Departamento de Engenharia Informática

Mestrado em Engenharia Informática e de Computadores – Alameda

Computação Paralela e Distribuída

Report

Longest Common Subsequence

2nd Stage

Teachers: José Monteiro
José Costa

Group 34

Fábio Antunes	66979
João Seixas	69746
Jorge Macedo	70035

1st Semester
2014/2015

I. Introduction

On this project, the main difficulties related with the parallelization came from the fact that each position of the matrix is heavily dependent on the ones surrounding it. As such, we thought of a way to reduce these dependencies, while keeping the same algorithm by having a different look at the matrix itself.

We have decided not just to process the matrix diagonally but also to make the lines of the new matrix, to be those same diagonals. And so we developed a way to make a new matrix with these restrictions, given the regular matrix parsed from the file. This approach made sense for us because now all the dependencies will be on the upper lines, leaving matrix cells in the same line free for parallel processing, benefiting the interweaving between MPI and OpenMP. Figure 1 and 2 show the dependencies of the cells in the normal matrix, and the comparison of both the matrix (how one is achieved from the other).

II. Parallel Implementation - MPI

For each line of the new matrix, each process starts by allocating space to store its part of the line. Then the root process broadcasts the two lines that are immediately above the one that the program is about to process.

Given this, all the processes have the necessary information to process their own part of the line (because with the new matrix, all the dependencies are on the two lines immediately above). This can be done in parallel, and even if their part of the line is composed of several matrix cells (which typically happens) these cells can also be processed in a parallel fashion. When each process finishes processing the part of the line assigned to it, they send those same parts back to the root process so it can build the line from each part given by each process (using MPI_Gatherv). Then they all move to the next line and repeat the process.

Considering the approach that we use with the new matrix, the parallelization of the processing of the matrix is a trivial issue since a line can be shared by multiple processes due to the fact that there are no dependencies between cells of the same line and thus process don't have to communicate with each other (something that we took very much into consideration, in order not to cause much overhead to the shared bus of the processes if we are considering a NUMA architecture or simply avoid unnecessary waits from threads in a UMA architecture).

The process itself is very simple. For each line of the matrix, each process is assigned a part of that line, and they all can work at the same time on the same line. When the processes are done, they send back their work to the root process, and move on to the next line, using the information they had already process.

➔ Decomposition

Since we had to maintain the same algorithm, even with a different view of the matrix, we had to adapt the way we were processing each position of the matrix and also the backtracking mechanism required to achieve the Longest Common Subsequence.

In the old approach it would be trivial to know the cells that had dependencies on each other because it would always be the same. The left one, the top one and the diagonally left and top one. As you can see on the first matrix of Figure 1, position 6 should depend on positions 5, 2

and 1. Given the matrix on the Figure 2 on a given cell, knowing the positions that it depends on isn't a trivial task. To address this issue, we devised an algorithm that allowed us to know, in a deterministic way, which were the cells that the current one depended one, needing only the line and the position in the current line, of the cell we wanted to know.

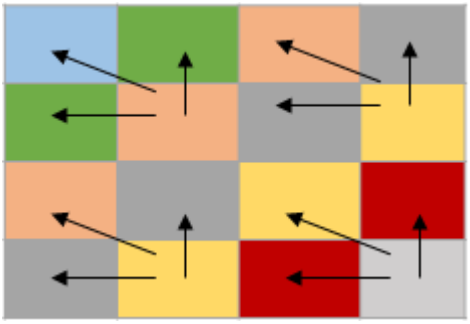


Figure 1 – Dependencies between cells.

