OpenMP-Based PCG Solver for Three-Dimensional Heat Equation

Dandan Li and Qun Wang

School of Information Engineering, China University of Geosciences (Beijing), Beijing 100083, P.R. China

Received: October 14, 2011 / Accepted: November 01, 2011 / Published: December 25, 2011.

Abstract: As one of the most important mathematics-physics equations, heat equation has been widely used in engineering area and computing science research. Large-scale heat problems are difficult to solve due to computational intractability. The parallelization of heat equation is available to improve the simulation model efficiency. In order to solve the three-dimensional heat problems 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 1.71 to 2.81 times of the parallel PCG program executed with different number of threads. The experiment results also demonstrate the available performance of the parallel PCG algorithm based on OpenMP in terms of solution quality and computational performance.

Key words: Three-dimensional heat equation, preconditioned conjugate gradient, compiler directives, OpenMP.

1. Introduction

As one of the most important mathematics-physics equations, heat equation has been widely used in engineering area and computing science research. Since traditional serial method is limited by its ability to make use of computer resources, it usually takes large computational efforts to solve the three-dimensional heat problems with massive grids. Thus, parallel technology is being increasingly used for solving heat problems. Meanwhile, preconditioned iterative methods and parallel computing methods have been proved to be two efficient ways to reduce 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.

With the development of multi-core technology in the past several years, the interest in OpenMP has increased significantly. The OpenMP can deliver good parallel performance for small number of threads. Thus, the objective of this research is to present the OpenMP parallelization toolkit to parallelize the preconditioned conjugate gradient (PCG) algorithm. Based on the three-dimensional heat equation model, we also present and discuss the performance of parallel PCG algorithm on a multi-core computer. The paper is organized as follows: Section 2 introduces the numerical simulation model; section 3 introduces the OpenMP parallel programming; section 4 introduces the parallel method based on OpenMP; section 5 presents results and discussions; section 6 gives conclusions and presents future work.
2. Numerical Simulation Model

The three-dimensional heat equation on the domain \( \Omega = (0, M) \times (0, N) \times (0, L) \) is often represented by the following equations:

\[
\begin{align*}
\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} + \frac{\partial u}{\partial t} &= 0, \quad (x, y, z) \in \Omega, t > 0 \\
u &= u^0, \quad t = 0 \\
u &= g, \quad (x, y, z) \in \partial \Omega, t \geq 0
\end{align*}
\]

where the function \( u^0 = u^0(x, y, z) \) and \( g = g(x, y, z, t) \) are known. They are defined in the internal and boundary of the domain \( \Omega \), respectively.

Traditionally, numerical methods provide an approximate transient solution to heat equation. In this study, the continuous domain is discretized into a set of small rectangular blocks through the finite difference method. The blocks have dimensions \( \Delta x, \Delta y \) and \( \Delta z \), where \( \Delta z \) is the thickness of the layer, and \( \Delta x \) and \( \Delta y \) are block lengths in the \( x-y \) coordinate directions as shown in Fig. 1. And the standard Cartesian mesh is used for the finite difference approximation. For space discretization, the heat equation is handled by 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. This approach results in a sparse linear algebraic system \( Ax = b \), in which \( A \) is symmetric positive definite. Detailed information on how the linear algebraic system is formed from the discretization of heat equation can be found in our previous work [8].

3. OpenMP Parallel Programming

The rapid and widespread acceptance of multi-core architectures has created a pressing demand for adopting an efficient way to program on these systems. The OpenMP is an application program interface (API) that supports multi-platform shared-memory parallel programming in C, C++ and FORTRAN on much architectures [9].

