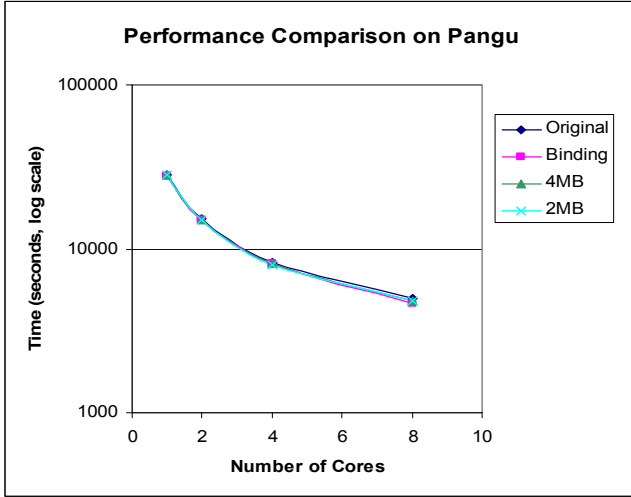


Figure 8. Performance comparison for the dataset D1 on Pangu

Figure 8 presents performance comparisons for the dataset D1 using different optimization techniques on Pangu. “Original” means the performance for our OpenMP implementation. “Binding” stands for the performance of the OpenMP program using processor binding. Pangu supports user-level large page sizes of 2MB or 4MB using the compiler option `-xpagesize=2M` or `4M`. “4MB” stands for the performance of the OpenMP program using the large page size of 4MB, and “2MB” stands for the performance of the OpenMP program using the large page size of 2MB. Although the performance looks close in Figure 8, there is still up to 6.04% performance improvement using processor binding, up to 3.31% performance improvement using the large page size of 4MB, and up to 3.95% performance improvement using the large page of 2MB shown in Table 4. Due to hardware efficiencies associated with larger pages, using large page can eliminate costly TLB misses to improve the application’s performance and throughput but at the expense of an increase in process start-up time.

Figure 9 presents performance comparisons for the dataset D1 using different optimization techniques on Hydra. “Original” means the performance for our OpenMP implementation. “64KB” stands for the performance of the OpenMP program using the large page size of 64KB. “64KB+binding” stands for the performance of the OpenMP program using the large page size of 64KB and processor binding. There is up to 6.98% performance improvement using the large page size of 64KB, and up to 7.05% performance improvement using the large page size of

64KB and processor binding on Hydra shown in Table 5. This is a big performance improvement.

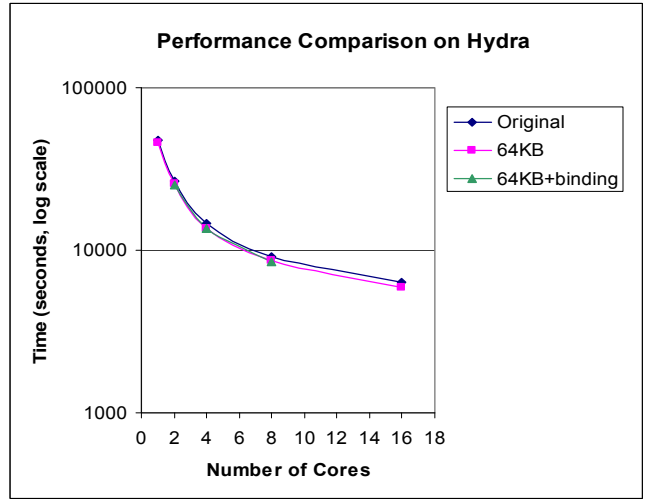


Figure 9. Performance comparison for the dataset D1 on Hydra

In brief, we evaluate the performance of our OpenMP implementation on the two CMP systems, and find that the OpenMP program has the good scalability. We apply the optimization techniques such as using the large page and processor binding to our OpenMP implementation to achieve the better performance. The optimization techniques we used do not require any code modification. It is useful to using the techniques to maximum the application performance.

6. Conclusions

In this paper, we used OpenMP to parallelize the EQdyna for modeling dynamic earthquake rupture along geometrically complex faults on the two CMP systems, IBM POWER5+ system and SUN Oteron server. The experimental results indicated that the OpenMP implementation has the accurate output results and the good scalability on the two CMP systems. Further, we applied the optimization techniques such as large page and processor binding to our OpenMP implementation to achieve up to 6.04% performance improvement on Pangu and up to 7.05% performance improvement on Hydra without any code modification. Our experimental results also indicate that OpenMP parallel programming within one CMP node can

take advantage of the globally shared address space and on-chip high inter-core bandwidth and low inter-core latency. For the future work, we plan to implement the hybrid MPI/OpenMP approach to the earthquake simulation to explore multiple levels of parallelism for large production runs on up to 10,000 processors.

Acknowledgements

The work is in part supported by the Institute for Applied Mathematics and Computational Science (IAMCS) and Department of Geology & Geophysics. The authors would like to thank the reviewers' valuable comments and the TAMU Supercomputing Facilities for the use of Hydra.

References

- [1] B. Duan and D. D. Oglesby, Heterogeneous Fault Stresses From Previous Earthquakes and the Effect on Dynamics of Parallel Strike-slip Faults, *J. Geophys. Res.*, 111, B05309, doi:10.1029/2005JB004138, 2006.
- [2] B. Duan and D. D. Oglesby, Nonuniform Prestress From Prior Earthquakes and the effect on Dynamics of Branched Fault Systems, *J. Geophys. Res.*, 112, B05308, doi:10.1029/2006JB004443, 2007.
- [3] B. Duan and S. M. Day, Inelastic Strain Distribution and Seismic Radiation From Rupture of a Fault Kink, *J. Geophys. Res.*, 113, B12311, doi:10.1029/2008JB005847, 2008.
- [4] R. A. Harris, M. Barall, et al., The SCEC/USGS Dynamic Earthquake-rupture Code Verification Exercise, *Seismol. Res. Letts.*, vol. 80, No. 1, 2009.
- [5] D. Hepkin, *Guide to Multiple Page Size Support on AIX 5L Version 5.3*, IBM, March 2006.
- [6] J. Levesque, J. Larkin, et al., *Understanding and Mitigating Multicore Performance Issues on the AMD Opteron Architecture*, LBNL-62500, March 7, 2007
- [7] H. Jin, M. Frumkin and J. Yan, *The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance*, NAS Technical Report NAS-99-011, October 1999.
- [8] K. Olukotun, L. Hammond, and J. Laudon, *Chip Multiprocessor Architecture: Techniques to Improve Throughput and Latency*, Morgan & Claypool Publishers, 2007.
- [9] OpenMP, <http://openmp.org/wp/resources/>
- [10] Ruud Van der Pas, OpenMP Tutorial and Getting OpenMP Up To Speed, *the 4th International Workshop on OpenMP*, May 2008.
- [11] Sun Studio 12 Fortran 95 compiler, *Fortran User's Guide*, <http://developers.sun.com/sunstudio/>, May 2007.
- [12] The SCEC/USGS Spontaneous Rupture Code Verification Project, <http://scedata.usc.edu/cvws>.
- [13] Texas A&M University Supercomputer Facility Hydra, <http://sc.tamu.edu/systems/hydra>.
- [14] Xingfu Wu, Valerie Taylor, Charles Lively and Sameh Sharkawi, Performance Analysis and Optimization of Parallel Scientific Applications on CMP Clusters, *Scalable Computing: Practice and Experience*, Vol. 10, No. 1, 2009.