Implementing Matrix Multiplications on the Multi-Core CPU Architectures

Nakhoon Baek*
School of EECS
Kyungpook National University
Daegu 702-701
KOREA

Hwanyong Lee
Solution Division
HUONE Inc.
Daegu 702-205
KOREA

Abstract: Recent commercial microprocessors are concentrating on the multi-core CPU architectures, while most parallel and/or distributed computing methods focus on the multi-CPU architectures. Therefore, there are needs to analyze and adapt traditional parallel algorithms for the new multi-core environments. In this paper, we use matrix multiplications as the target problem, and implemented it using various methods including the traditional serialized and parallel versions using OpenMP and Windows-threads, etc. We measure the execution times for each implementation, to finally analyze their overall performance. The most important factor for the execution time is the efficient use of level-2 caches in the CPU, according to our experimental results. We expect to develop a more efficient implementation method and design a new matrix multiplication method for the multi-core CPU's.

Key–Words: Multi-core CPU, parallel computing, performance analysis.

1 Introduction

The development of computer hardware is highly dependent on the semiconductor technology. After a sequence of technical innovations, current semiconductor technology is now at its technical limit. Especially, the CPU processing speed is now even restricted by the absolute physical limit, the speed of electrons. To overcome it, we need a new paradigm, and the parallel and/or distributed computation is the most suitable candidate at this time[1].

Conventional parallel and distributed computing methods are focusing on the multi-CPU environment, where multiple CPU's are interconnected and their communications are minimized for efficient processing[2]. On the other side, the latest CPU's for commercial PC's and workstations are concentrating on the multi-core architectures, where several CPU cores are integrated into a single CPU package.

In this multi-core environment, the conventional serialized computing paradigm would be totally inefficient, while the usual parallel computing methods may even be unsuitable[3, 4]. Thus, we need to verify whether the parallel algorithms originally designed for multi-CPU environments are also suitable for multi-core CPU's.

In this paper, starting from these requirements, we select the matrix multiplication as a target problem, and represent various methods to efficiently solve that problem. After implementing all these methods, we will measure their execution time to finally select the most efficient way of implementation. This work would be one of verifications for conventional parallel algorithms toward the multi-core environment. In these days, there are also emphases on the multi-core CPU-based embedded systems[5, 6], and thus, this work is a meaningful result as the basic researches on the acceleration techniques for multi-core CPU's including next generation embedded systems.

2 Matrix Multiplications

In this paper, we will use the matrix multiplications as the target problem, which consists of p matrix equations, $C_h = A_h \cdot B_h$, as shown in Figure 1. All the matrices are $n \times n$ square matrices. As already known in basic algebra texts[7], an element c_{ij} in the matrix C_h is calculated as:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}, \tag{1}$$

where a_{ij}, b_{ij}, c_{ij} are the element in i-th row and j-th column, for the matrices A, B, and C, respectively.

There are variety of matrix multiplication methods, including the Strassen's algorithm[8] which decreases the number of calculations avoiding to evalu-

*Corresponding Author.

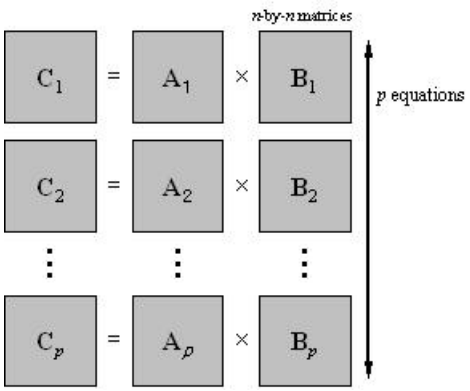

Figure 1: Matrix multiplications.

ate Equation (1) in a brute-force way. Since this paper focuses on the implementation methods for a single algorithm rather than verifying various algorithms, we will represent the direct implementation of Equation (1) as follows:

- a serialized method
- two OpenMP-based methods
- two Windows thread-based method

Now these methods are explained in the following sections.

2.1 Serialized method

This method means the direct implementation of Equation (1), as the pseudo-code shown in Figure 2. Although it is the most straight-forward implementation in the serialized programming paradigm, it uses only one CPU core even for multi-core CPU's and thus, will show inefficiency.

