PARALLEL COMPUTING OF NUMERICAL MANIFOLD METHOD WITH OPENMP

Qinghai Miao[1], *Min Huang*[1], *Qing Wei*[2]

1. Graduate University of Chinese Academy of Sciences
2. Kunming University of Science and Technology

ABSTRACT

This paper introduced a parallel implementation of Numerical Manifold Method based on multiprocessor platforms. First, the computing performance for a class of rock engineering problems was analyzed. For solving simultaneous equations is the most time consuming, we choose parallelized Jacobi's iterative method to speed up the computing process. The implementation of parallel Jacobi's iterative method with OpenMP was introduced. A series of experiments shows that our parallel algorithm is an effective way to improve computing performance for such class of engineering problems.

Index Terms—parallel computing, Numerical Manifold Method, Jacobi iteration, OpenMP

1. INTRODUCTION

Nowadays, parallelization has become the most important way to accelerate engineering computing and simulations. With development of High Performance Computing (HPC), a variety of parallel processors have been used in different situations. These processors can be classified into three types: CMP (Chip Multi-Processors), GPGPU (General Purpose GPU), and Heterogeneous Multiprocessor. These processors are further organized as SMP (Shared Memory Processors), MPP (Massively Parallel Processors) or DSM (Distributed Shared Memory) style to form cluster systems. Each type of processor has its own features. CMP, such as Intel Core2 Duo, Xeon and AMD Phenom, has the most market share. In fact, CMP with 2 or 4 cores has been used for nearly all laptops, desktops and workstations. GPGPU based computing, proposed by NVidia and AMD, is now a hot topic in fast developing. It requires cooperation between GPU and CPU hardware. Heterogeneous Multiprocessor, such as IBM Cell used in the Top One HPC cluster Roadrunner [1], is powerful but not easily available for common customers.

On the other hand, engineering simulations have different scales and backgrounds. In order to get best parallel efficiency, a simulation of certain scale need to run on a certain computing platform. For example, astrophysical simulations of galaxy usually need to run on giant cluster systems. But for a lot of small or middle scale simulations, running on large cluster systems results low efficiency, without expected speedup. Simulations using NMM to analyze displacement and stress in rock engineering is a good example [2]. Such simulations usually take from tens of seconds to several minutes for given numbers of steps, but required to repeat frequently with different configurations. So, utilizing workstations that has 4 or 8 cores organized in SMP style is a suitable solution.

This paper focused on the parallel implementation of Numerical Manifold Method with openMP. Section 2 introduced the engineering backgrounds and principles of NMM, as well as computing bottle necks for a series of simulations. Section 3 first introduced the Concurrency Analysis of NMM, and then emphasized on parallel implementation for solving linear algebraic equations using openMP. In section 4, we tested more than twenty examples to verify the parallel implementation and got speedup curves. Lastly in section 5, we draw conclusions and discussed the works to do in the near future.

2. NUMERICAL MANIFOLD METHOD

Numerical simulation has become the third method besides theory analysis and experiments. Numerical methods like Finite Element Method (FEM), Discrete Element Method (DEM), played important roles in design, construction and failure analysis of most of engineering projects. Numerical Manifold Method (NMM) is a new generation method put forward by Shi [3]. Before the parallel implementation, it is necessary to have an over look at NMM's engineering backgrounds, principles, and computing performances.

2.1. Engineering backgrounds

FEM, regarded as a revolutionary method of 20th century, has been widely used in almost all areas involving analysis of continuous medium. However, it is difficult to apply FEM to discontinuous problems, for example the dynamic simulation of block systems in slope or tunneling projects. On the other hand, DEM is dedicated to analysis of discontinuous medium. But within a single block, DEM can not get stress distribution with high precision as FEM.

NMM covers the gap between continuous and discontinuous analysis. NMM utilizes two types of meshes, mathematical mesh and physical mesh, to form finite cover

978-1-4244-5076-3/09/$26.00 ©2009 IEEE

Table. 1. Statistics of NMM performance Analysis based on VTune

example / step	1	2	3	4	5	6	7	8	9	10
contact	15.2	9.5	7.4	1.1	1.2	2.2	32.6	2.5	8.8	23.4
assemble	6.1	0.5	3.4	0.5	0.8	0.8	4.4	0.7	1.9	2.5
solver	66.7*	76.2	84.1	97.0	96.3	94.8	55.8	94.4	84.3	59.0

*NOTE：The solver executing time counts 66.7% of all CPU time of the whole NMM computing procedure.

systems covering the domain. Cover functions are setup on each cover, and weight functions connected each cover function according to the minimum energy principle. In such a way, with the help of Simplex Integration, NMM can deal with contact between blocks, as well as get displacement and stress distribution with high precision. Now, NMM has become one of the most promising numerical methods especially in rock engineering. Details of NMM are beyond this paper, one can refer to [3] and [4] for more information. The parallel implementation was proposed to improve performance of NMM on parallel platforms.

