PERFORMANCE COMPARISON OF MULTICORE PROCESSORS USING VARIOUS SOFTWARE PLATFORMS

Conference Paper · September 2007

CITATIONS
0

READS
575

3 authors:

Igor Mishkovski
Ss. Cyril and Methodius University in Skopje
71 PUBLICATIONS **402** CITATIONS
SEE PROFILE

Dimitar Trajanov
Ss. Cyril and Methodius University in Skopje
158 PUBLICATIONS **626** CITATIONS
SEE PROFILE

Aksenti Grnarov
Ss. Cyril and Methodius University in Skopje
53 PUBLICATIONS **410** CITATIONS
SEE PROFILE

PERFORMANCE COMPARISON OF MULTICORE PROCESSORS USING VARIOUS SOFTWARE PLATFORMS

Igor Miskovski[1], Dimitar Trajanov[1], and Aksenti Grnarov[1]

[1]University SS. Cyril & Methodius, Faculty of Electrical engineering and Information Technologies, Department of Computer Science, Karpos 2 bb, Skopje, R. Macedonia, igorm@feit.ukim.edu.mk

Abstract—**In the last two years, with appearance of multicore processors, the need of parallel programming has become essential. Programmers using multicore processors are forced to deliver the best performance. But in their effort they must face the selection of the programming model. We have examined the performance on multicore processors for most popular languages and models of parallel programming. For analysis we chose the C++, C# and Java programming languages and MPI and OpenMP programming models.**

Index terms—**Performance Measurement, Multicore processors, MPI, OpenMP, Java, C#, Threading**

1. INTRODUCTION

Concurrent programming is difficult [1], yet many technologists predict the end of Moore's law will be answered with increasingly parallel computer architectures—multicore or chip multiprocessors (CMPs) [2]. If we hope to achieve continued performance gains, programs must be able to exploit this parallelism.

Because of the big explosion of the multicore processors for home use and the need of parallelizing the existing software sequential applications, in order to help the programmer, we present a performance comparison of the message passing paradigm MPI and shared memory paradigms (OpenMP, multithreading with system calls, C#.NET and Java). Nowadays widely accepted programming languages like C/C++ offer the opportunity to use functions from MPI libraries, OpenMP directives and integrated multithreading, resulting in decreasing the overall execution time of the today's application. For our analysis we used two parallel algorithms, the first one for calculation the PI number and the second one for matrix multiplication. We have chosen these two algorithms because the first one is only computational intensive and there is no input data, whereas on the other hand multiplication of square matrices is algorithm both computational and data intensive. Thus, this paper provides two contributions: the comparison of performance of different multicore processors and comparison of performance of different programming languages and parallel programming models.

The paper is organized in the following way. The second section describes the multicore platform by explaining multicore processors from top industry multicore processor manufacturers: Intel and AMD. It also gives an overview of used processors in our simulation. Section 3 describes the software solutions for parallel programming on multicore processors, while section 4 defines the problems and algorithm implementation in a certain model. Section 5 presents performance comparison results. Section 6 concludes.

2. MULTICORE PLATFORM

Multicore is the hot topic in the latest round of CPUs from Intel, AMD and Sun. As clock speed increases becoming more and more difficult to achieve, vendors have turned to multicore CPUs as the best way to gain additional performance. Even though the concept of using concurrent CPUs to increase overall software performance has been around for at least 35 years, remarkably little in the way of development tools has made it to the commercial marketplace. As a result, the vast majority of applications are single-threaded [3]. Today, multicore architectures are an inflection point in mainstream software development because they force developers to write parallel programs [4]. Intel and AMD, the top industry rivals, have already introduced dual-core and

quad-core chips for desktop PCs. And that's just the start of a trend that could bring an important change to PCs: multicore processing [5].
Intel currently uses Core Duo (based on Pentium M), Core 2 Duo and Xeon (based on Core) microprocessors with dual-core technology for low-end computers. Besides increased processor speed, one of the primary differences between the Intel's Core Duo and the Core 2 Duo is an increase in the amount of the shared Level 2 cache. The Core 2 Duo has doubled the amount of on-board cache to 4 MiB. Both chips have 65 nm process technology architecture and support a 667-1066 MHz front-side-bus (FSB).