2.2 OpenMP-based method

Since the `.NET` 2005 version of the Microsoft Visual Studio compiler, we can use OpenMP facility. OpenMP enables C and C++ programs to integrate

```
for (h = 0; h < p; h++) {
   for (i = 0; i < n; i++) {
      for (j = 0; j < n; j++) {
         C[i][j] = 0;
         for (k = 0; k < n; k++)
            C[i][j] += A[i][k] * B[k][j];
      } // j
   } // i
} // h
```

Figure 2: Pseudo code for the serialized method.

```
#pragma omp parallel for private(i,j,k)
for (h = 0; h < p; h++) {
   for (i = 0; i < n; i++) {
      for (j = 0; j < n; j++) {
         C[i][j] = 0;
         for (k = 0; k < n; k++)
            C[i][j] += A[i][k] * B[k][j];
      }
   }
}
// end of #pragma
```

Figure 3: Pseudo code for the OpenMP-based method.

parallel processing features using `#pragma` directives of the C/C++ preprocessor[9, 10]. We first use the OpenMP feature to execute a single matrix equation for each thread, as shown in Figure 3. The line starting with the `#pragma` directive activates the OpenMP to automatically invoke multiple threads. In this implementation, each thread needs to simultaneously access the memory area for different matrices, and thus, we can guess that the CPU cache will be somewhat inefficiently used.

2.3 Another OpenMP-based method

As a variation of the OpenMP-based method explained in the previous subsection, this implementation lets the multiple threads simultaneously calculate a single matrix equation, as shown in Figure 4. Dividing the totally n rows by q threads, each thread calculates the n/q rows in the final result matrix C. From the viewpoint of cache usage, the multiple threads share a single matrix information and thus will be more efficient.

```
for (h = 0; h < p; h++) {
#pragma omp parallel for private(j,k)
   for (i = 0; i < n; i++) {
      for (j = 0; j < n; j++) {
         C[i][j] = 0;
         for (k = 0; k < n; k++)
            C[i][j] += A[i][k] * B[k][j];
      }
   }
// end of #pragma
}
```

Figure 4: Pseudo code for another OpenMP-based method.

```
T = number of threads;
for (i = 0; i < T; i++) {
   begin[i] = n / T * i;
   end[i] = n / T * (i + 1);
}

create T threads
for each thread with thread number q {
   for (h = 0; h < p; h++) {
      for (i = begin[q]; i < end[q]; i++) {
         for (j = 0; j < n; j++) {
            C[i][j] = 0;
            for (k = 0; k < n; k++)
               C[i][j] += A[i][k]*B[k][j];
         }
      }
   }
} // end of each thread
join all threads;
```

Figure 5: Pseudo code for the thread-based method.

```
T = number of threads;
for (i = 0; i < T; i++) {
   begin[i] = n / T * i;
   end[i] = n / T * (i + 1);
}

create T threads
for each thread with thread number q {
   for (h = 0; h < p; h++) {
      for (i = begin[q]; i < end[q]; i++) {
         for (j = 0; j < n; j++)
            C[i][j] = 0;
         for (k = 0; k < n; k++)
            for (j = 0; j < n; j++)
               C[i][j] += A[i][k]*B[k][j];
      }
   }
} // end of each thread
join all threads;
```

Figure 6: Pseudo code for the row-wise thread-based method.

2.4 Thread-based method

Instead of using OpenMP features provided by the Visual Studio compiler, this implementation uses conventional Windows thread library[11] to create and control threads. Similar to the method in the previous subsection, each thread takes charge of n/q rows, as shown in Figure 5. When the current OpenMP implementation is already fully optimized, this method will show the same execution speed in comparison with the previous OpenMP-based method.

2.5 Row-wise thread-based method

Row-wised matrix multiplication[10] is already used in traditional parallel processing areas, and it changes the loop order, to process the matrix B in a row-majored order, rather than usual column-majored order, as shown in Figure 6. Though it may look inefficient at the first glance, it can much increase the cache-hit ratio especially for large-sized matrices, and thus has the potential of very efficient execution.

3 Experimental Results

We have implemented all 5 methods represented in sections 2.1 to 2.5. This section will analyze their experimental results.

In our experiments, we use PC's with 2GB main memory and a single Intel XQ6700 CPU. The XQ6700 CPU has 4 CPU cores, and two of them share a 4MB level-2 cache, respectively, with 8MB level-2 cache in total. Windows XP professional version and Visual Studio `.NET` 2005 professional version are used for the experiments, with OpenMP and Windows thread libraries.

