Blue Waters Petascale Semester Curriculum v1.0

Unit 4: OpenMP

Lesson 2: Longest Common Subsequence

Developed by Paul F. Hemler for the Shodor Education Foundation, Inc.

Browse and search the full curriculum at http://shodor.org/petascale/materials/semester-curriculum

We welcome your improvements! You can submit your proposed changes to this material and the rest of the curriculum in our GitHub repository at https://github.com/shodor-education/petascale-semester-curriculum

We want to hear from you! Please let us know your experiences using this material by sending email to petascale@shodor.org

Longest Common Substring

Paul F. Hemler, Hampden-Sydney College

Abstract

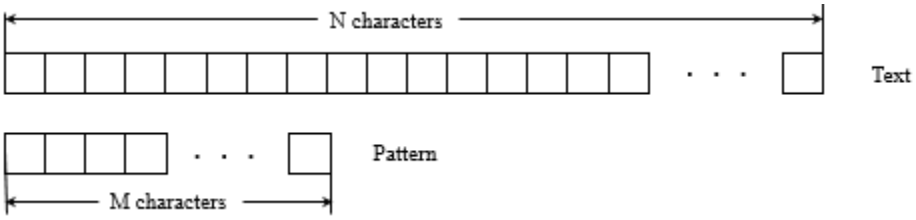
One of the very powerful features of OpenMP is the simplicity of converting a serial program into a parallel program by looking for "for" loops and placing a #pragma omp before them. Sometimes this works and conversion minimally changes the serial code. Unfortunately, there are times when conversion to a parallel program are not so simple because the loop computations are not independent. In that case we say there is a data dependency. Sometimes the problem can be fixed by adding some statements inside the loop but other times a completely different solution is required.

In this module the problem of finding the longest common subsequence between a text string and a pattern string is developed. An efficient serial solution using the dynamic programming problem solving methodology is derived and implemented. Unfortunately, this solution cannot be simply made parallel with OpenMP because of an underlying data dependency. The module presents the serial solution, the data dependency, and finally an alternative way of filling the dynamic programming table that can be simply made parallel by adding some OpenMP decorations. The serial and parallel solutions are then executed on a large data set, showing improved program performance as the number of threads increases.

Introduction

The problem of finding a pattern in a given text string has been prevalent in Computer Science for several decades. One standard application of this problem is determining the differences between two text files. Among other applications this is useful and necessary in source code management systems. Roughly two decades ago Molecular Biologist found that DNA could be encoded as long sequences of the four characters ACGT that represent four different bases. When a new DNA sequence is made it is often useful to determine if other sequences are found in it. We will consider this application in this module.

The naïve solution is to align the sequences and perform a character comparison.



In this solution, there are $N - M + 1$ possible positions for the pattern with respect to the text. If on average $M/2$ characters in the pattern are compared before deciding there is no match the complexity of this solution is $O(NM)$. The computational cost of this method is not prohibitively expensive but the major problem is that an exact match between the pattern and text is required. In the biological setting it is acceptable for only some of the characters in the pattern to match and gaps are permissible. This inexact match allows for mutated genes in the DNA to correspond to non-mutated or genes with different mutations to be found.

A more flexible match approach allowing for gaps to occur between the Pattern and text results in a recursive formulation of the problem. Letting P represent the pattern string and T the text string, the solution is,

```
int lcs (char *T, char *P)
{
    if (*T == '\0' || *P == '\0')
        return 0;
    else if (*T == *P)
        return 1 + lcs (T+1, P+1);
    return max(lcs (T+1,P), lcs (T, P+1));
}
```

This solution will correctly find the length of the longest matching substring. The two recursive calls in the last line of the function perform redundant work (overlapping sub problems) resulting in a running time that is exponential. The problem demonstrates the hallmarks of a dynamic programming solution because we seek a maximum length subsequence (an optimization problem) and there are overlapping sub problems in the recursive formulation.

Dynamic programming solutions utilize a table of some kind, which keep track of the optimal solution for the sub problems. In this application we can envision the text string going across the top of the table and the pattern going down the left side of the table. For example, let the text be the string: "GCTCAGC" and the pattern be "AGGTAC". Then the table is:

		G	C	T	C	A	G	C
A								
G								
G								
T								
A								
C								

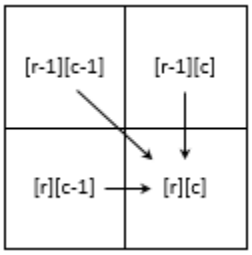
The values to be filled in the table correspond to the number of matching characters in the subsequence. One way of thinking about filling the table is to consider one letter in the pattern at a time, filling in one row at a time. If the letter in the pattern match the corresponding letter in the text, the value in the cell is one greater than not considering the letter at all. For example, when considering the element in cell [*r*][*c*] the number of previously matching characters is found in cell [*r – 1*][*c – 1*]. So, if the r^{th} character in the pattern matches the c^{th} character in the text,

$$table[r][c] = 1 + table[r-1][c-1]$$

If the r^{th} character in the pattern does not match the c^{th} character in the text then the best match so far is the best match so far considering the r^{th} character, *table[r][c – 1]* or the best match without considering the r^{th} character, *table[r - 1][c]*. With this in mind a systematic table filling algorithm,

```
for(int r = 0; r < nr; ++r)
    for(int c = 0; c < nc; ++c)
        if (p[r] == t[c])
            table[r][c] = 1 + table[r - 1][c - 1];
        else
            table[r][c] = max(table[r - 1][c], table[r][c - 1]);
```

where **nr** and **nc** correspond to the number of rows and columns respectively. The value in a single cell in the table is dependent on only three other nearby cells. The one above, to the upper left and the left as shown in the figure below.



