Assignment #1a: Report

Advanced Multicore Systems

Tudor Voicu – 4422066

Misael Hernandez – 4423615

May 12th, 2015

Introduction

The first assignment of the Advanced Multicore Systems course consists on optimizing vector multiplications through the use of OpenMP. With this purpose, we profiled the parallelization capabilities of our system, and compared the optimum parallel execution times with sequential execution times.

OpenMP is an API for C/C++ with facilitates shared memory management for parallelization purposes. Through the use of compiler directives, OpenMP takes care of thread creation, load distribution and memory management for parallelization of C/C++ code.

In this lab we used OpenMP both for optimizing vector multiplication and matrix multiplication. This report includes the tasks required for this, along with a comparison of results and the respective conclusions.

Task 1

The first exercise consisted on obtaining the parallel execution time of the vector-matrix multiplication, keeping the size fixed, but changing the number of threads each time. Because the execution time is also dependent on the number of threads, we can observe the exact number of threads that gives the best speedup in our system. Table 1 shows the results for a vector size of 8000, and a number of threads from 1 to 12. The sequential execution ran at an average of 2.504 seconds, which is the baseline used for all the speedup calculations. Figure 1 shows the parallel execution time for each of the number of threads, and Figure 2 does the same for speedup.

Table 1. Parallel execution time and speedup by number of threads

# Threads	Execution time	Speedup
1	2.755	0.909
2	1.449	1.728
3	0.980	2.555
4	0.810	3.091
5	0.901	2.779
6	0.897	2.792
7	0.797	3.142
8	0.791	3.166
9	0.800	3.130
10	0.828	3.024
11	0.831	3.013
12	0.853	2.936

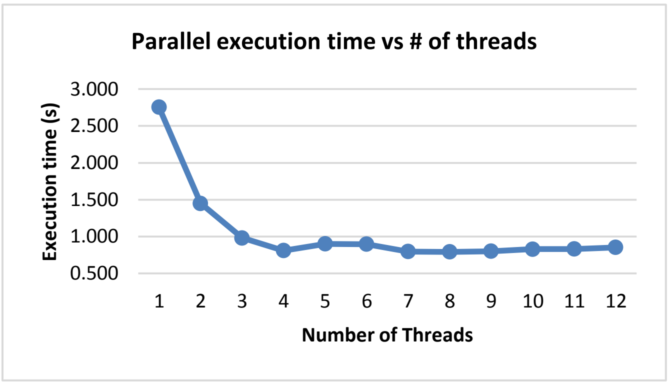

Figure 1. Effect of the number of threads on the execution time

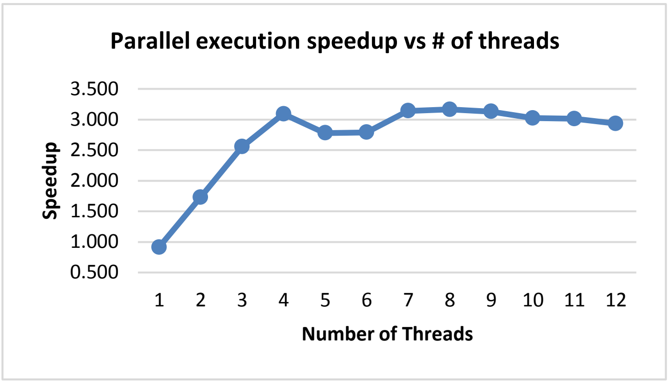

Figure 2. Effect of the number of threads on the speedup achieved by parallelization

What is noticeable from the results is that there are two local minima. The first one occurs at 4 threads and the second one at eight threads. The system in which the program was run has 4 cores, but each one has hyper-threading as well. From 1 thread to 4, the improvement comes from using one more core each time (efficiently). After that, it gets slower because at least one core has to run 2 threads (in average), which comes with a noticeable overhead. However when running 8 threads, the 4 cores with hyper-threading (which allows two threads to be run at each core in a somewhat parallel fashion, for a total of 8) are used most efficiently. More than 8 threads, which is the most that could be run in parallel, the overhead of managing the threads (moving them between the ready queue and execution) affects the performance negatively. So effectively, 8 threads is the most suitable for this system.

Task 2

For the second task we now keep the number of threads constant at the most suitable number, which is 8, and plot the effects of the input size on the execution time. This is done on both the parallel and the serial functions with the purpose of comparing them. Table 1 shows the results. In Figure 3 we can see the comparison of the execution times for the parallel and serial functions in a logarithmic scale. Figure 4 shows the speedup of the parallel version compared to the serial one.

