DEEC
DEPARTAMENTO DE ENGENHARIA ELETROTÉCNICA E DE COMPUTADORES
TÉCNICO LISBOA

Parallel and Distributed Computing

10/31/2014

Longest Common Subsequence

1st Part

Daniel Arrais, 69675
Miguel Nobre da Costa, 73359
Ricardo Amendoeira, 73373
PROFESSOR JOSÉ COSTA

Introduction

In this project, the aim was to compute the longest common subsequence (LCS) of two given strings. A subsequence is a sequence of zero or more elements left out of a given string. In this problem (LCS), we try to compute the maximum length common string between the two strings. This is a problem which has applications in many areas, such as comparison between DNA of two different organisms. The longer the common subsequence, the more likely is for the two organisms to be similar.

Serial Implementation

In order to compute the LCS, first we must read both strings, which are given in a file. The file is decoded, saving the strings in an allocated vector with the corresponding string size. Being M and N the string sizes, we can then allocate a matrix with (M+1) by (N+1) entries in which we can then run an algorithm to evaluate the longest common subsequence. By allocating a matrix this size, we can make a direct correspondence between each line and column to the different letters of the string.

This algorithm, inside function *computeMatrix*, uses an iteration method for computing each entry of the matrix. It does so according to the equation in the Project Assignment, part 2.1. Let *i* and *j* be the indexes of the matrix and x_i be the corresponding letter of *i* line and y_j be the letter corresponding to column *j*. If both *i* and *j* are zero, the entry is zero; if *i* and *j* are greater than zero and x_i is equal to y_j, then we add what function *cost* returns to the number of entry [*i-1*, *j-1*], meaning we found a match between the two strings; finally, if *i* and *j* are greater than zero and x_i is different to y_j, we simply find the maximum value between entries [*i*, *j-1*] and [*i-1*, *j*], meaning there is a mismatch and we must consider the best subsequence found so far. By the end of this algorithm, the length of the longest common subsequence can be found at position [*m*, *n*]. At this point, in order to find the actual LCS, we must traverse the matrix backwards starting from this entry.

This is done in the function *computeLCS*. Starting from position [*m*, *n*], we check whether there is a match, and then we go back diagonally in the matrix, and if they are different, we go to the entry with the bigger value or left if they are equal to one another. While doing this, each letter is saved in an array that functions as a stack.

Finally, all the allocated memory is freed and both length and actual subsequence is showed to the user. We can conclude that the complexity of this algorithm is $O(m \times n)$.

Parallel Implementation: OpenMP

In order to parallelize our serial implementation, we started by checking which part of our program took longer to compute. Using the tool GNU gprof, we found out that the matrix calculation was, by far, what took the most time. We then proceeded to parallelize function *computeMatrix*. We quickly understood that by simple parallelizing the *for* loops wasn´t good enough, because of the many data dependencies between an element and the preceding ones (left, up and left-up-diagonal). We then thought how we could improve by computing the matrix through its anti-diagonals. By computing each entry in this way, we guarantee that the matrix integrity is kept and therefore all the computational results are correct. Although this method may increase a certain unbalance in thread distribution and

consequential overhead, it causes less overhead than by simply parallelizing the cycles *for*, in which we would need to lock cells and lines.

Results and Conclusions

Through this project we can easily understand how much benefits there are from parallelizing a certain code. This is even more noticeable when there is a huge computational cost (i.e. computing a big matrix), which can be divided by splitting the work among the existing cores. By using the ompP profiler, we concluded that our parallel section takes 99.99% of all the computational work, but there is an overhead which is caused by thread imbalance. However, if we examine the times that each thread spends on the parallel regions, these are all very similar. We tried to solve this problem using different schedules and chunk sizes, but nothing seemed to improve the overhead problem.

The following table shows the average times we got by executing the program in serial and parallel versions, alongside the corresponding speedups for all the experimental instances provided by the professors. The parallel results are divided by the amount of cores used (2, 3 and 4). All times results are displayed in seconds and were obtained using the function *omp_get_wtime()* in the Laboratory PC's (Intel Core i5 760 – Quadcore, 8GB RAM). The speedup corresponds to the relation between serial and parallel version:

$$S = t_{serial}/t_{parallel}$$

Instance	Serial Version	Parallel Version					
		2 Cores	Speedup	3 Cores	Speedup	4 Cores	Speedup
ex10.15.in	0.0002	0.0002	1	0.0002	1	0.0002	1
ex15.200.in	0.011	0.006	1.83	0.0048	2.29	0.004	2.75
ex3k.8k.in	8.11	4.44	1.83	3.04	2.67	2.54	3.2
ex18k.17.in	103.53	56.35	1.84	37.51	2.76	32.51	3.2
ex48k.30k.in	478.29	281.02	1.7	178.76	2.68	139.28	3.43

We would also want to report that we tested our program in different computers and, in some cases (i.e. our personal computers) the speedups obtained were low (around 2,9 for 4 cores). We believe that this disparity of values may be caused by different access memory bandwidth and cache sizes.