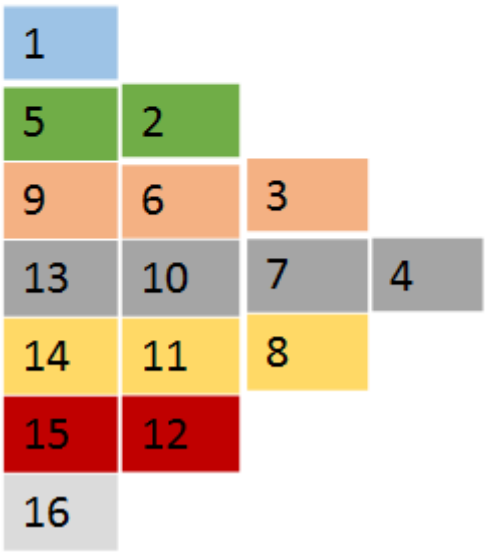
➔ Synchronization concerns

On our new approach, the only concern regarding synchronization is just that the program can only start process the following line, after all processes have finished processing their own parts of the current line (since there are dependencies with the previous lines). This small issue was taken care of automatically and without effort, by just iterating through the matrix line by line and on each line, having the processes doing their tasks.

➔ *Load-Balancing*

Once we divided the line for the processes, we only had to take into account that each process would get the same amount of part of line as the others. Basically, we take the number of existing processes and split the line among them, having each process assigned with the same amount, and thus guaranteeing an even load balancing between the processes.

➔ *MPI + OMP*



With this new approach we devised, it was pretty simple to add and interweave OpenMP with MPI.

In this case, since each process would go through a part of line and the positions of the lines are independent of one another, we just made that part parallel, so that each process can run the cells concurrently.

When a line is divided by the process, each process creates threads that will process each cell of that part of line, concurrently.

III. Perfomance Results

The performance of the program wasn't nearly as good as expected. For this, many reasons have contributed, but the main reason was the fact that the condor system, provided by the course to run the program, wasn't functioning very well and it was a major hurdle when we wanted to test the program. We had the program finished two weeks before the deadline, and in those two weeks, we tried and failed many times to run the program on the condor cluster. The only time when we could ran, and where we actually got the results that we are going to show, was one day at 5 a.m., because all other times the system would always fail to run the jobs, or when it did ran, it would have some kind of error like failing to opening a file, or simply not delivering output (some Shadow Exceptions found in the condor log). But, it wasn't the only reason for the performance to have been poor. Our algorithm relies on the division of the line into pieces that are to be scattered around the several processes, while this is great when handling big lines, it can produce quite a bit of overhead with smaller ones. Another possible cause for overhead could be the consecutive cache misses since each line depends on the two lines above it.

MPI + OMP

MPI+OMP	ex10.15.in	ex150.200.in	ex3k.8k.in	ex18k.17k.in	ex48k.30k.in
SERIAL	0,001	0,011	8,161	105,509	488,874
2 Processes	0,0012493	0,0018615	0,0205796	15,003412	72,557321
4 Processes	0,0010345	0,0019826	0,0195708	25,335976	102,254984
8 Processes	0,0020942	0,0042352	3,0306451	41,003412	172,356412

On this table, it's clear to see that some results don't make much sense with the rest, namely the ones in ex3k.8k.in (2 Processes and 4 Processes). These were the values that we got on condor, after running once (we couldn't run them any more time), but they can't be correct since they do not fit with the rest of the times.

MPI + OMP – Speedups

Speedups	ex10.15.in	ex150.200.in	ex3k.8k.in	ex18k.17k.in	ex48k.30k.in
2 Processes	0,80044825	5,909213	396,557756	7,03233371	6,737762548
4 Processes	0,96665056	5,54826995	416,998794	4,16439454	4,780930776
8 Processes	0,47750931	2,59727994	2,69282603	2,57317611	2,836413188

Our best speedup (ignoring the ones that are absurd – 396 and 416) is 7.03, which in theory is very small. Also from this data, we can access another problem with our solution that is the scalability. With MPI and OpenMP the speedup should increase (for big inputs), with the number of processes, and this is not what we see here. This can be explained by the same reason as in the overview. When the line is split among processes, each process runs each cell concurrently. For some line sizes, having more processes would be beneficial, but, as seen here, it's only introducing unnecessary overhead.