In the matrices, each element can be easily integer types for usual mathematical calculations, or especially for graph algorithms. Our experiments starts from 100×100 matrices up to $1,000 \times 1,000$ matrices, stepping 100 rows and columns, to totally 10 difference sizes. For each size, we randomly generate $m = 4$ matrix equations and calculate them using 5 difference methods represented in section 2.

To get more precise results, we measured the elapsed time for 5 times and use their average as the final result. The final results are shown in Table 1, and its graphical representation is shown in Figure 7. It is noticeable that the execution times for the OpenMP-based method in section 2.3 are very close to those

Table 1: Measured execution times.

(unit: sec)

n	methods				
	serial	OpenMP[1]	OpenMP[2]	thread	rowwise
100	0.013	0.006	0.009	0.003	0.003
200	0.078	0.022	0.022	0.022	0.013
300	0.259	0.069	0.075	0.066	0.034
400	0.628	0.159	0.163	0.159	0.081
500	1.225	0.313	0.313	0.313	0.156
600	2.841	0.813	0.809	0.806	0.275
700	4.856	1.384	1.337	1.337	0.419
800	7.778	2.628	2.000	2.006	0.622
900	12.716	4.191	3.166	3.169	0.884
1,000	19.425	7.478	4.897	4.916	1.216

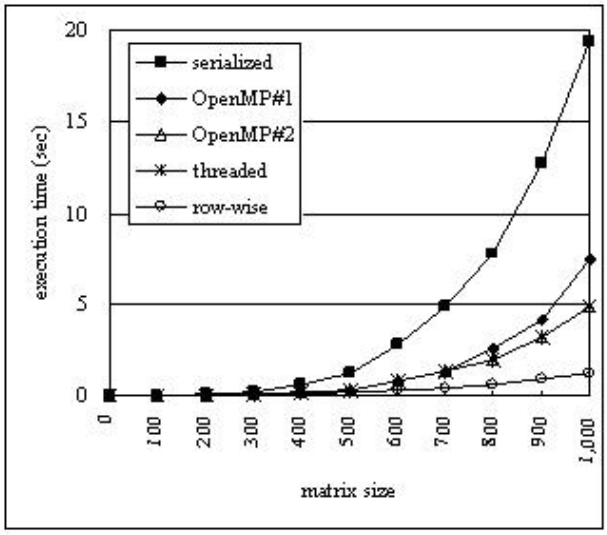

Figure 7: Execution times.

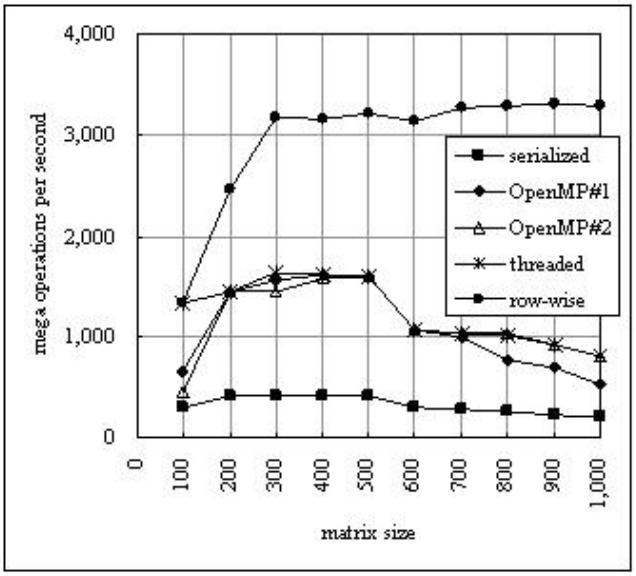

Figure 8: Performance analysis.

of the thread-based method in section 2.4. Since the OpenMP features are implemented using Windows thread in the Window XP environment, and in relatively simple cases such as these, the automatically generated codes from OpenMP can be nearly optimal to the hand-written codes, and thus, the execution times would be very similar to each other.

Although the execution time itself is an important factor, the operations performed in a given time period can be act as a more important factor. Given m matrix equations with $n \times n$ matrices, the required calculations are totally $O(mn^3)$ for all methods represented in section 3. Thus, we simply calculate the amount of required operations as mn^3 and divide it with the execution time. The final result of operations per second are summarized in Table 2, and its corresponding graph is given in Figure 8.

