 Parallel and Distributed Computing

12/4/2014

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

Professor josé costa

Longest Common Subsequence

2nd part

12/4/2014

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

Professor josé costa

Longest Common Subsequence

2nd part

# 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 (refer to that report for a detailed explanation of the implementation). As a summary, the complexity of the algorithm used is and uses a little over bytes of memory ( and are the lengths of the two provided strings).

# 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 ways 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. MPI also has another big advantage, instead of 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, like raspberry PI’s (depending on the aim of the program, can have multiple advantages compared to using a normal x86 based computer). It comes, however, with the downside of adding communication overhead that can be significant, depending on implementation, so too much communication should be avoided.

Our MPI implementation to discover the LCS of two strings divides the matrix in P blocks of horizontal lines, where is the number of lines of the matrix and the number of processes. Each process gets assigned its own block of lines, which it solves in squares of height and width. For some matrix sizes and/or number of machines isn’t an integer value and the remainder blocks are handled by the first/last process. This work division leads to more work imbalance overhead than MPI communication overhead, since each process can compute a fairly large amount of data before requiring access to data from another processor.

Because of the calculation dependencies (each anti-diagonal of the matrix depends on the values of the previous anti-diagonal) it’s not possible to start solving all line blocks at the same time (the first block of the last process depends on the last block of the first process, for example), causing the last process to wait for the first to solve its whole line before it can begin, slowing down the program. The solution found was to continuar.

Just like in out OpenMP implementation the vast majority of the execution time is on the function *computeMatrix*, so that’s the only part that we chose to parallelize.

# 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 and has more overhead it does allow us to use more computational power for our program 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 amount of memory.

Although not a direct consequence of using MPI, we did get some better speedups than our implementation in part 1. The following table shows our execution times using the RNL cluster and the corresponding speedups for the