MPI

MPI	ex10.15.in	ex150.200.in	ex3k.8k.in	ex18k.17k.in	ex48k.30k.in
SERIAL	0,001	0,011	8,161	105,509	488,874
2 Processes	0,014345	0,070941	6,938155	75,637734	339,083737
4 Processes	0,02043	0,103449	6,725674	60,753916	259,517766
8 Processes	0,04171	0,130219	6,835193	56,705548	217,269473

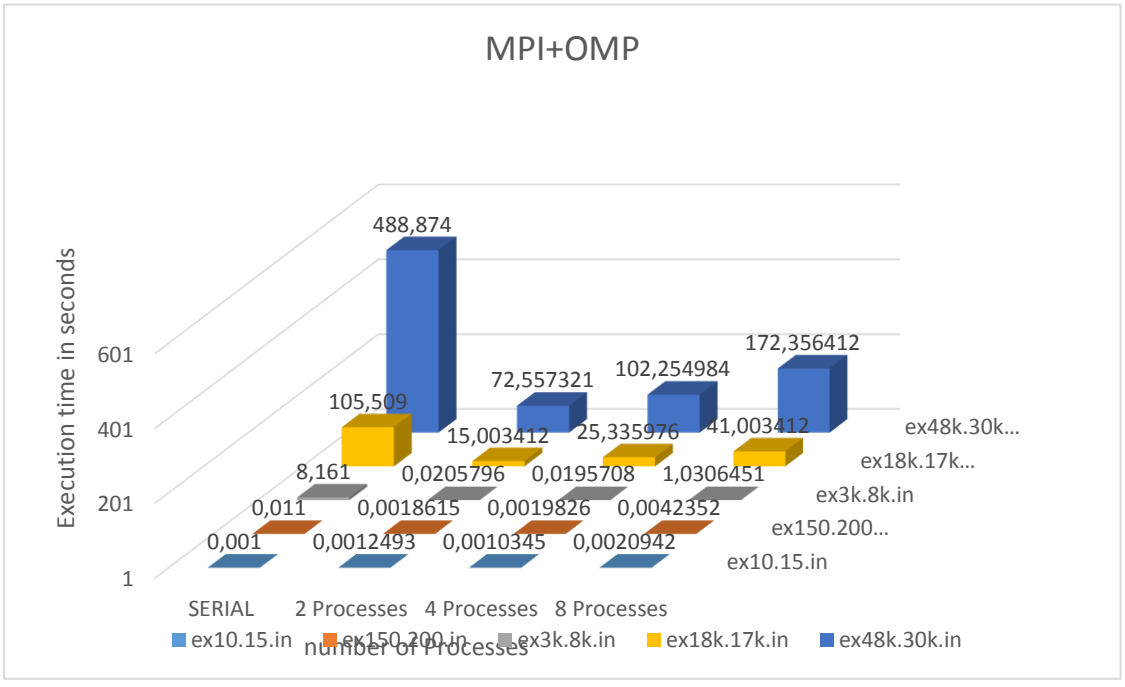
These results were taking using mpirun with an hostfile file, that would point to 8 pc's in Lab13. This was the way we found in order to run more times, since the condor cluster was mostly unavailable. These results, comparing to the previous ones, are slower, which makes sense since on the other we have two levels of parallelism.

