2010 Ninth International Symposium on Distributed Computing and Applications to Business, Engineering and Science

A Parallel PCG Solver Based on OpenMP for Three-dimensional Heat Equation

Dandan Li, Tangpei Cheng, Qun Wang*

School of Information Engineering

China University of Geosciences (Beijing)
P.R.China
xingyunnancy@sina.com

***Abstract*—**Heat equation has been widely used in engineering, such as numerical simulation of groundwater flow. The parallelism of heat equation is an important means of accelerating the simulation process. In order to solve the three-dimensional heat equation problem more rapidly, the OpenMP was adopted to parallelize the preconditioned conjugate gradient (PCG) algorithm in this paper. A numerical experiment on the three-dimensional heat equation model was carried out on a computer with four cores. Based on the test results, it is found that the execution time of the original serial PCG program is about 1.61 to 2.53 times of the parallel PCG program executed with different number of threads. The experiment results also demonstrate that using OpenMP to parallelize the PCG algorithm is an effective way for solving the three-dimensional heat equation.

Keywords- three-dimensional heat equation; precondintioned conjugate gradient; compiler directives; OpenMP

I. INTRODUCTION

Heat equation is one of the most important mathematic equations, which is widely applied in engineering application. However, traditional serial programs take large computational efforts when they are applied to solve the heat equation problems with massive grids or three-dimensional. For instance, adopting traditional serial programs are quite difficult to solve large-scale three-dimensional ground water flow models. Thus, the parallel solution of the three-dimensional heat equation is extremely important. Meanwhile, preconditioned iterative methods and parallel computing methods have been proved to be two efficient ways to shorten execution time. For this reason, considerable effort is being expanded into parallel computing and preconditioned iterative methods for heat equation[1-7]. Although much research has been undertaken on increasing the stability and convergent rate of iterative methods, less work has focused on adopting high performance parallelization toolkits to parallelize the preconditioned iterative methods for solving the three-dimensional heat equation.

Nowadays, OpenMP, one of the most well-known application programming interfaces is increasingly adopted as a high performance parallelization toolkit. The OpenMP can deliver good parallel performance for small number of threads. And with the OpenMP compiler directives, the parallelization is divided among multiple threads without changing the rest of the serial program. Thus, the main goal of this paper is to present the OpenMP parallelization toolkit to parallelize the preconditioned conjugate gradient (PCG) algorithm. Based on the three-dimensional heat equation model, experiment results show that the execution time for solving the large-scale heat equation is remarkably shortened by applying parallel PCG algorithm.

II. THREE-DIMENSIONAL HEAT EQUATION MODEL

In general, the three-dimensional heat equation can be expressed as

$$\begin{cases} \dfrac{\partial^2 u}{\partial x^2}+\dfrac{\partial^2 u}{\partial y^2}+\dfrac{\partial^2 u}{\partial z^2}=\dfrac{\partial u}{\partial t}, & (x,y,z)\in\Omega, t>0 \\ u=u^0, & t=0 \\ u=g, & (x,y,z)\in\partial\Omega, t\geq 0 \end{cases}$$

on the domain $\Omega=(0,M)\times(0,N)\times(0,L)$.

Finite difference method is used for discretizing the three-dimensional heat equation. For space discretization, we apply the seven-point stencil finite difference method. For time discretization, the heat equation is handled by the backward Euler method which is a fully implicit method. Consequently, we obtain a sparse linear algebraic system $Ax=b$, in which A is symmetric positive definite. For details about the deduction, readers can refer to our previous work[8].

III. OPENMP MULTIPLE THREADS PROGRAMS

OpenMP is a standard and portable application programming interface (API) for writing multiple threads programs on a shared memory computer. It is comprised of three primary API components: compiler directives, runtime library routines and environment variables. OpenMP is supported by Fortran and C/C++ compilers and is available for a variety of platforms, from PCs to high performance computers[9].

As described in Fig.1, OpenMP provides the fork-and-join execution model. At the beginning of a program execution, only a single thread is active. This thread executes sequentially unless a parallel construct is found. At the moment, the thread creates a team of threads and it becomes the master thread. During the parallel region, the master

*Corresponding author Tel:+8601082322116
Email address: qunw@cugb.edu.cn

978-0-7695-4110-5/10 $26.00 © 2010 IEEE
DOI 10.1109/DCABES.2010.40

thread and derived threads will work together. Upon completion of the parallel region, those derived threads will quit or hang up, and only the master thread continues, which is called a join.