2.2. Diagram of NMM computing flow

A typical calculation loop of NMM is composed of three steps: contacts, assembling global stiffness matrix, and solving simultaneous equations. Contacts calculate the collision detection between blocks in a system, and also the forces generated by the collisions. Next, matrix for initial stress, point loading, body loading, fixed point loading, inertia force, friction force and contact springs are added to form the global stiffness matrix, which is the coefficient matrix of the simultaneous equations. By solving the simultaneous equations, variables such as displacements are updated, thus, this loop finished and a new loop will begin.

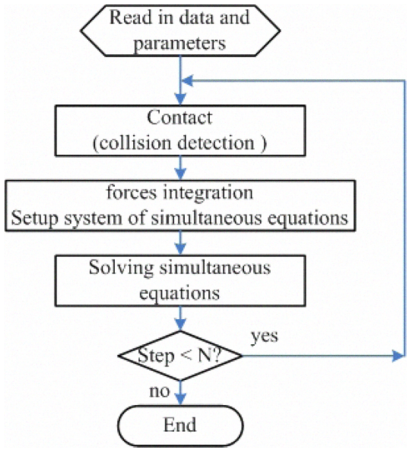

Figure. 1. NMM Computing Diagram

Diagram of a NMM loop is shown in Figure 1.

2.3. Performance analysis of NMM

In order to find out computing bottle beck of NMM, we took performance analysis based on 10 examples. These examples are small or middle scale problems including joint computations, block computations, slope sliding and failure of structures. Some of these examples will be introduced with graphics in section 4. We run each example with intel's performance analysis tool – Vtune [5]. Time accounts of the three steps of one NMM computing loop are listed in Table 1 above.

From the table, we can see that the third step – solving simultaneous equations – account about 60% - 95% of all computing time. So, this computing bottle neck is the place we take parallel implementation.

3. PARALLELIZATION WITH OPENMP

To implement a parallelized solver of simultaneous equations, we must choose a proper method first, which guarantees the solution is correct. After theoretical verification, we choose a proper parallel programming tool and realize the parallelization at a certain granularity. These two steps, together with memory allocation, are introduced as follows.

3.1. Choosing solver of simultaneous equations

There are two types of solvers for linear equations system: direct method and iteration method. Though direct methods, for example Gaussian elimination method, can have exact solution, they usually need more multiplication and division operations, as well as mass memory consumptions. On the contrary, iteration methods need less algebraic operations, less memory, but only have approximate solutions. Further, there are three iteration methods: Jacobi (Simple iteration), Gaussian-Seidel, and SOR. SOR is the widely used method on serial processors, which has the best performance if proper relax factor has been chosen. But there are data dependences between variables in both the same iteration and two succeed iterations. So, it is hard to take parallel implementations of SOR in fine granularity. Gaussian-Seidel method has the same problems as SOR. Jacobi iteration method was usually used as an example to explain the principles of iteration methods in textbook. It is too slow on serial processors that it has never been applied to practices. But things changed as multiprocessors become popular. There are no dependences between each variable

all through Jacobi iterations, which is naturally suitable for parallelization.

One thing we have to pay attention is that, when using iteration methods, convergence must be guaranteed. Thanks to the inertia control mechanism, we need not to worry about this. In the step of forces integration, NMM introduced in a mass matrix, which related to step time. In detail, the mass matrix is a diagonal matrix, whose elements are in inverse ratio of square of time step. If the iteration cannot converge within certain of steps, we reduce the time step, which will enlarge the elements of mass matrix and thus make the whole stiffness matrix diagonal dominance. According to [6], a diagonal dominant matrix is always to be convergent.