The AMD's first desktop-based dual core processor family — the Athlon 64 X2 can be distinguished from Intel's early dual-core design, as the X2 mated two cores into a single chip, rather than two chips on a single package. Intel's method with the Pentium D may have had theoretical yield advantages, but it gave up some performance advantage since interprocessor communication still happened over external pins, rather than internally. The X2 improved upon the performance of the original Athlon 64, especially for multi-threaded software applications.

2.1 Used microprocessors

For the purpose of our simulation we have used three different dual-core processors and a single-core processor. A machine has been used with two **AMD Opteron model 275 Dual Core 2.2GHz** processors with a 2 x 1024 KB L2 cache, 64KB L1 Data Cache, 64KB L1 Code Cache and HyperTransport Technology Speed of 1GHz.

The second PC we have used is equipped with **Athlon 64 X2 Dual Core Processor 4200+**, that runs at 2.2GHz with 2 x 512 KiB L2 Cache, 64KB L1 Data Cache, 64KB L1 Code Cache and HyperTransport Technology Speed of 1GHz.

We used another dual core processor, **Pentium D 805 Dual Core processor, that runs at 2.66GHz** with a 533MHz FSB and shares two separate 1MB L2 caches that are located on the processor. This processor fully supports 64-bit computing via the Intel EM64T technology.

We have also used PC with **AMD Sempron 2600+ processor** that runs on 1.6 GHz with a 200.9 MHz FSB. The cache memory consists of 64KB L1 Data Cache, 64KB L1 Code Cache and 128KB L2 Cache.

3. PROGRAMMING MODELS

Throughout the years, many different parallel programming techniques were implemented. Through the development of parallel application most dominant alternatives have become message passing and multithreading programming. These two approaches differ in how the concurrent segments of the application share the data and how they synchronize their work.

3.1 Message Passing Interface

The Message Passing Interface (MPI) [6] is a specification for a set of functions for managing the movement of data among sets of communicating processes. MPI defines functions for point-to-point communication between two processes, collective operations among processes, parallel I/O and process management. In addition, MPI's support for communicators facilitates the creation of modular programs and reusable libraries. Communication in MPI specifies the types and layout of data being communicated, allowing MPI implementations to both optimize for noncontiguous data in memory and support clusters of heterogeneous systems.

One implementation of MPI is MPICH2, whose goal is to provide an MPI implementation for important platforms, including clusters, SMPs and massively parallel processors. It also provides a good start for developing new and better parallel programming environments.

3.2 Multithreaded Programming in C++

C++ does not contain any built-in support for multithreaded application [7]. Instead it relies entirely upon the operating system to provide this feature. In this way, C++ allows directly to utilize the multithreading features provided by the operating system. The operating system defines a rich set of thread related functions that enable finely grained control over the creation and management of a thread. There are several ways to control access to shared resource, including semaphores, mutexes, event objects, waitable timers and critical regions. With C++, one can gain access to all the features that the operating system provides. This is a major advantage when writing high-performance code.

3.3 OpenMP

OpenMP [8] provides three kinds of directives: parallelism/work sharing, data environment, and synchronization.

OpenMP uses the fork-join model of parallel execution. An OpenMP program begins execution as a single process, called the master thread of execution. The fundamental directive for expressing parallelism is the parallel directive. It defines a parallel region of the program, which is executed by multiple threads. When the master thread enters a parallel region, it forks a team of threads (one of them being the master thread), and work is continued in parallel among these threads. Upon exiting the parallel construct, the threads in the team synchronize (join the master), and only the master continues execution. The statements in the parallel region, including functions called from within the enclosed statements, are executed in parallel by each thread in the team. The statements enclosed lexically within a construct define the static extent of the construct. The dynamic extent further includes the functions called from within the construct.

3.4 Multithreading in .NET

By nature and architecture .NET is a multi-threaded environment. This environment has been chosen because of the popularity of the programming language C#.NET [9]. Today, most applications tend to be written in C#.NET and because this paper is intended for measuring performance on multicore processors, processors for future low-end computers, we decided to include multithreading in .NET. There are two main ways of multi-threading which .NET encourages: starting own threads and using the pool either directly or indirectly using asynchronous methods. In general, one should create a new thread "manually" for long-running tasks, whereas for short-running tasks, particularly those created often, the thread pool is an excellent choice. The thread pool can run many jobs at once, and uses framework classes internally. In case of synchronization as there is limited amount of recourses, there can be a restriction on the access to the resource by one thread at a time. In these situations one can consider implementing locking on the thread.