Table 2. Sequential and parallel execution time with speedup by matrix size

Matrix size (N)	Seq Ex	Parallel Ex	Speedup
10	0.000002	0.001697	0.001179
20	0.000006	0.003949	0.001519
40	0.000033	0.007544	0.004374
80	0.000084	0.002200	0.038182
160	0.000327	0.003768	0.086783
320	0.001271	0.002323	0.547137
640	0.007210	0.006441	1.119391
1000	0.019220	0.009433	2.037528
2000	0.111796	0.036506	3.062401
4000	0.540668	0.193470	2.794583
8000	2.712651	0.828773	3.273093
25000	32.525736	13.382240	2.430515

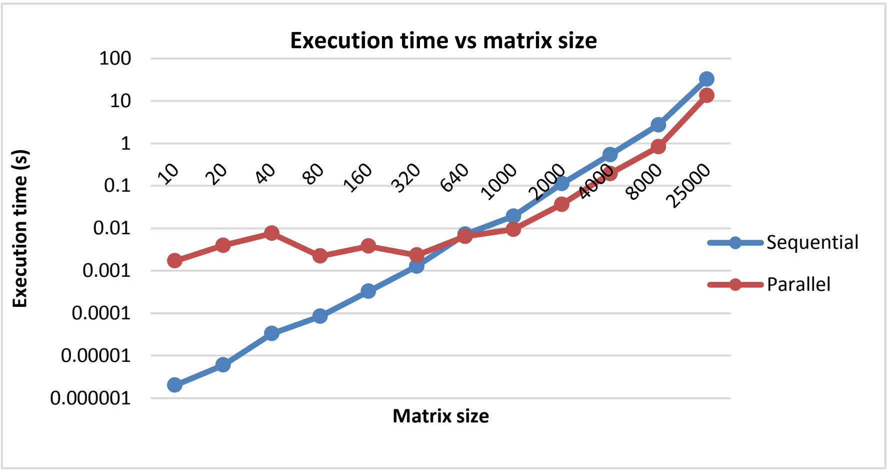

Figure 3. Effect of matrix size on sequencial and parallel execution time

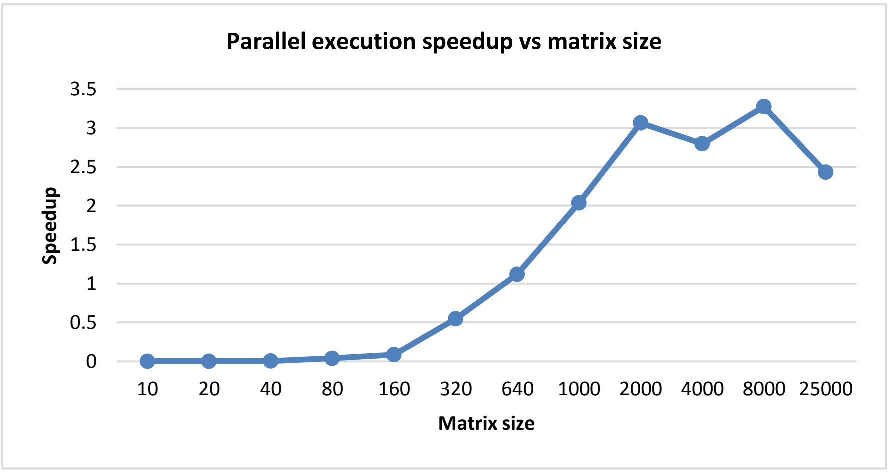

Figure 4. Effect of matrix size on speedup

As seen on Figure 3, the parallel execution time is higher for matrix sizes smaller than 640x640. This can be explained by the overhead involved in parallelization. This involves the creation of the 8 threads and the management of these threads by the operating system and the hardware itself. For small sizes, the overhead is big relative to the calculations, so the execution time ends up being bigger. However, the overhead can be considered more or less constant relative to the input size. This means that as the input grows, the effect of the overhead becomes negligible and the parallel execution becomes faster than the sequential.

Task 3

This task consisted on modifying the current program to execute matrix multiplication. The code for executing the multiplication itself is shown below. Refer to the source code packaged with this report for the rest.

```
void matrix_mult_sq(int size, double *matrix_in,
                      double *matrix_in2, double *matrix_out){
  int rows, cols;
  int j;

  for(rows=0; rows<size; rows++)
    for(cols=0; cols<size; cols++){
      matrix_out[rows*size + cols] = 0.0;
      for(j=0; j<size; j++)
        matrix_out[rows*size + cols] += matrix_in[rows*size + j] * matrix_in2[j*size + cols];
    }
}
```

Figure 5. Matrix multiplication function

The code as it is has a slight problem with the cache. The pointer for the second input matrix, matrix_in, increases by size, which means that if size is big, it is very possible that the next data is not found in the same cache line, causing more cache misses per inner loop execution. If the output matrix is initialized with zeros beforehand (before the start of the outermost loop), this problem can be solved by exchanging the innermost loop with the middle one. In this way the pointers to the three matrices only increase by one, causing less cache misses per innermost loop execution. For the purpose of this assignment, the code was left as shown, to be faithful to the original code.