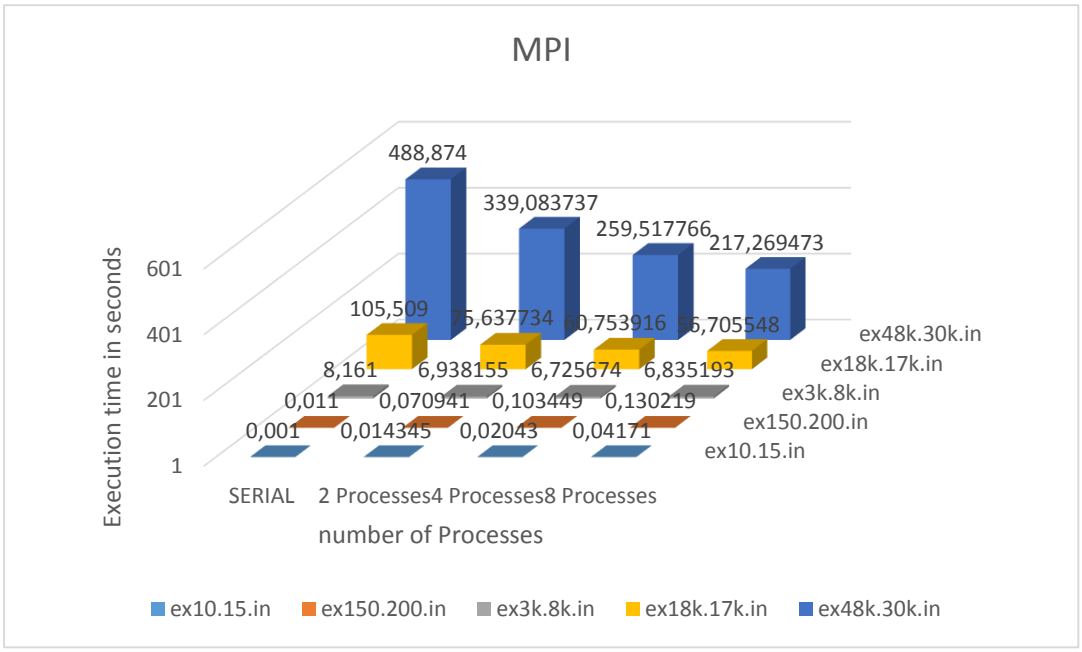
MPI – Speedups

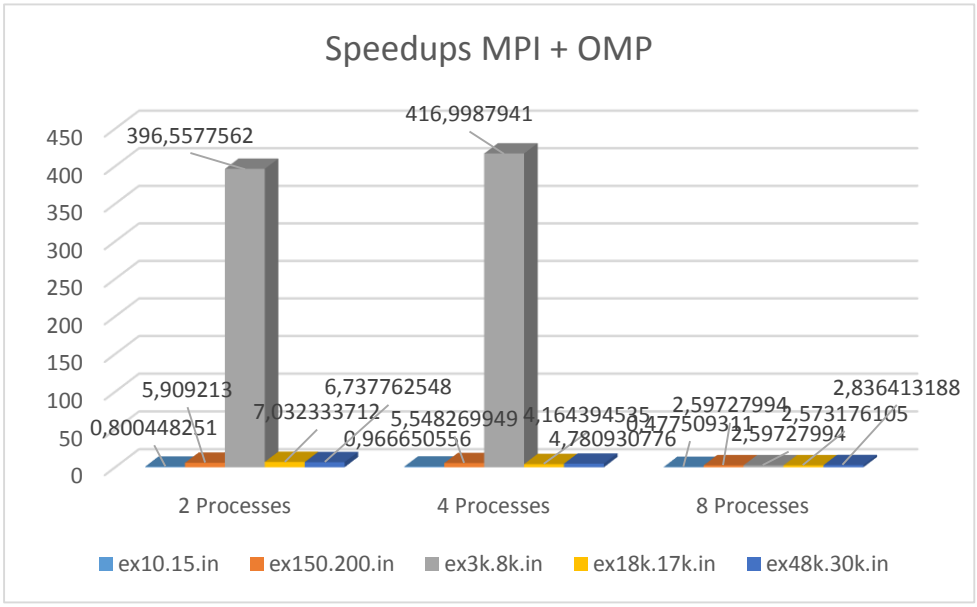
Speedups	ex10.15.in	ex150.200.in	ex3k.8k.in	ex18k.17k.in	ex48k.30k.in
2 Processes	0,0697107	0,15505843	1,17624931	1,39492545	1,441750065
4 Processes	0,04894763	0,10633259	1,21340999	1,73666172	1,883778546
8 Processes	0,02397507	0,08447308	1,19396775	1,86064686	2,2500814