3.4 Multithreading in Java

Java is a programming language which has been designed to support concurrent programming [10], and all execution in the language takes place in the context of a thread. It is important for a Java programmer to understand both the power and limitations of Java threads.

In the JVM (Java Virtual Machine) [11], objects and resources can be accessed by many separate threads; each thread has its own path of execution but can potentially access any object in the program. The programmer must ensure that threads do not interfere with each other, and that resources are properly coordinated (or "synchronized") during both read and write access. The Java language has built-in constructs to support this coordination. When the object resources are being used by multiple threads produced by the re-entrant-capable process i.e. object, each of these threads are competing. The Java environment handles the competition on the object resources by these threads by providing the monitor mechanism, similar to the semaphores in other languages. While the monitors are being acquired or released for implementing the synchronization, the threads enter into the waiting pool, internal to the Java environment.

The Java Language Specification does not say how the JVM designer should implement the multithreading primitives specified, because there is so much variation among the various operating systems and hardware on which the JVM is expected to run.

4. IMPLEMENTATION ISSUE

4.1 Used algorithms

For the purpose of this paper we used two algorithms: calculation of PI and dense matrix multiplication.

The value of PI can be calculated with integration of non-negative function in given interval (1).

$$\int_0^1 \frac{4}{1+x^2}\,dx \qquad (1)$$

The integration is approximately done by dividing the region in regular geometric forms and summation of the area of these forms. In our simulation we divided the region in 500 millions forms and obtained the value of PI by summation of the calculated area.

For matrix multiplication we have used the well-known sequential algorithm. In our simulation we have multiplied two square matrices with dimension 1000 x 1000.

4.2 From sequential to parallel programming

Given the sequential algorithms we used manual method for developing parallel code. The parallel solution is computationally intensive with minimum communication. Each parallel task works on a portion of the data, which is block partitioned. When parallelizing the given algorithm we have used coarse-grain parallelism with high computational to communication ratio which implied more opportunity for performance increase. The following text gives a short description of the parallel algorithms.

OpenMP has many utility functions, but the most of the work is done with #pragma commands. We use the *parallel for* directive for creating concurrent threads and *for* statement for delivering the work. The *private* clause means that every thread has its own local copy of the *x* variable (fig. 1).

```
#pragma omp parallel for reduction(+:sum) private(x)
for (i=1;i<= num_steps; i++)
{
        x = (i-0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
}
pi = step * sum;
```

Fig. 1 Calculating PI number using OpenMP

Implementation with MPI required set of function for managing the movement of data among sets of communication processes (fig. 2). The programmer must foresee the decomposition of the problem.

```
MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);
sum = 0.0;
h   = 1.0 / N;
for (i = 1+rank; i <= N; i += nprocrs) {
        x = h * (i - 0.5);
        sum += 4.0 / (1.0 + x*x);
}
mypi = h * sum;
MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

Fig. 2 Calculating PI number using MPI

With multithreading in C++ program developers face the race conditions, deadlocks, mutexes. It's very hard for the programmer to isolate the defects and debug the program. The program code becomes unreadable and uncommon for the way the programmers' mind works (fig. 3).

```
for(i=0; i<num_threads; i++) threadArg[i] = i+1;
        InitializeCriticalSection(&hUpdateMutex);
        for (i=0; i<num_threads; i++)
        {
                thread_handles[i] = CreateThread(0, 0,(LPTHREAD_START_ROUTINE)
                Pi,&threadArg[i], 0, &threadID);
        }
        WaitForMultipleObjects(num_threads,thread_handles, TRUE,INFINITE);
}
pi = global_sum * step;
```

Fig. 3 Calculating PI number using Multithreading in C++

In our implementation of multithreading with C#.NET we used the Thread class from the System.Threading namespace. We manually created threads from the current thread and each of these threads used different portion of data (fig. 4). Also we used the thread pool but the results in either case were the same.