OpenMP, based on compiler directives and a set of supporting library calls, is a portable approach for parallel programming on shared memory systems. The well-known advantage of OpenMP is its global view of application memory address space that allows relatively fast development of parallel applications. The directive based approach makes it possible to write sequentially consistent code for easier maintenance. An advantageous over MPI in the parallel code development cycle is that OpenMP supports an incremental parallelism. That means the majority of the serial code is not changed and the user only needs to identify and parallelize just the most time consuming operations of the code, which are usually loop [9]. The operations such as matrix-vector multiplication and vector inner product in PCG easily fit in the OpenMP framework [10]. The following section will address how OpenMP is used to facilitate the most time-consuming loop.

4. Parallel Method Based on OpenMP

As stated in section 2, 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 robustness
and efficiency of the CG can be improved by employing preconditioning techniques. Thus, the conjugate gradient combined with a preconditioner has proved to be one of the most efficient ways among the simple iterative methods [11].

4.1 Preconditioned Conjugate Gradient Algorithm

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

Step 1: 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, \( P \) is obtained by adopting Cholesky factorization method.

Step 2: Iterate \( i = 0, 1, 2, \ldots \) until convergence

\[
\begin{align*}
\mathbf{w}^i & = Ap^i \\
\alpha_i & = \frac{(r^i, r^i)}{(p^i, w^i)} \\
x^{i+1} & = x^i + \alpha_i p^i \\
r^{i+1} & = r^i - \alpha_i w^i \\
\beta_{k+1} & = \frac{(r^{k+1}, r^{k+1})}{(r^k, r^k)} \\
p^{k+1} & = z^{k+1} + \beta_{k+1} p^k
\end{align*}
\]

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

4.2 Parallelization of Serial PCG Program

The first step of our parallelization process is to find the most time consuming operations and then use OpenMP to parallelize them incrementally. After compiling the serial PCG code on our computer, we got the following results as shown in Fig. 2, where digit denotes the percentage of the execution time that the operation consumes.

From Fig. 2, it is easy to see that the most time consuming operations are part 4, part 1 and part 2. The three parts denote solving preconditioned equations, matrix-vector multiplication and vector inner product, respectively. Hence, the OpenMP is applied to parallelize the three parts in order to improve the computational efficiency.

4.2.1 Parallelization of Matrix-Vector Multiplication

In the original PCG code, we adopt the compressed sparse row (CSR) format to store the matrix in order to save memory overhead. 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. As shown in Code segment 1, the array \( A\text{row} \) 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.

```c
#pragma omp parallel for private(...) firstprivate(...) for i := 0 -> n
for j := Arow[i] -> Arow[i + 1]
calculating col;
calculating Aelement;
A[i] += Aelement * x[col];
end for
end for
```

**Fig. 2** The execution time percentage of each part in serial PCG.
computational efficiency, simply direct the compiler to execute the iterations of the loop indexed by $i$. However, we have to set the correct scope for each of the variables in the loop. 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 causes 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.

4.2.2 Parallelization of Vector Inner Product

The code of vector inner product is shown in Code segment 2. 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 $answer$.

```
#pragma omp parallel for reduction (....)
for $i := 0 \to n$
    $answer += x[i] \times y[i];$
end for
```

**Code segment 2:** The code of vector inner product.

To parallelize the code, we use the OpenMP compiler directives “parallel for” 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.

4.2.3 Parallelization 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 Code segment 3, 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.

```
#pragma omp parallel for private(...)
for $i := 0 \to n$
    dpotrs();
end for
```

**Code segment 3:** The code of solving preconditioned equations.

Obviously, the main time consuming operation of the code is iterations of the loop indexed by $i$. Hence, for the sake of reducing the execution time of solving preconditioned equations, we add the OpenMP compiler directives “parallel for” to execute iterations of the loop in parallel. Similarly, we should set the correct scope for each of the variables in the loop in order to avoid the data race problems. We employ `private()` clause to state those variables which occur in the $k$ loop. Other variables are shared except the loop index variable by default.

5. Numerical Experiments and Discussions

In this paper, we carried out a numerical experiment on the four cores computer with 8 Gb memory, 4 Intel(R) Xeon(R) 5110 1.6 GHz cores and Windows 2003 Operating System.