A vital advantage of the OpenMP is the parallelization can be done incrementally, that is, the majority of the serial code is not changed and the user only needs to identify and parallelize just the most time-consuming parts of the code, which are usually loops[9]. This feature is very helpful for parallelizing the PCG algorithm[10]. As the OpenMP supports the incremental parallelization, it has been widely adopted in the scientific computing community.

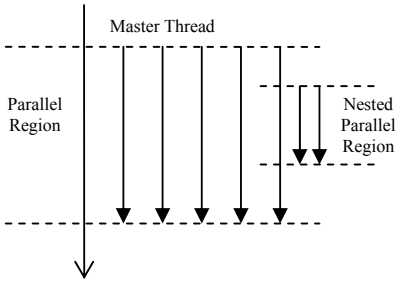

Figure 1. Fork-Join Model in OpenMP

IV. PARALLELIZATION OF THE PCG ALGORITHM

As stated in section Ⅱ, the heat equation is discretized to a linear algebraic system $Ax = b$, where A is a symmetric positive definite matrix. For solving the positive definite linear algebraic system $Ax = b$, the conjugate gradient (CG) method is an effective iterative method[11]. Meanwhile, both the roubustness and efficiency of the CG can be improved by empolying preconditoning techniques. Thus, the conjugate gradient combined with a preconditoner has proved to be one of the most efficient ways among the simple iterative methods[11].

A. Preconditioned Conjugate Gradient Algorithm

The main operation for PCG is loop iterations. Specific calculation steps of the PCG are as follows.

STEP1. Choose an arbitrary x^0, set $r^0 = b - Ax^0$, $p^0 = z^0 = P^{-1}r^0$, where P is a preconditioner. In our study, the P is obtained by adopting Cholesky factorization method.

STEP 2. Iterate *for* i ----0,1,2,.., until convergence

$$w^k = Ap^k \quad (1)$$

$$\alpha_k = \frac{(z^k, r^k)}{(p^k, w^k)} \quad (2)$$

$$x^{k+1} = x^k + \alpha_k p^k \quad (3)$$

$$r^{k+1} = r^k - \alpha_k w^k \quad (4)$$

$$Pz^{k+1} = r^{k+1k} \quad (5)$$

If $(r^{i+1}, r^{i+1}) < \varepsilon$, stop

$$\beta_{k+1} = \frac{(z^{k+1}, r^{k+1})}{(z^k, r^k)} \quad (6)$$

$$p^{k+1} = z^{k+1} + \beta_{k+1} p \quad (7)$$

B. Parallelization of Serial PCG Program

The time of solving the linear algebraic system $Ax = b$ with PCG algothrim occupies most of execution time. Hence, in this paper, our parallel work mainly focus on parallelizing PCG algorithm.

By analyzing the PCG algorithm, the most time-consuming are three parts: matrix-vector multiplication, vector inner product and solving preconditoned equations. Hence, the OpenMP is applied to parallelize the three parts in order to improve the computational efficiency.

1) Parallelization of Matrix-vector Multiplication

```
# pragmaomp parallel for private(...) firstprivate(...)
for i := 0 → n
 for j := Arow[i] → Arow[i + 1]
    calculating col;
    calculating Aelement ;
    Ax[i]+ = Aelement * x[col];
 end for
end for
```

Figure 2. The Code of Matrix-vector Multiplication

In order to save memory overhead, we adopt the compressed sparse row (CSR) format to store the matrix. In this CSR format, we need to create three arrays. The first array stores the values of all nonzero elements of the matrix. The second array stores the column indexes of the elements in the first array. The third array stores the locations in the first array that start a row. In the block code shown in Fig.2, the array *Arow* is the third array. The value of n is the dimensions of the vector. The variable Ax is an array which is used to store the results of multiplying matrix by vector.

As shown in Fig.2, there are two level loops in the code of matrix-vector multiplication. In order to improve the computational efficiency, simply direct the compiler to execute the iterations of the loop indexed by i. However, extra attention should be paid to the variables. All variables except the loop index variables are shared by default. That makes it easy for threads to communicate with each other, but it also cause data race problems. We add the private() clause to OpenMP compiler directives for avoiding problems of data race. Besides, we adopt the firstprivate() clause to state those temporary private variables whose values are initialized by using their original values in the master thread.