```
static void ThreadJob(object st)
{
   int j;
   double x, sum = 0.0;
   int start = Convert.ToInt32(st);
   AutoResetEvent are = (AutoResetEvent) evs[start];
   start = threadArg[start];

   step = (1.0 / num_steps);
   for (j = start; j <= num_steps; j += num_threads)
   {
      x = (j - 0.5) * step;
      sum = sum + 4.0 / (1.0 + x * x);
   }
   Thread.BeginCriticalRegion();
   global_sum += sum;
   Thread.EndCriticalRegion();
   are.Set();
}
```

Fig. 4 Calculating PI number using Multithreading in .NET

The Java environment handles race conditions by providing monitor mechanism. While the synchronization is implemented, the threads enter into the waiting pool, internal to the Java environment (fig. 5).

```
   public synchronized void run() {
   int i;
   double step = 1.0/(double)this.num_steps;
   double x, sum = 0.0;

   for (i=this.start;i<= this.num_steps; i=i+tredovi)
   {
            x = (i-0.5)*step;
            sum = sum + 4.0/(1.0+x*x);
   }
   global_sum += sum;
}
```

Fig. 5 Calculating PI number using Multithreading in Java

5. PERFORMANCE COMPARISON RESULTS

All the measurements were carried out after all unnecessary processes were stopped and the network was unplugged, thus lowering to minimum CPU load. In order to analyze the influence of multiple threads we ran the parallelized applications with different number of threads (from 1 to 8 threads). For each run we make 10 measurements and then we get the average value.

So as to notice the possible performance decrease of the parallelized applications when they run on single-core processors, we measure performance on single-core AMD Sempron 2600+ processor. Fig.6 shows the speedup of PI calculation relative to sequential application depending on the number of threads for MPI, OpenMP, C++, Java and C#. The speedups of all applications are near 1 except for Java which has smaller speedup of around 0.8. In addition the speedup is not dependent on the number of threads. Fig. 7 shows the speedup of the matrix multiplication relative to sequential application depending on the number of threads for MPI, OpenMP, C++, Java and C#. The best result shows the MPI, second is OpenMP and third is C++. The C# and Java implementations have the poorest performance, because their execution is on virtual machines.

