DEEC
DEPARTAMENTO DE ENGENHARIA ELETROTÉCNICA E DE COMPUTADORES
TÉCNICO LISBOA

Parallel and Distributed Computing

5/12/2014

Longest Common Subsequence

2nd part

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

Introduction

In the first part of this project we used OpenMP to parallelize our serial solution to the Longest Common Subsequence (LCS) problem using one machine with multiple cores. The aim of the second part is to use the Message Passing Interface, MPI, another method of program parallelization: using a cluster of multiple machines (although it can run on a single machine with multiple or even only a single processor) running independent processes with no shared data on each of their processors and achieving parallelization by passing messages between them with the required data. OpenMP can be integrated with MPI to make use of the multiple cores of each processor and further improving the performance of the parallel implementation.

Serial Implementation

The serial implementation used was the one delivered in the 1st part of the project, as the problem to be solved is the same, please refer to that report for a detailed explanation of the implementation.

Parallel Implementation: MPI

One big problem arises when working with multiple machines: the lack of shared memory. Using the LCS problem as an example, if one process computes a part of the matrix, this change isn't accessible to the other processes and there's no trivial way to solve this complication without shared memory.

The Message Passing Interface (MPI) provides methods for processes to communicate and transfer data between them, allowing for one-to-one communication as well as some more complex forms like scatter, combine and others, and even process synchronization. Using MPI we can solve the complication mentioned earlier and share the work done by one process with the others. It comes, however, with the downside of adding communication overhead that can be significant, depending on the implementation. Besides that, MPI also has the advantage of instead storing all the data in a single machine, which may require a more complex machine with more memory, it can be spread out by many simpler machines

In our MPI implementation, generally, the lines of the matrix are divided by the number of processes. However, this division is not always an exact integer (N may not be divisible by p) and, in this cases, every process gets a chunk of size $(N/p) + 1$ except for the last one that gets the remainder.

Because of the data dependencies it's not possible to start solving all line blocks at the same time (each cell depends on its left, upper, and upper-left cell), causing one process to wait for the previous one to compute its data before it can begin.

To solve this problem, we decided to proceed in the following way: the process 0 computes the values of the column 1 from its chunk. The last value is then sent to process 1, allowing it to start computing the values of column 1 from its chunk. As process 1 starts, process 0 continues to compute the values of the next column and so forth. The idea behind is for a process to wait for the value sent from the previous process to then compute its column and then send its last calculated value to the next process.

After computing all the matrix values, the process with $id = p - 1$ has the actual size of the LCS and sends it to process 0. The last process starts to traverse its own matrix backwards and, every time there is a match, he stores the character on a stack. As he reaches the first line of its matrix, he proceeds to send the corresponding column where he stopped (column j_1) to all processes as well as sending the already computed part of the LCS to process 0. The previous process can then start to compute the next part of the LCS starting from column j_1 until either reaches column 0 or reaches the first line of its own matrix, where it sends the corresponding column to all processes. This happens until the LCS is computed (by reaching $j = 0$ or, in the case of process 0, $i = 0$).

Performance results

The test results shown were calculated using only the *mpirun* method using a host file with the Lab13 computers. Condor functionalities were unavailable at the time the tests were conducted. This means that some instances were run while the others computers were being used, which may imply some delays in the matrix calculation.

The results, for the instances provided, were as follows:

Instances	**Serial**	**Parallel – MPI**					
		2	S	3	S	4	S
ex10.15.in	0.002	0.39	-	0.17	-	0.33	-
ex150.200.in	0.011	0.42	-	0.18	-	0.31	-
ex3k.8k.in	8.11	4.64	1.75	4.43	1.83	2.85	2.85
ex18k.17k.in	105.85	56.40	1.88	41.07	2.58	28.33	3.74
ex48k.30k.in	491.10	267.62	1.83	176.51	2.78	133.25	3.7

We can notice from the table above that MPI has no applicability for problems with small amount of data as the communication overhead is too high. As the amount of data gets larger, the communication time becomes irrelevant in comparison to the computation time. Consequently, for bigger instances the speedup reaches our estimated values.

Parallel Implementation: MPI+OMP

We tried to implement a version with MPI and OMP combined but, unfortunately, although it presents the correct outputs the speedups were not desirable. However, this version differs from the previous in terms of work division.

The lines are divided for each process as before. However, at each iteration, instead of computing just one column of the matrix, each process computes a block of the matrix. Whenever M is divisible by p, each block has a width of $M/(log(M) \times p)$. When it isn't, all blocks except the last are of width $\frac{M}{log(M) \times p} + 1$ and the last gets the remainder. Because each process now works with a block of data at each iteration, the last line of the computed block must now be sent to the following process so it can start to compute its own block. This method is repeated along all the matrix.

MPI vs. OMP

Although the speedup results are very similar, MPI proved to be a little better. To conclude, we arranged this table with the advantages and disadvantages of each parallelization method:

Topic	MPI	OMP
Scalability	Better	Worst
Programming	Harder	Easier
Applicability	Wider	Narrower
Memory Architecture	Distributed/Shared	Shared
Control of Memory	Easier	Harder

Results and Conclusions

With the first part of the project we learned, by using openMP, the benefits of parallelizing programs in a single machine. In this second part we went further by using MPI to parallelize our program in a cluster of machines. Although it is more complex, it allows us to use more computational power than one machine can provide and improve performance further. Also allows the usage of many simpler machines instead of one more complex machine, if the bigger limitation is the amount of memory needed.