5.1 Experiment Results

The experiment was executed with discretization of $200 \times 200 \times 120$ spatial grids by finite difference method. We adopted CG preconditioned with block Jacobi method. The convergence criterion used for iterative linear solver is based on the l2-norm of the residual. The stopping criteria used for linear solver is if $\|r_k\| < 10^{-8}$ at iteration $k$ where $r_k = b - Ax_k$, $r_k$ and $b$ denote the residual and the right-hand side, respectively. The numerical experiment was focused on investigating the performance of the parallel program by using OpenMP to parallelize the PCG
algorithm. The test results are shown in Tables 1-2.

5.2 Performance Discussions

According to the statistics provided by Table 1, it is easy to see that the parallel PCG can reduce the execution time for solving the large-scale three-dimensional heat problem. And the speedup is a linear increasing function of the number of the threads, while the efficiency is a linear decreasing function of the number of the threads. This phenomenon can be explained by the definition of speedup and efficiency. Speedup is defined as \( Sp = T_1 / T_p \), where \( T_1 \) is the wall-clock time of solving one problem on one processor and \( T_p \) is the wall-clock time for solving problem on \( p \) processors. Efficiency is defined as \( E_p = Sp / P \).

The second line of data in Table 2 shows 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 have a high level parallelism. Because of overhead involved in starting up OpenMP library and threads, memory coherence maintenance and memory bandwidth limitation, the measuring speedup cannot reach the theoretical value. However, the test results indicate that the parallelization of matrix-vector multiplication is very effective.

Table 1  Execution time of the parallel program with different number of threads.

<table>
<thead>
<tr>
<th>The number of threads</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution time (s)</td>
<td>57.05</td>
<td>33.36</td>
<td>24.42</td>
<td>20.31</td>
</tr>
<tr>
<td>Speedup</td>
<td>1.00</td>
<td>1.71</td>
<td>2.34</td>
<td>2.81</td>
</tr>
<tr>
<td>Efficiency (%)</td>
<td>100</td>
<td>85.6</td>
<td>78.0</td>
<td>70.3</td>
</tr>
</tbody>
</table>

Table 2  Speedup of parallelizing different parts of PCG.

<table>
<thead>
<tr>
<th>The number of threads</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix-vector multiplication</td>
<td>1.98</td>
<td>2.95</td>
<td>3.60</td>
<td></td>
</tr>
<tr>
<td>Vector inner product</td>
<td>1.95</td>
<td>2.53</td>
<td>2.91</td>
<td></td>
</tr>
<tr>
<td>Solving preconditioned equations</td>
<td>1.31</td>
<td>1.54</td>
<td>1.72</td>
<td></td>
</tr>
</tbody>
</table>

The third line of data in Table 2 describes the speedup obtained by parallelizing vector inner product. According to the statistics provided by Table 2, 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 waiting for synchronization among different cores, called synchronization overhead. 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 have led to this performance degradation.

The last line of data in Table 2 portrays the measuring speedup achieved by parallelizing the solving 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 performance of parallelizing the solving 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 performance.

6. Conclusions

Preconditioned iterative methods and parallel computing methods are two efficient ways for accelerating the simulation process of the heat equation.
This paper provides an efficient parallel PCG algorithm based on OpenMP for solving the large-scale three-dimensional heat equation on a multi-core computer. This parallel PCG solver provides an accurate simulation of heat equation. In addition, 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.

Nowadays, NVIDIA’s CUDA (Compute Unified Device Architecture) is a general purpose scalable parallel programming model for writing highly parallel applications. The further research will involve proposing a programming approach using hybrid CUDA and OpenMP programming to achieve dramatic speedup for heat equation problem.

Acknowledgment

We would like to express gratitude to Tangpei Cheng for his suggestions that greatly improved this paper. The research was supported by the Fundamental Research Funds for the Central Universities of China and the Financial Code is 53200959333.

References