The performance of the same applications for Intel Pentium D Dual Core processor is shown in fig. 8 (PI calculation) and fig. 9 (matrix multiplication). When applications are running with more than one thread the gained speedup for PI calculation is near 2 for all applications except for Java which has very bad speedup (around 0.5 for one thread and 1 for two threads). For matrix multiplication the performances are similar. The best speedup is achieved with OpenMP application only when even number of threads is used and this is the only case where OpenMP is better than MPI.

The measured performance on AMD AthlonX2 procesoor is similar to that on the Intel Dual Core processor. The only difference is that the Java application has better speedup than that achieved on the Intel Dual Core processor. The results are shown in fig. 10 (PI calculation) and fig. 11 (matrix multiplication).

When using AMD Opteron 275 one can see the linear increase of the speedup relative to the number of execution threads (until the number oft threads reach the number of processors), for both the PI calculation and the matrix multiplication (fig. 12 and fig. 13). It is noticeable that using five threads gives the worst performances and what is more the speedup is almost the same as when three threads are used. C# performances with matrix multiplication are worse than the ones with PI calculation. Yet still Java is the worst solution.

Fig. 14 shows that all software platforms have nearly the same average speedup of multithreaded version of PI calculation relative to single threaded version, whereas fig. 15 shows that in general Java has the best average speedup of multithreaded version of matrix calculation relative to single threaded version for all test processors.

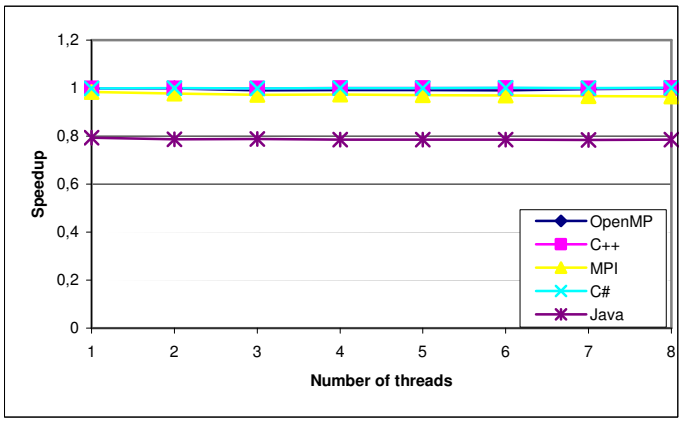

Fig. 6 Speedup of PI calculation relative to sequential application depending on the number of threads for AMD Sempron 2600+

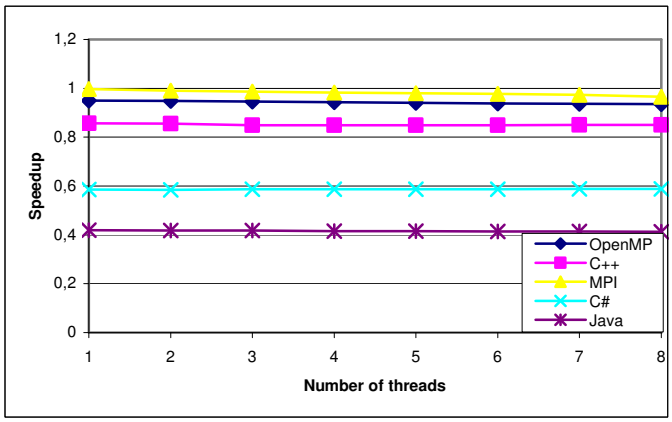