3.2. parallel Jacobi iteration with openMP

There are several tool kits for parallel programming, such as MPI, openMP, intel TBB [7]. MPI has advantages on parallel computing with big granularity, which usually runs on cluster systems. OpenMP is an API that supports parallel programming for parallelization with small granularity, especially on SMP platforms. TBB is based on C++ Standard Template Library (STL) and designed for Object Oriented programming. For our mission is to parallelize the equation solver running on SMP platforms, and our program is written in plain C for efficiency, openMP is the most suitable tool kit [8].

The fundamental idea is, based on Jacobi's iteration, parallelizing the loop of each variable into a fork-join style. In practice, we use the "parallel for" directive, with shared or private specified for each variable. We adopted the default static schedule with block size M/N, which M is the matrix order and N is the number of CPU cores. At last, we took '+' as reduction for computing average iteration err. When the average error becomes less than the threshold, say 10-6, the iteration exits. Pseudo-code is shown in Figure. 2 as follows:

```
#pragma omp parallel for private() reduction (+: e)
 for each row of coefficient matrix
    x(m+1) = D-1  (A-D) x(m) +D-1F
    e = abs(a-b)
```

Figure. 2. Iteration Format

4. EXPERIMENTS

We took 15 examples to test the performance of parallel algorithm. These examples are problems from engineering practices, including block and joint computations, slope sliding, and failure of structures, etc. Some of them are illustrated by graphics, shown in Figure. 3, 4, and 5. Two aspects were analyzed: convergence and speedup.

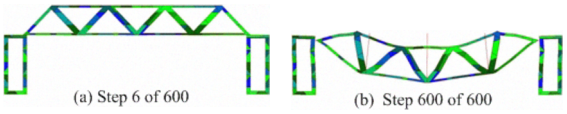

Figure. 3. Structure failure of supported beam

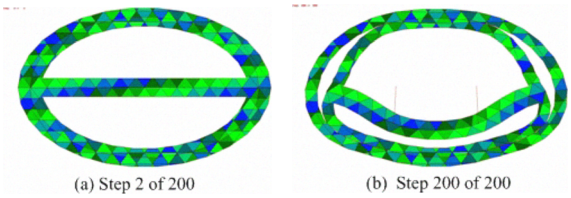

Figure. 4. Structure failure of tunnel

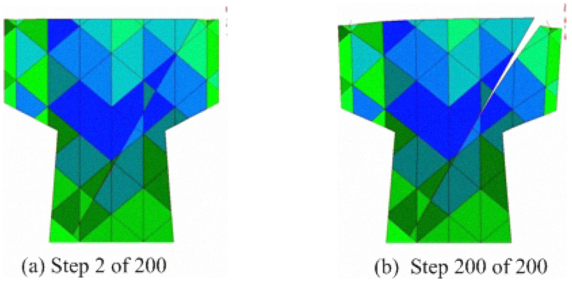

Figure. 5. Single joint of strut

4.1. Convergence speed

Convergence speed is a key factor when solving simultaneous equations using iteration methods. A faster convergence speed always results in good performance. For the same example with same parameters, for an example as shown in Fig. 4, convergence speed at the same step (step 2) is recorded for both serial and our parallel algorithms. Curves in Fig. 6 shows the comparison, from which we can see that parallel Jacobi method has better performance than SOR with relax factor 1.9. In this example, serial SOR took 600 steps to reach an error at 10^{-5}, while parallel Jacobi method only took about 400 steps to reach an error at 10^{-6}, which satisfied the precision requirement.

4.2. Speedup

Speedup is the most important factor to evaluate the performance of parallel algorithms. Speedup is defined as the computing time ratio of optimized serial algorithm to parallel algorithm. We ran each example on three different SMP platforms and obtained three speedup curves respectively. On Core 2 Duo platform, we got an average speedup about 1.6; on Xeon 5420 (quad cores) and Xeon 5482 (double quad cores) platforms about 2.5, as shown in Figure 7.