Table 2: Operations per second.

(unit: 10^6 operations/sec)

n	methods				
	serial	OpenMP^1	OpenMP^2	thread	rowwise
100	307.6	666.6	444.4	1,333.3	1,333.3
200	410.2	1,454.5	1,454.5	1,454.5	2,461.5
300	416.9	1,565.2	1,440.0	1,636.3	3,176.4
400	407.6	1,610.0	1,570.5	1,610.0	3,160.4
500	408.1	1,597.4	1,597.4	1,597.4	3,205.1
600	304.1	1,062.7	1,067.9	1,071.9	3,141.8
700	282.5	991.3	1,026.1	1,026.1	3,274.4
800	263.3	779.3	1,024.0	1,020.9	3,292.6
900	229.3	695.7	921.0	920.1	3,298.6
1000	205.9	534.9	816.8	813.6	3,289.4

The serialized method shows the worst performance, as expected. Two OpenMP-based methods and the thread-based methods show similar results. As shown in Figure 8, their performances are rapidly dropped in the interval corresponding to the matrix size 500 to 600. In the case of row-wise threaded methods, it shows the best performance even for matrix sizes up to 1,000.

We infer that the performance mainly depends on the level-2 cache size of the CPU. The XQ6700 CPU used in our experiment has two 4MB level-2 caches. Since they are completely separated, each CPU core can use at most 4MB cache memory, which is actually shared by another CPU core. Since a 500×500 integer matrix needs $4 \times 500 \times 500 = 1M$ bytes, the matrix equation $C = A \cdot B$ requires 3 matrices, and the total memory space comes to over than 3M bytes. Thus, the time delays due to cache faults will be first introduced when the matrix size becomes more than 500.

In contrast, the row-wise method needs to get the whole matrix B, but only one rows of matrix C and A. In this case, the cache can be used up to the matrix size of $1,000 \times 1,000$. Thus, the row-wise method shows consistently outstanding performance over the all cases up to $1,000 \times 1,000$.

4 Conclusion

In this paper, we analyzed which implementation method shows the best performance, on the quad-core CPU's, which are newly introduced into the PC market. We use the matrix multiplication as the target problem and showed 5 different implementation

methods, even with OpenMP and Windows thread libraries, to finally get overall experimental results.

After comprehensively analyzing our experimental results, we conclude that the performance is mainly affected by the usage of level-2 caches by the CPU cores, rather than the detailed steps or used libraries. Conclusively, we had better design algorithms to efficiently use the cache, especially for massive calculations such as matrix multiplications.

Acknowledgements: This work is financially supported by the Ministry of Education and Human Resources Development(MOE), the Ministry of Commerce, Industry and Energy(MOCIE) and the Ministry of Labor(MOLAB) through the fostering project of the Industrial-Academic Cooperation Centered University.

References:

[1] W. Knight. Two heads are better than one. *IEE Review*, 51(9):32–35, 2005.

[2] K. A. Berman and J. L. Paul. *Algorithms: Sequential, Parallel, and Distributed*. Course Technology, 2005.

[3] K. Wackowski and P. Gepner. Hyper-threading technology speeds clusters. In *Parallel Processing and Applied Mathematics, 5th Int'l Conf.*, pages 17–26, 2003.

[4] T. Martinez. Understanding dual processors, hyperthreading technology, and multicore systems, 2005. http://www.devx.com/goparallel/article/27399.

[5] M. Levy. Dealing with the design challenges of multicore embedded systems, 2006. http://www.embedded.com/showArticle.jhtml?articleID=177102340.

[6] S. Daily. Software design issues for multicore/multiprocessor systems. In *Embedded Systems Conference*, April 2006.

[7] H. Anton. *Elementary Linear Algebra, 9th Ed.* John Wiley & Sons, 2004.

[8] V. Strassen. Gaussian elimination is not optimal. *Numer. Math.*, 13:354–356, 1969.

[9] T. Mattson and R. EigenMann. OpenMP: An API for writing portable SMP application software. In *SuperComputing 99*, November 1999.

[10] M. J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill, 2003.

[11] J. Beveridge and R. Wiener. *Multithreading Applications in Win32: The Complete Guide to Threads*. Addison-Wesley, 1997.