2) Parallelization of Vector Inner Product

```
# pragmaomp parallel for reduction (…)
for i := 0 → n
    answer+ = x[i] * y[i];
end for
```

Figure 3. The Code of Vector Inner Product

The code of vector inner product is shown in Fig.3. In the block code, array x and y are used to indicate vectors. The value of n is the dimensions of the two vectors. The result of computing vector inner product is stored in the variable *answer*. To parallelize the code, we use the OpenMP compiler directives to parallelize the iterations of loop. When parallelizing the code, we encounter a problem that the variable *answer* must be both private and shared for avoiding data race and ensuring the proper implementation of multiple threads. This problem can be solved by employing the OpenMP reduction() clause to declare the variable *answer*. The OpenMP reduction() clause creates a private copy of the variable *answer* for each thread. At the end of the reduction, the variable *answer* is applied to all private copies of the shared variable, and the final result is written to the global shared variable.

3) Parallelization of solving preconditioned equations

```
# pragmaomp parallel for private(...)
for i := 0 → n
    dpotrs();
end for
```

Figure 4. The Code of Solving Preconditioned Equations

In the original serial program, we adopt the Cholesky factorization method to construct the preconditioner. In the block code described in Fig.4, the value of n is the number of the equations. And the $dpotrs()$ is a function which can solve the linear algebraic system $Ax = b$ with a symmetric positive definite matrix A using the Cholesky factorization. Obviously, the main time-consuming of the code is iterations of the loop indexed by k. Hence, for the sake of shortening the execution time of solving precondtioned equations, we resort to the OpenMP complier directives to execute interations of the loop in parallel. Similarly, we should pay attention to variables in order to avoid the data race problems. We employ private() clause to state those variables which occure in the k loop. Other variables are shared except the loop index variable by default.

V. NUMERICAL EXPERIMENT

In this paper, we carried out a numerical experiment on the four cores computer with 8 Gb memory, 4 Intel(R) Xeon(R) 5110 1.6GHz cores and Windows 2003 Operating System. The experiment with discretization of 200*200*120 spatial grids by finite difference method focused on investigating the execution time of the parallel program by using OpenMP to parallelize the PCG algorithm. Part of the test results are shown in TABLE Ⅰ.

TABLE I. EXECUTION TIME OF THE PARALLEL PROGRAM WITH DIFFERENT NUMBER OF THREADS

the number of threads	1	2	3	4
execution time(s)	57.05	35.44	28.17	24.24
speedup	1.00	1.61	2.23	2.53
efficiency(%)	100%	80.5	74.3	63.3

According to the statistics provided by TABLE Ⅰ, it is easy to see that the parallel PCG can shorten the execution time for solving the large-scale three-dimensional heat equation problem. With the number of threads increases, the speedup increases while the execution time and the efficiency decline. Because the speedup is defined as the ratio of the serial PCG program execution time and the parallel PCG program execution time, the speedup increases with the number of threads. The efficiency declination mainly due to the system overhead brought by making the PCG paralleled increases with the number of threads. The system overhead involves the overhead of synchronization between threads, data race problems, creation threads as well as hang up threads.

TABLE II. THE SPEEDUP OF PARALLELIZING DIFFERENT PARTS OF PCG

the number of threads	1	2	3	4
Matrix-vector multiplication	1	1.98	2.95	3.60
Vector inner product	1	1.95	2.53	2.91
Solving preconditioned equations	1	1.31	1.54	1.72

The second line of data in the TABLE Ⅱ show the speedup of parallelizing matrix vector multiplication. From the test results it follows that the measuring speedup increases with the number of threads. Moreover, the measuring speedup is very close to the theoretical speedup. The parallelization of matrix-vector multiplication can achieve a desirable speedup mainly due to itself has a high level parallelism. And the reason measuring speedup can not reach the theoretical value is that making the code paralleled also brings some system overhead like the overhead of copying, creation threads and hang up threads. The test results indicate that the parallelization of matrix-vector multiplication is very effective.