For these test examples, the computing scales (the order of coefficients matrix) range from 194 to 2193. We can see from the curves in Figure 7 that, the lager the problems

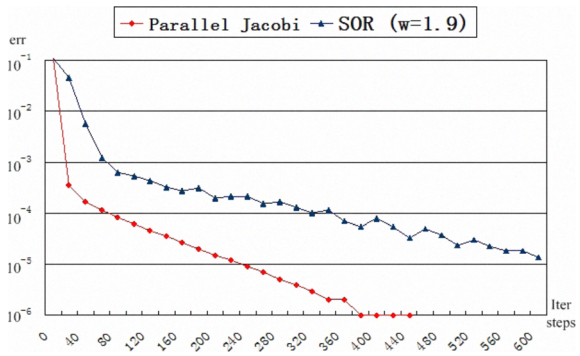

Figure. 6. Iteration Format

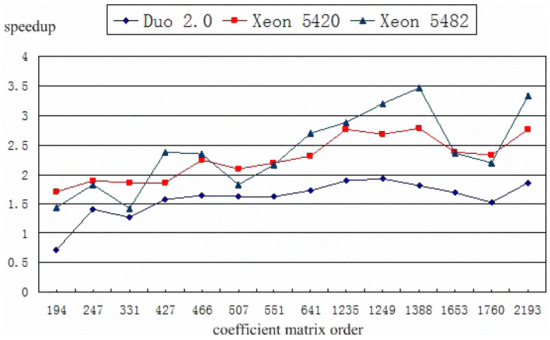

Figure. 7. Iteration Format

scale is, the better speedup we can obtained. But when running problems of small scale on platforms with few processors, for example running 194 on Core 2 Duo platform, a speedup is less than 1, i.e., the parallel algorithm has worse performance than serial ones. The reason is that thread scheduling overwhelms the speedup from parallelization.

We can also see from Figure 7 that, the speedups on Xeon 5420 (quad cores) are nearly the double of those on Core 2 Duo platform, but speedups on Xeon 5482 (double quad cores) have no advantages than on Xeon 5420 (quad cores) platforms. The reason comes from the difference on architecture. The Xeon 5420 (quad cores) is a contact coupled SMP with four cores sharing the same cache within one chip. On the other hand, Xeon 5482 platform is a loosely coupled SMP, with 8 cores on two separate chips connected by off chip memory. The access speed of off chip memory is much lower than on chip caches, which drops down the speedup on Xeon 5482 platform.

5. CONCLUSIONS

For a class of numerical computing problems in rock engineering, we implemented a parallel algorithm based on multiprocessors. The work laid emphasis on solving simultaneous equations using Jacobi iteration method, which was regarded as poor performance on serial platforms. The experiment results show that, the parallel Jacobi iteration method speedups the computing time of a type of small or middle scale problems. For multiprocessor are popular platforms, it is an effective way to solve simultaneous equations using parallel Jacobi iterations. Not only for rock engineering, but all numerical computing algorithms, like FEM, DEM, can benefit from the work introduced in this paper.

Further work leads to two aspects: parallel collision detection and GPU based parallelization. First, for a part of problems with more block number, more computing time will needed to do collision detection and responds. The piece-by-piece algorithm for collision detection has the time complexity of $O(n^2)$, in which n is the block number. We planed to substitute this poor method with parallel algorithms based on BSP, Oct-tree. A good speed up can be obtained theoretically. Second is to implement NMM on GPU platforms. As GPUs usually have much more processors than multi-core CPUs, their parallelization attracts our attentions. What's more, software toolsets like CUDA of nVidia make it easier to utilizing GPU cores.

ACKNOWLEDGEMENTS

The authors would thank Dr. Gen-hua Shi, who is the inventor of Numerical Manifold Method, for his instructions and kindly help. This work was supported by a Dean's Fund of Graduate University of Chinese Academy of Sciences (class A, 085101FM803).

REFERENCES

[1] http://www.top500.org/.2009.6.
[2] Weiyuan Zhou, Qiang Yang, "Numerical Computational Methods for Rock Mechanics", China power press, Beijing, 2005, pp. 364-378.
[3] Shi Genhua. "Manifold method". In Proc. of the First Int. Forum on DDA and Simulations of Discontinuous Media. Berkeley, California, USA; 1996, pp. 52-204.
[4] Pei Juemin, "Numerical Manifold Method and Discontinuous Deformation Analysis", Chinese Journal of Rock Mechanics and Engineering, Vol. 16, No. 3, 1997, pp.279-292.
[5] The VTune™ Performance Analyzer Reader/Writer API (TBRW) User's Guide, http://www.intel.com/cd/software/products/apac/zho/cluster/tanalyzer/343064.htm
[6] Tongfu Lv,etc. "Numerical Computational Methods", Tsinghua University Press, Beijing, 2008.
[7] Intel® Threading Building Blocks Tutorial, version 1.11, http://www.intel.com/cd/software/products/apac/zho/319508.htm.
[8] http://openmp.org/wp/. 2009.7