Fig. 7 Speedup of matrix multiplication relative to sequential application depending on the number of threads for AMD Sempron 2600+

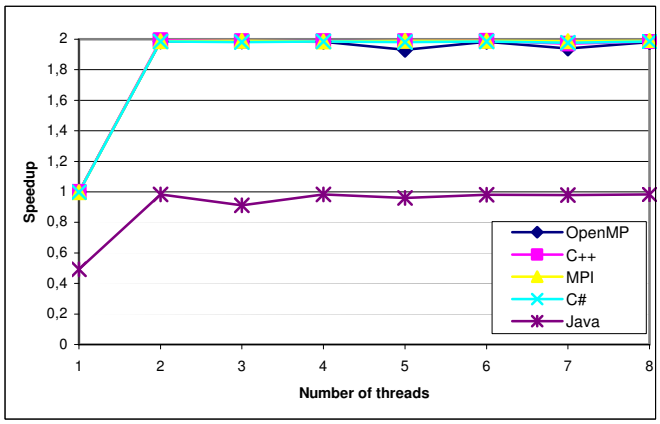

Fig. 8 Speedup of PI calculation relative to sequential application depending on the number of threads for Intel Pentium D Dual Core

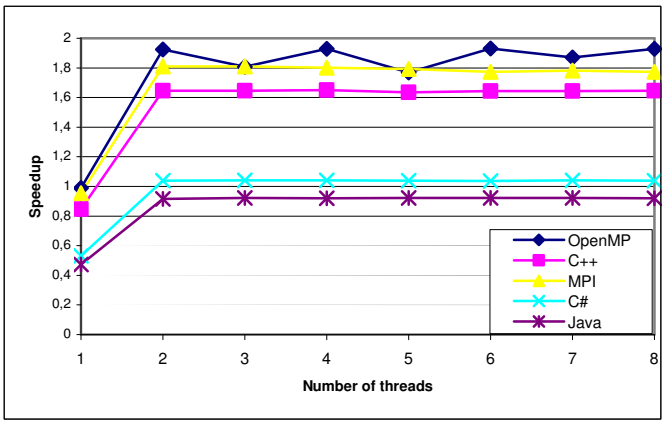

Fig. 9 Speedup of matrix multiplication relative to sequential application depending on the number of threads for Intel Pentium D Dual Core

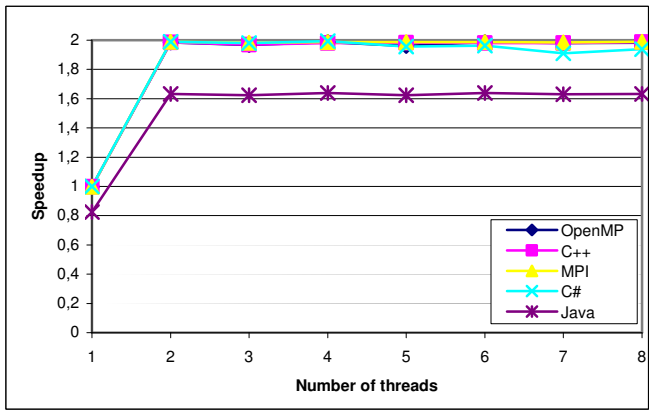

Fig. 10 Speedup of PI calculation relative to sequential application depending on the number of threads for AMD Athlon X2

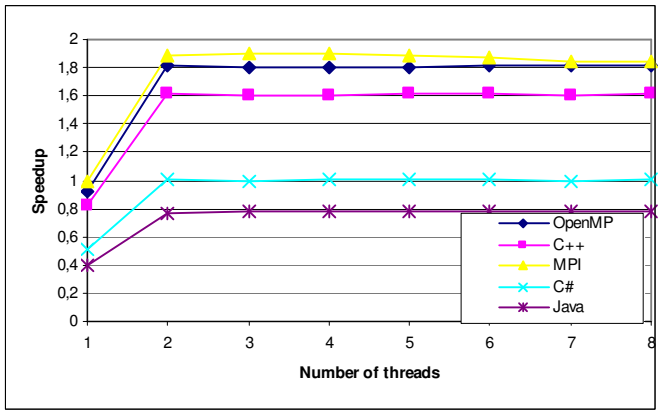

Fig. 11 Speedup of matrix multiplication relative to sequential application depending on the number of threads for AMD Athlon X2

6. Concluding Remarks

Every programmer chooses certain platform for parallel programming depending on the algorithm that he uses. He must predict the performance drawback when his parallel application will be used on a machine with a single core. The ideal speedup is very rarely achieved. When parallelizing loops, OpenMP seems the easiest way and also gives very good results. Programmers must allow dynamic change of the number of threads depending on the number of cores in the processor. Our conclusion is that parallel programming with OpenMP is the easiest solution for the programmer to choose. The performance loss at a single-core machine when using parallel application for PI calculation is relatively small. For

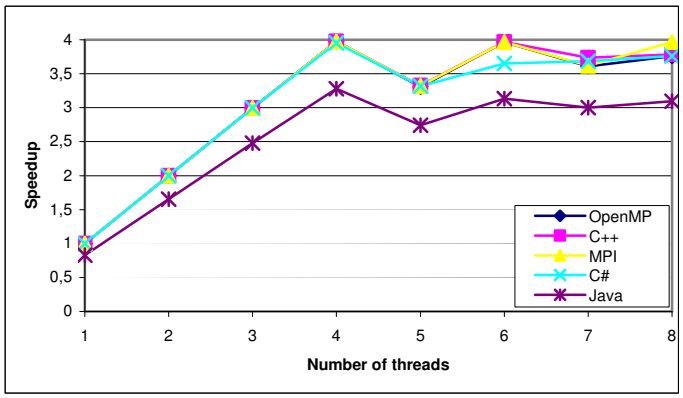

Fig. 12 Speedup of PI calculation relative to sequential application depending on the number of threads for AMD Opteron 275

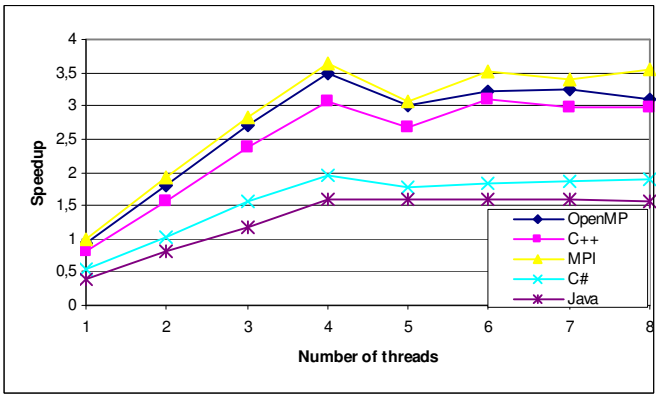

Fig. 13 Speedup of matrix multiplication relative to sequential application depending on the number of threads for AMD Opteron 275

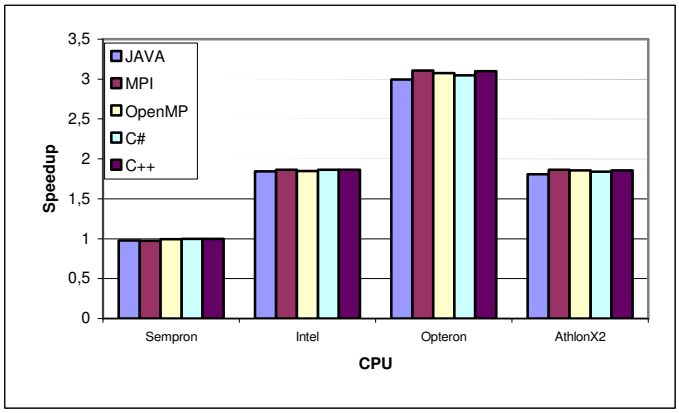

Fig. 14 Average speedup of multithreaded version of PI calculation relative to single threaded version

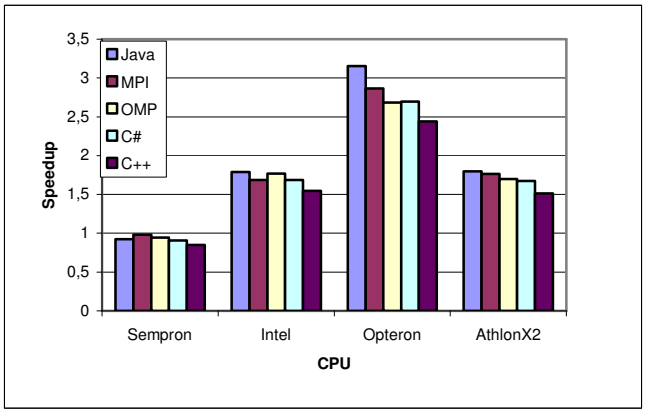

