Maximum sub-array problem

Lab 1

Georgios Dikaros
Embedded Systems
KTH Royal Institute of Technology
Stockholm, Sweden

Ioannis Papakostas
Embedded Systems
KTH Royal Institute of Technology
Stockholm, Sweden

***Abstract*—For the first lab assignment of the course Parallel Computer Systems we implemented the parallelization of a serial algorithm for the solution of the Maximum sub-array problem. The problem is described as follows. [2] Given a 2-dimensional array of natural integers (between -32000 and +32000) the Maximum sub-array problem asks for the rectangular area within the array that maximizes the sum of the array elements found in the area.**

I.INTRODUCTION

To deal with the parallelization of the Maximum sub-array problem 's serial algorithm we used the Gothmog platform [2]. The platform consists of a four socket, 48 core shared memory multiprocessor, 32 GB of memory, 2 TB disk and is equipped with the Red Hat enterprise Linux version 6. We were required to create two implementations, one using pthreads or c++11 threads and the other using the OpenMP or Cilk Plus Frameworks or the TBB Library. The serial algorithm was provided at the beginning of the lab assignment as a reference for the students to use. The program is given an array as an input and it produces a solution to the Maximum sub-array problem for this specific array.

II.REQUIREMENTS

There were several requirements that needed to be satisfied for this lab assignment. First of all , the input array had to be of specific dimensions so that the serial execution time would be greater than 1 second. Moreover, it was required that we perform several experiments giving arrays of different sizes as inputs (thus increasing the workload) and also perform the same experiments using an increasing number of cores. Critical would be the determination of how the computational complexity changes with increasing workload and also how the program scales with increasing number of cores. In addition, we had to identify patterns to use on our parallelization and describe both the program and methodology, as well as the decomposition methods used. Finally, we had to present satisfying results from the program experiments and perform an analysis of performance and reach some important conclusions regarding the lab assignment.

III.IMPLEMENTATIONS

In this part we will describe the two main implementations of the parallelization of the serial algorithm for the solution of the Maximum sub-array problem. The complexity of the problem itself made us decide to work with the methods that we were more familiar with, thus choosing the c++11 threads and the OpenMP Framework. It was also clear that the map pattern [1][3]had to be used in order to deal with the parallelization of the reference implementation [4] because we needed to apply a function to each element of a collection of data items. Both implementations are described in detail below.

A. *Implementation of parallelization using the c++11 threads.*

One part of the reference implementation that was more than profound that parallelization could be applied to, was for the creation of the array ps[dim][dim]. However, deciding which is the most efficient way for parallelizing the kandane algorithm was quite obscure. Taking into consideration that the number of computations needed in order to calculate the result is approximately the same independent of the loop we choose to parallelize, we decided to create two basic code implementations in order to figure out ourselves which runs faster. Thus, at the first one we chose to parallelize the outer for loop (threads are assigned to different "i"s) whereas at the second one we used parallelization at the inner for loop of "k"s. We tried to make similar codes without big differences that could influence significantly their execution times and we found that the first one with the parallelization at the outer for loop was much faster. So we proceeded with further optimization of the first code in order to achieve an even better speedup. To avoid significant load imbalance we mapped the rows of the array we used as input to the threads as equally as possible. For example, in our implementation using 48 cores, the first thread (thread 0) is responsible for calculating the partial result that corresponds to rows 0, 48, 96, etc…In a similar way, thread 1 makes the computations for rows 1, 49, 97, etc. We did this because many more computations are needed for the first rows compared to the last ones. Another thing that was crucial as far as the execution is concerned was the use of a mutex in order to protect the variables max_sum, top, left, bottom, right. One way to avoid this was to create these variables separately for each thread and to compute the total max once the kandane algorithm had finished. However, we noticed that the use of a mutex inside the kandane algorithm led to almost the same execution time as without using it, so we chose to include it in our code.

