TÉCNICO LISBOA

Parallel and Distributed Computing

Project Assignment

LONGEST COMMON SUBSEQUENCE

Version 1.0 (27/09/2014)

2014/2015

1st Semester

Contents

Revisions

Version 1.0 (September 27th, 2014)	Initial Version

1 Introduction

The purpose of this class project is to gain experience in parallel programming on an SMP and a multicomputer, using OpenMP and MPI, respectively. For this assignment you are to write a sequential and two parallel implementations of a Longest Common Subsequence algorithm.

2 Problem Description

A subsequence of a given sequence is just the given sequence with zero or more elements left out. Given two sequences X and Y , we say that a sequence Z is a common subsequence of X and Y if Z is a subsequence of both X and Y. For example, if $X = \{A, B, C, B, D, A, B\}$ and $Y = \{B, D, C, A, B, A\}$, the sequence $\{B, C, A\}$ is a common subsequence of both X and Y.

In the longest-common-subsequence (LCS) problem, we are given two sequences $X = \{x_1, x_2, \ldots, x_m\}$ and $Y = \{y_1, y_2, \ldots, y_n\}$ and wish to find a maximum-length common subsequence of X and Y. For the case above, the maximum length is 4, for example $\{B, D, A, B\}$.

This is a general problem that has applications in many areas. For example, in biological applications, we often want to compare the DNA of two (or more) different organisms. A strand of DNA consists of a string of molecules called bases, where the possible bases are adenine, guanine, cytosine, and thymine. Representing each of these bases by their initial letters, a strand of DNA can be expressed as a string over the finite set A, C, G, T. One goal of comparing two strands of DNA is to determine how "similar" the two strands are, as some measure of how closely related the two organisms are. Similarity can be and is defined in many different ways. One way to measure the similarity of strands S1 and S2 is by finding a third strand S3 in which the bases in S3 appear in each of S1 and S2; these bases must appear in the same order, but not necessarily consecutively. The longer the strand S3 we can find, the more similar S1 and S2 are.

2.1 Sketch of the Algorithm to Implement

The LCS problem is classical case of a problem that can be solved efficiently using dynamic programming. It can be shown that the following simple recursion computes the optimum solution to this problem.

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max\left(c[i, j-1], c[i-1, j]\right) & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

This basically computes the optimum LCS solution by considering one extra element of the input sequence at a time. If there is a match (second line of the system above), we increase the count. If there is a mismatch (third line), we consider the best subsequence found so far.

The length of the LCS is stored in position $c[m, n]$. In order to find the actual subsequence we need to traverse the matrix backwards, starting from this position:

- if there is a match go back diagonally;
- if not go up or left depending on which is larger (if equal it's indifferent – for this project we will move left).

2.2 Illustrative Example

To illustrate the algorithm above, the matrix that is computed for the instance presented before, $X = \{A, B, C, B, D, A, B\}$ and $Y = \{B, D, C, A, B, A\}$ is:

		0	1	2	3	4	5	6
			B	D	C	A	B	A
0		0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	3	3
5	D	0	1	2	2	2	3	3
6	A	0	1	2	2	3	3	4
7	B	0	1	2	2	3	4	4

In order to obtain the actual subsequence, we start at position (7,6). There is a mismatch, hence we check which is larger (7,5) or (6,6). In this case they are the same, meaning that we have more than one solution with length 4. Let's move left to (7,5). We now have a match, so we move to (6,4) and we have obtained the last element of the subsequence, "B". This process continues until one of the sequences is done (ie, either index being 0):

(6,4): match, move to (5,3), "AB"
(5,3): mismatch, could move to (5,2) or (4,3), "AB"
(5,2): match, move to (4,1), "DAB"
(4,1): match, move to (3,0), "BDAB"

This movement is illustrated in the figure below:

		0	1	2	3	4	5	6
			B	D	C	A	B	A
0		0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	**1**	1	2	2	3	3
5	D	0	1	**2** ←	**2**	2	3	3
6	A	0	1	2	2	**3**	3	4
7	B	0	1	2	2	3	**4** ←	**4**

3 Implementation Details

3.1 Input Data

The description of the problem is contained in a file (e.g., `ex1.in`) that starts with two integers (positive, $< 2^{16}$) indicating the size of each of the two input sequences. Then, each on a separate line, come the two sequences. For our purposes, these sequences will be strings of characters.