Task 4

The last task consisted on improving the performance of the matrix multiplication code using techniques such as OpenMP, SSE or such. For further improvements over the OpenMP implementation we have used the AVX extension, which supports simultaneous 4 operations on double-precision numbers (64 bit floating point numbers) per CPU core. The application to be benchmarked is the matrix-matrix multiplication as requested at exercise 1a.3).

The theoretical improvement should be 4 over the OpenMP configuration (in our case, 8 threads achieved highest performance, running on a 4-core/8 threads Intel CPU). However, from Figure 6 it can be seen that the maximum speed-up is slightly above 2 over the parallel implementation and aprox. 6.3 over the sequential one.

The implementation details and the code are provided in the updated matrix3.c file. The application was running on an Intel i7-4700MQ quad-core mobile (laptop) CPU, with tuned frequency at 2.5 Ghz for any core count (we had to bypass the Intel TurboBoost technology in order to get relevant results) and 8 GB of dual-channel DDR3 RAM (2x4GB).

Table 3. Results in seconds for running different size (N) matrix x matrix multiplication in the three mentioned implementations. Speed-up results are shown in the two left-most columns.

Matrix size (N)	Sequential (sec)	Parallel (sec)	AVX + FMA (sec)	Speed-up over OpenMP	Speed-up over sequential
384	0.5229	0.1807	0.0972	1.8599	5.3806
480	0.8463	0.2772	0.1362	2.0342	6.2115
512	1.2852	0.4540	0.2044	2.2213	6.2884
640	2.4994	0.8280	0.6106	1.3560	4.0933
720	2.9814	0.9706	0.7355	1.3195	4.0533
800	4.4947	1.4029	1.1412	1.2293	3.9386
1024	17.6088	4.4478	3.3742	1.3182	5.2187
2048	217.2149	64.9183	48.7952	1.3304	4.4516

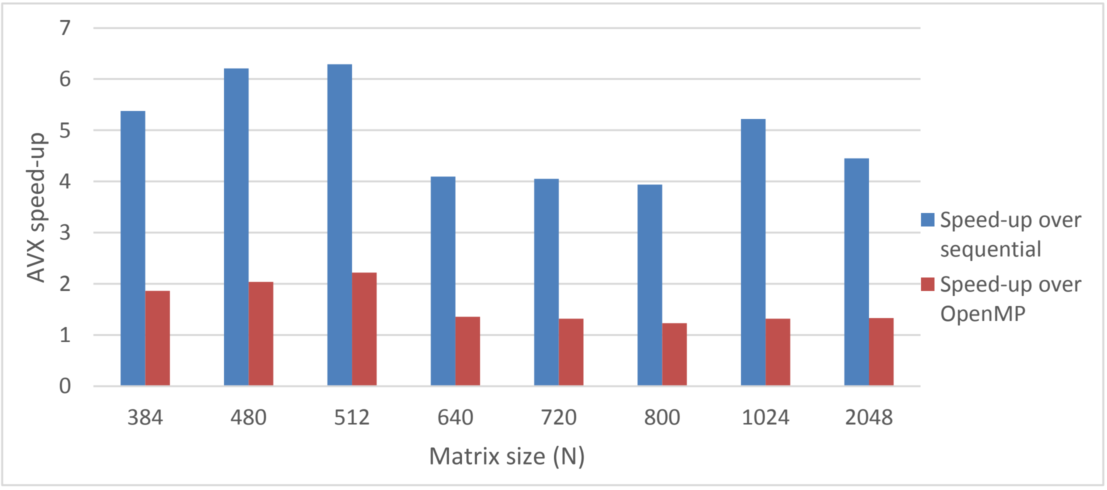

Figure 6. Computed speed-ups where the reference is the AVX with FMA implementation over parallel and sequential code

Conclusion

Due to the direction taken in modern computer architectures, an optimal code now requires the use of parallelization to efficiently use the capabilities provided by the hardware. However, this method requires thorough consideration.

As easy as it is to run C code in different threads through the use of OpenMP, one should also keep in mind the burden that comes included with this commodity. As seen from the results of the assignment, making the most of parallelized code involves a deep understanding of the underlying hardware, such as number of cores, threads and memory organization.

But not only is broader hardware knowledge needed, but also an understanding of the limitations of parallelization. In the results we could also see that the overhead caused by the creation and maintenance of more than one thread could actually have a negative impact on the performance.

As such, this method of improving performance can be a double-edged sword when thorough care is not given into the details of the underlying hardware and the application itself.