B. *Implementation of parallelization using the OpenMP Framework.*

The same kind of doubt that we faced during the first implementation had once more to be overcome. It was once again impossible to identify whether it would be more efficient to apply parallelization at the outer or the inner loop. So we created again two implementations to find out which is the fastest among the two. It came out that using parallelization at the inner loop (for k=i;k<dim;k++;) leads to an approximately two times faster code. Our task was easier comparatively to the implementation with the C++11 threads since we neither had to create a separate function that was called by the threads nor we had to map the elements of the array used as input to them. We just had to include in our code the omp library and use two directives that perform all this work automatically. The first one is the #pragma omp parallel directive which is used to define the point from which parallel execution starts and the second one is #pragma omp for which is used to define which for loop will be executed in parallel. The number of threads created each time can be defined with the num_threads() clause placed right next to the #pragma omp parallel directive. Although OpenMp offers precious help and saves us a lot of time, one thing we have to be extremely careful with is the declaration of the various variables used by the threads. Thus, in our implementation some variables like i,j,sum,local_max had to be declared private in order their values not to be changed by other threads performing computations using the same variables. Furthermore some other variables that their values was desirable to be changed were declared shared and in order to avoid conflicts and miscalculations we chose to protect them using a #pragma omp critical directive which is similar to what we did at the first implementation. The trivial delay that the use of the mutex introduced to the execution time was decisive so as to keep it in our code and not to calculate the total max in a different way. Finally the array ps, which was also calculated in a parallel way just like it was done in our previous implementation, was declared firstprivate. This way the values of its elements as those were computed before the kandane algorithm are available to the threads which use them to perform their calculations.

IV.EXPERIMENTAL RESULTS

For the experimental part of the assignment me and my lab partner were given several alternatives. Using the Gothmog platform we could choose to work with up to 48 cores and we were also given a program for generating random arrays of specified dimensions. We chose to create four square arrays of random integer elements with dimensions of 500, 750, 1000 and 2000. Furthermore, we chose to work with four different numbers of cores varying from 4 to 48 (4, 16, 32, 48) as we can specify the number of created threads both in c++11 threads and OpenMP. All experiments were conducted through compiling our programs using the provided makefile by configuring its fields both according to the method we used as well as the input array for each experiment. We configured CC = gcc , CFLAGS = -std=c99 for the serial execution, CC = g++ , CFLAGS = -std=c++11 , LFLAGS = -lpthread for c++11 threads and last CC = gcc, CFLAGS = -fopenmp for OpenMP. Each experiment was conducted ten times and we took the median value for better accuracy. The results of all experiments are presented in the following tables where the first column represents the input array, the second column represents the time in serial execution, the third column represents execution time for c++11 threads and the fourth column represents execution time for OpenMP.

TABLE I. 48 CORES

Input	***Serial***	***C++11***	***OpenMP***
500	**2.349**	**0.096**	**0.076**
750	**7.938**	**0.211**	**0.231**
1000	**18.732**	**0.432**	**0.45**
2000	**149.474**	**3.188**	**3.618**

TABLE II. 32 CORES

Input	***Serial***	***C++11***	***OpenMP***
500	**2.349**	**0.1**	**0.097**
750	**7.938**	**0.284**	**0.278**
1000	**18.732**	**0.619**	**0.624**
2000	**149.474**	**4.7**	**4.73**

TABLE III. 32 CORES

Input	*Serial*	*C++11*	*OpenMP*
500	**2.349**	**0.166**	**0.162**
750	**7.938**	**0.543**	**0.523**
1000	**18.732**	**1.187**	**1.2**
2000	**149.474**	**9.323**	**9.415**

TABLE IV. 4 CORES

Input	*Serial*	*C++11*	*OpenMP*
500	**2.349**	**0.6**	**0.599**
750	**7.938**	**1.995**	**2.02**
1000	**18.732**	**4.656**	**4.709**
2000	**149.474**	**37.097**	**37.443**

V.CONCLUSIONS

In this lab assignment we parallelized a serial program to improve its speed while maintaining its reliability and providing accurate results. Before conducting the experiments we expected the values of speedup for the parallelized implementations to be at most at the level of used cores. Of course, such thing would be practically impossible because the speedup [1] is constrained by a number of factors such as the serial part of execution, the load imbalance that is encountered in many occasions because the number of iterations is not equally divided to threads and also the resource contention. The results of our experiments show that we nearly achieved the hypothetical speedup in most occasions and that was expected because the vast majority of the program parts were parallelized. We also witness how much the computational complexity rises with the increasing workload as we can see that for the serial implementation the execution time is increased exponentially. On the other hand, in the parallel implementations the execution time stays relatively low even for the largest input array, with the exception of the experiment with the four cores. Moreover, both the c++11 threads implementation and the OpenMP implementation have an almost equal rate of speedup though the OpenMP is much easier to use and requires minimal programming effort. In conclusion, through this lab assignment we were able to figure out the extent of speedup at which parallelization can lead to even for very complex computational problems.

REFERENCES

[1] Structured Parallel Programming by Michael McCool, James Reinders, Arch Robison

[2] https://www.kth.se/social/course/IS2200/

[3] http://software.intel.com/en-us/blogs/2009/06/10/parallel-patterns-3-map

[4] http://stackoverflow.com/questions/2643908/getting-the-submatrix-with-maximum-sum