Fig. 15 Average speedup of multithreaded version of matrix calculation relative to single threaded version

matrix multiplication the loss of the performance is significant, about 2% for MPI, 6% for OpenMP, 17.5% for C++, 70% for C# and 140% for Java. Thus when we have application similar to Matrix multiplications it might be better to have two versions, one parallel and one sequential.

The Java and C# have poor performance. Their two-threaded versions when run on multicore processors have similar performance like C++ version with a single thread. Thus writing parallel application in C# and Java does not mean performance increase, and it might be better and easier to rewrite the application in C++ rather than parallelizing the original application. It is interesting to note that Java has the worst performances, but it has the best speedup relatively to his single threaded version, which indicates good implementation of multithreading subsystem.

For future work, parallelizing open source programs will take part in our simulation and this parallelization will be also done with other parallel languages like JavaPP, HPF, Parallel C and C++.

7. REFERENCES

[1] H. Sutter and J. Larus, “Software and the Concurrency Revolution”, ACM Queue, vol. 3, no. 7, 2005, pp. 54-62.

[2] Hesham El-Rewini and Mostafa Abd-El-Barr, *Advanced Computer Architecture and Parallel Processing*, Wiley-Interscience, 2005

[3] Mache Creeger „Multicore CPUs for the Masses”, ACM Queue vol. 3, no. 7 - September 2005

[4] Ali-Reza Adl-Tabatabai, Christos Kozyrakis, Bratin Saha, “Unlocking Concurrency”, ACM Queue vol. 4, no. 10 - December 2006 / January 2007

[5] Laurianne McLaughlin, “Multicore mania”, december 2005

[6] J. Dongarra et al, Morgan “Sourcebook of parallel computing”, Elsevier, 2003

[7] http://www.devarticles.com/c/a/Cplusplus/Multithreading-in-C/

[8] OpenMP ARB, “OpenMP API”, http://www.openmp.org/

[9] Mark Strawmyer, “Multithreading in .NET Applications”,http://www.codeguru.com/columns/dotnet/article.php/c4611/, July 2003

[10] http://en.wikipedia.org/wiki/Java_concurrency

[11] http://java.ittoolbox.com/documents/popular-q-and-a/multithreading-fundamentals-2185