 Departamento de Engenharia Informática

Mestrado em Engenharia Informática e de Computadores – Alameda

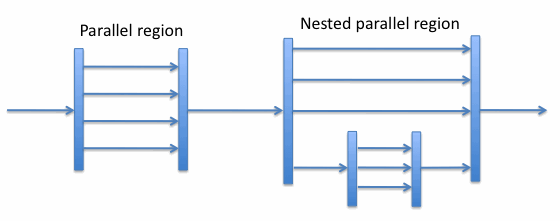
Computação Paralela e Distribuída

*Report*

**Longest Common Subsequence**

1st Stage



****

Teachers: José Monteiro

José Costa

**Group 34**

Fábio Antunes 66979

João Seixas 69746

Jorge Macedo 70035

1st Semester

2014/2015

1. **Introduction**

One of the greatest challenges in Computer Science nowadays is to make computationally viable solutions, which reduce the processing time and supply even more precise answers. Often there are proposals with new ways to solve these problems, or improvements to the existing solutions. This need is due to the fact that today, clock speed have pretty much stalled due to the self-deterioration that the heat of the high frequencies causes on itself. Taking this into consideration, the effort has been made in the direction of having more cores, in order to achieve better computation power. To do so, it’s required that the programmer is able to take advantage of this architecture, by having its code running in parallel, whenever possible.

This first work, intends, to get to know coding techniques that allow parallelization of work, using the OpenMP library, which allows the creation and management of threads running on multiple cores, with ease.

1. **Parallel Implementation – OpenMP**

The approach we used for parallelization, mainly consists of pipelining. As matrix cells have lots of dependencies between themselves, we opted for this solution, which thus involved some synchronization, it seemed adequate to the problem at hand. Also this solution addressed the problem of optimizing cache misses because the matrix is processed line by line. When the thread starts reading the first cell, the whole line is cached for easier access, if we were handling the matrix column-wise, the time would be much bigger due to many cache misses.

Like the image shows, each thread is assigned one line on the matrix, since cells depend of the cell directly on top of it, and the one on the top-left side, some synchronization has to be done in order for everything to run in parallel

* **Decomposition**

The decomposition of the matrix we chose, to tackle this problem, as previously seen, was the division of the matrix in lines. We took this approach to minimize cache misses and because it was the one more suited for pipelining considering the dependencies between cells.

* **Synchronization concerns**

The main synchronization concerns were to keep threads from stepping in front of each other. Like previously stated, the dependencies among the cells, causes for the program to have a lot of synchronization. In order to achieve this, every cell in the matrix starts by being locked (each threads locks the cells that it’s going to process). Then the first thread starts processing the first line (that has no dependencies), and once it finishes processing each cell, this cell’s lock is opened. The immediately following thread, was blocked, trying to access this position, and now it can, and will do the same thing the previous did, so the next thread can start processing as well.

We had some problems that lead us to this solution. In early development stages, we weren’t locking the entire matrix, which would lead to sometimes threads to step ahead of others, and read data that was unprocessed, thus ruining the process of the longest common substring. This would happen due to cell processing time not always being the same (for example the top lines have more zeros, which makes their processing faster), and the operative system can, at any time, change the CPU to other processes.

* ***Load-Balancing***

On our solution load balancing was taken very much into consideration. Since we wanted to evenly distribute the workload among the threads, and since the first lines in the matrix take less time to process (due to more zeros) than the following ones, we developed a solution that addressed these issues. By having the loop that goes through the matrix running in parallel with a schedule static of 1, we assign each line to a thread. Since threads are dependent of the threads immediately above, and these are quicker to process, there is a perfect load balance to keep everything working.

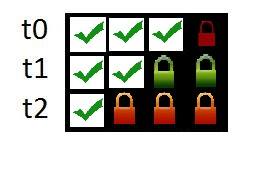


Figure 1: Scheme of threads processing

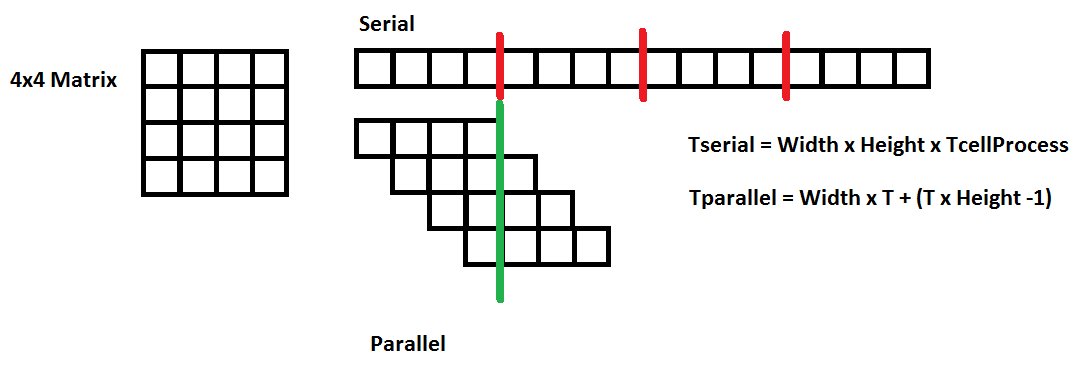
1. **Perfomance Results**

In order to perform the following tests, computer in RNL were used, in Pavilhão de Informática 1.

The tests correspond to the examples given in the course page: ex10.15.in, ex150.200.in, ex3k.8k.in, ex18k.17k.in, ex48k.30k.in.





Here we have a simple explanation on how we are achieving the speedups, and what would be the best theoretical speedup we could get. Being T, the time it takes to process each cell, our theoretically best time would be Width \* T + ( T \* (Height – 1)). In practice T value changes from cell to cell, which causes some hurdles about knowing its value.

Taking into consideration the tables presented before, we can take some conclusions. First of all is that it didn’t take that much of an input for the parallel code to start outperforming the serial code, although, as expected, with small inputs, the performance is pretty much even.

On the cases where the number of threads is bigger than the number of cores, as expected, some overhead was created due to the operative system having to scale CPU time between threads. This was most visible especially with small inputs, getting even worse times than the serial version.

Also, as expected, the optimum number of threads turned out to be the number of cores of the computer, and with that we achieved our best speedup of 3.37, and an overall speedup of 3.03 (disregarding the smallest input)

Below is the comparative graphs of execution times for these last three tests.