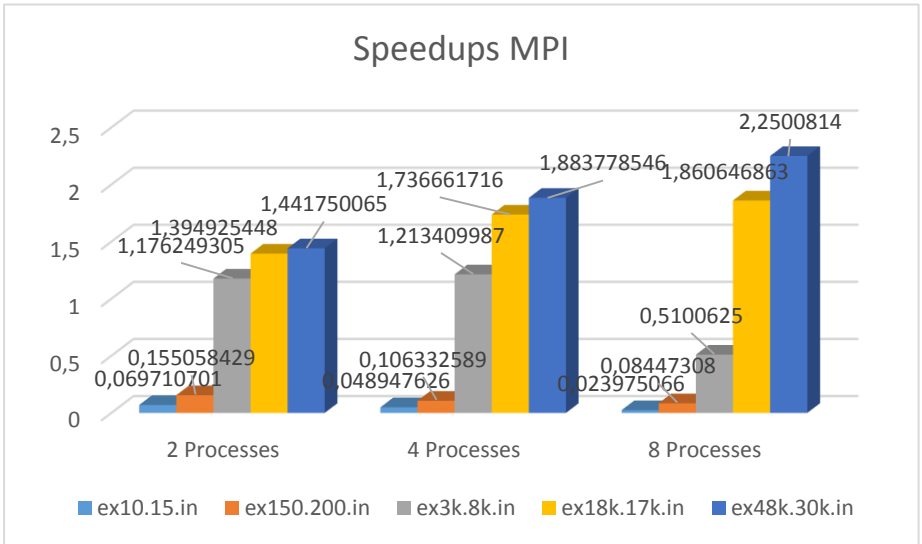
With MPI only, our speedups fell very short from our objective. The reason for this, is the same as explained before (with the division of lines) but even more with the overhead that each cell is done sequentially. One upside of these results is that for big enough inputs, the speedup scales with the number of processes (something that wasn't achieved previously).

Here we show some graphs presenting the information shown on the tables:




MPI
Execution time in seconds
601
401
201
1
488,874
339,083737
259,517766
217,269473
105,509
75,637734
60,753916
56,705548
8,161
6,938155
6,725674
6,835193
0,011
0,070941
0,103449
0,130219
0,001
0,014345
0,02043
0,04171
ex48k.30k.in
ex18k.17k.in
ex3k.8k.in
ex150.200.in
ex10.15.in
SERIAL
2 Processes
4 Processes
8 Processes
number of Processes
ex10.15.in
ex150.200.in
ex3k.8k.in
ex18k.17k.in
ex48k.30k.in


Speedups MPI + OMP
450
400
350
300
250
200
150
100
50
0
396,5577562
416,9987941
5,909213
0,800448251
6,737762548
7,032333712
5,548269949
0,966650556
4,164394535
4,780930776
2,59727994
0,477509311
2,573176105
2,59727994
2,836413188
2 Processes
4 Processes
8 Processes
ex10.15.in
ex150.200.in
ex3k.8k.in
ex18k.17k.in
ex48k.30k.in


Speedups MPI
2,5
2
1,5
1
0,5
0
2,2500814
1,883778546
1,860646863
1,736661716
1,441750065
1,394925448
1,213409987
1,176249305
0,5100625
0,155058429
0,106332589
0,08447308
0,069710701
0,048947626
0,023975066
2 Processes
4 Processes
8 Processes
ex10.15.in
ex150.200.in
ex3k.8k.in
ex18k.17k.in
ex48k.30k.in