The third line of data in the TABLE Ⅱ describe the speedup obtained by parallelizing vector inner product. According to the statistics provided by TABLE Ⅱ, it can be seen that the measuring speedup of parallelizing vector inner product increases with the number of threads. However, the implementation of parallelizing vector inner product does not achieve a desirable scalability of the speedup. One reason for the results is the data race problems. When the code of vector inner product is executed with multiple threads, the data race problems can be caused. As the number of threads increases, the data race problems occur more frequently. Another reason is that the reduction operation which causes the overhead of synchronization between threads. The synchronization overhead also increases with the number of threads. Besides, some system overhead like overhead of creation threads and hang up

threads could also influence the scalability of the measuring speedup. The above mentioned factors has led to this performance degradation.

The last line of data in the TABLE Ⅱ portray the measuring speedup achieved by parallelizing the sloving preconditioned equations. Although the measuring speedup increases with the number of threads, the performance of measuring speedup is deviation from the theoretical speedup. One reason for affecting the peformance of parallelizing the sloving preconditioned equations is the problems of data race. When the code of solving preconditioned equations is executed in parallel, it is easy to produce data race. And with the number of threads increases, the data race problems occur more frequently. Another reason is that making the code of solving preconditioned equations paralleled brings a lot of system overhead, such as the overhead of copying, creation threads and hang up threads. The system overhead could influence the parallel peformance.

VI. CONCLUSION

Preconditioned iterative methods and parallel computing methods are two efficient ways for accelerating the simulation process of the heat equation. This paper provides an approach using OpenMP to parallelize the PCG algorithm for solving the large-scale three-dimensional heat equation on a multi-core computer. The parallel approach produces an impressive reduction of the execution time and this approach achieves great improvement in computational efficiency. Based on the experimental results, it is evident to conclude that the parallel PCG solver based on the OpenMP parallelization toolkit is suitable for solving three-dimensional heat equation problems with massive grids.

REFERENCES

[1] Jacques-Louis Lions, Yvon Maday, Gabriel Turinici, “A "parareal" in time discretization of PDE's”, Comptes Rendus de l’Académie des Sciences - Series I - Mathematics, 332(7), pp. 661-668, 2001.

[2] S. Contassot-Vivier, R. Couturier, C. Denis, F. Je´ze´quel, “Efficiently solving large sparse linear systems on a distributed and heterogeneous grid by using the multisplitting-direct method”, Fourth International Workshop on Parallel Matrix Algorithms and Applications, PMAA’06, pp. 21-22, 2006.

[3] P.R. Amestoy, I.S. Duff, S. Pralet, C. Vo¨mel, “Adapting a parallel sparse direct solver to architectures with clusters of SMPs”, Parallel Computing 29 (11-12), pp. 1645-1668, 2003.

[4] Hasan Dağ, “An approximate inverse preconditioner and its implementation for conjugate gradient method”, Parallel Computing, vol. 33, pp. 83-91, March 2007.

[5] Torsten Hoefler, Peter Gottschling, Andrew Lumsdaine, Wolfgang Rehm, “Optimizing a conjugate gradient solver with non-blocking collective operations”, Parallel Computing, vol. 33, pp. 624-633, September 2007.

[6] V. Hernandez, J.E. Roman, A. Tomas, “Parallel Arnoldi eigensolvers with enhanced scalability via global communications rearrangement”, Parallel Computing, vol. 33 , pp. 521-540, August 2007.

[7] Zeyao Mo, Xiaowen Xu, “Relaxed RS0 or CLJP coarsening strategy for parallel AMG”, Parallel Computing, vol. 33, pp. 174-185, April 2007.

[8] Tangpei Cheng, Qun Wang, “Parallel-Computing Strategy for Large-scale Heat Equation based on PETSC”, Computer Science, vol. 36, pp. 160-164, 2009.

[9] M.T.F Cunha, J.C.F. Telles, A.L.G.A. Coutinho and J. Panetta, “On the parallelization of boundary element codes using standard and portable libraries,”

[10] Yanhui Dong and Guoming Li, “A Parallel PCG Solver for MODFLOW,” GROUND WATER, vol. 47, pp. 845-850, November-December 2009.

[11] ARANY, “The Preconditioned Conjugate Gradient Method with Incomplete Factorization Preconditioners,” Computers Math. Applic., vol. 31, pp. 1-5, 1996.