Your program should allow one and only one input parameter in the command line, used to specify the name of this input file.

3.2 Output Data

The output of this problem should be, in two separate lines:

- an integer, representing the size of the longest subsequence
- a string with this subsequence

Your program should send these two output lines (and nothing else) to the standard output!

As is the case in the example above, there may be more than on subsequence with the maximum length. In order for the solution to be unique you should follow these rules: associate the first string with the rows (and the second with the columns); when moving backwards in the matrix to determine the subsequence, in case up and left have the same value always move left. These rules were followed when there were equal options in the example above.

3.3 Sample Problem

If we have the two input strings ACCGGTCGAGTGCGCGGAAGCCGGCCGAA and GTCGTTCGGAATGCCGTTGCTCTGTAAA, the input file `ex1.in` will be:

```
29 28
ACCGGTCGAGTGCGCGGAAGCCGGCCGAA
GTCGTTCGGAATGCCGTTGCTCTGTAAA
```

The output after we execute your program should be:

```
$ lcs-serial ex1.in
20
GTCGTCGGAAGCCGGCCGAA
$
```

3.4 Computing the Match Value

This problem is computationally so simple that the overheads incurred by parallelizing it are hard to recover except for very large input sequences. Hence, to make your life easier, we introduce a minor cheat.

Instead of simply adding 1 when there is a match, your program should add the value returned by the following routine:

```
short cost(int x)
{
    int i, n_iter = 20;
    double dcost = 0;
    for(i = 0; i < n_iter; i++)
dcost += pow(sin((double) x),2) + pow(cos((double) x),2);
    return (short) (dcost / n_iter + 0.1);
}
```

Note that this routine always returns 1, independently of the input value, it just takes some time to compute it... You should call it using a variable, not a constant, for example the index of the loop.

You may need to include `math.h` and compile your code with the math library `-lm`.

4 Part 1 - Serial implementation

Write a serial implementation of the algorithm in C (or C++). Name the source file of this implementation `lcs-serial.c`. As stated above, your program should expect exactly one input parameter.

This will be your base for comparisons, it is expected that it should be as efficient as possible.

5 Part 2 - OpenMP implementation

Write an OpenMP implementation of the algorithm, with the same rules and input/output descriptions. Name this source code `lcs-omp.c`.

You can start by simply adding OpenMP directives, but you are free, and encouraged, to modify the code in order to make the parallelization more effective.

Be careful about synchronization and load balancing!

6 Part 3 - MPI implementation

Write an MPI implementation of the algorithm as for OpenMP, and address the same issues. Name this source code `lcs-mpi.c`.

For MPI, you will need to modify your code substantially. Besides synchronization and load balancing, you will need to take into account the minimization of the impact of communication costs.

Extra credits will be given to groups that present a combined MPI+OpenMP implementation.

7 What to Turn in, and When

You must eventually submit the sequential and both parallel versions of your program (please use the filenames indicated above) and the times to run the parallel versions on input data that will be made available (for 1, 2, 4 and 8 parallel tasks). Note that we will **not** be using any level of compiler optimizations to evaluate the performance of your programs, so you also shouldn't.

You must also submit a short report about the results (1-2 pages) that discusses:

- the approach used for parallelization
- what decomposition was used
- what were the synchronization concerns and why
- how was load balancing addressed
- what are the performance results, and are they what you expected

You will turn in the serial version and OpenMP parallel version at the first due date, with the short report, and then the serial version again (hopefully the same) and the MPI parallel version at the second due date, with an updated report. Both the code and the report will be uploaded to the Fenix system in a zip file. Name these files as `g<n>omp.zip` and `g<n>mpi.zip`, where `<n>` is your group number.

1st due date (serial + OMP): **October 31st**, until 5pm.

Note: your project will be tested in the practical class just after the due date.

2nd due data (serial + MPI): **December 5th**, until 5pm.

Note: your project will be tested in the practical class just after the due date.