To eliminate the special cases of these neighbors missing when filling the first row and the first column an extra row and column initialized to all zeros is added to the table. The table before the nested loops in the code above is

		G	C	T	C	A	G	C
	0	0	0	0	0	0	0	0
A	0							
G	0							
G	0							
T	0							
A	0							
C	0							

The code then progresses in a systematic fashion filling in the contents row by row. In essence, this is process is determining the solution of smaller problems until the final element in the table is filled in, which represents the solution to the entire problem. The first sub problem considered is matching the letter A in the pattern to the letter G in the text. Since the characters do not match a zero is entered into the table. The next sub problem considered is matching the letter A in the pattern to the string "GC" in the text. Since the pattern does not match the final letter in the considered text, another zero is entered in the table. This process continues until the sub problem is to match the letter A in the pattern to the string "GCTCA", at which point there is a match and a one is placed in the table. Since there is only one letter in the pattern it can at most match one letter in the string, so the rest of the row is filled with ones. Using the code above the table after completing the first row is,

		G	C	T	C	A	G	C
	0	0	0	0	0	0	0	0
A	0	0	0	0	0	1	1	1
G	0							
G	0							
T	0							
A	0							
C	0							

The second row means the sub problem of matching the pattern with two letters, “AG” is solved by considering one letter at a time in the pattern. The pattern matches the text when considering the string “GCTCAG”, so a two is entered into the table at this point. Since no other matches are possible, The table is,

		G	C	T	C	A	G	C
	0	0	0	0	0	0	0	0
A	0	0	0	0	0	1	1	1
G	0	1	1	1	1	1	2	2
G	0							
T	0							
A	0							
C	0							

The third row means the sub problem of matching the pattern with the first three letters “AGG” is considered. The third row is a repeat of the second row because at most only the pattern “AG” or “A G” matches the text, which is two letters. The fourth row considers the sub problem of matching the pattern “AGGT” with the text, one letter at a time. In this case, the letter T matches the T in the text and again at most two letters match. The resulting table is,

		G	C	T	C	A	G	C
	0	0	0	0	0	0	0	0
A	0	0	0	0	0	1	1	1
G	0	1	1	1	1	1	2	2
G	0	1	1	1	1	1	2	2
T	0	1	1	2	2	2	2	2
A	0							
C	0							

The fifth row in the table represents matching the pattern “AGGTA” to the string one letter at a time. In this case the letter G matches the first letter G in the text and the letter C in the text does not match any letters in the pattern. When considering the first three letters of the text, two letters match and again there is no match between the pattern and the fourth letter in the text. One additional matching letter exists when considering the sub string text “GCTCA”, so a three is entered into the table. No other matches are possible, so the table at this point is,

		G	C	T	C	A	G	C
	0	0	0	0	0	0	0	0
A	0	0	0	0	0	1	1	1
G	0	1	1	1	1	1	2	2
G	0	1	1	1	1	1	2	2
T	0	1	1	2	2	2	2	2
A	0	1	2	2	2	3	3	3
C	0							

The final letter in the pattern matches the final letter in the text and three previous letters matched so the final contents of the table is,

		G	C	T	C	A	G	C
	0	0	0	0	0	0	0	0
A	0	0	0	0	0	1	1	1
G	0	1	1	1	1	1	2	2
G	0	1	1	1	1	1	2	2
T	0	1	1	2	2	2	2	2
A	0	1	2	2	2	3	3	3
C	0	1	2	2	3	3	3	4

The value of four in the bottom right cell of the table means that after considering all the letters in the pattern and all the letters in the text, there are four letters that match. This is the solution to the original problem and it indicates the longest common subsequence between the text and patterns is four letters.

As is very typically in a dynamic programming solution, the value in the table is the size of the longest common subsequence but it is not the resulting subsequence. To find the actual sequence of characters that comprise the LCS further processing is necessary and it starts by considering the bottom right cell in the table. If the text and pattern characters match, that character is the last character in the LCS. If the characters match we know we should look at the cell that is one row and one column before the cell, which is the top left neighbor. If the characters do not match we simply consider the cell to the left. This process continues until a cell in the columns of all zeros (the first column) is considered. The table below shows the backtracking through the table.

		G	C	T	C	A	G	C
	0	0	0	0	0	0	0	0
A	0	0	0	0	0	1	1	1
G	0	1	1	1	1	1	2	2
G	0	1	1	1	1	1	2	2
T	0	1	1	2	2	2	2	2
A	0	1	2	2	2	3	3	3
C	0	1	2	2	3	3	3	4
		G		T		A		C

For this example, the longest common subsequence is GTAC, which of course is four characters long.

The parallel solution

There is a data dependency in the nested for loops that fill in the table. Specifically, the previous element in a row in needed to determine the current element. This means each computations along the rows are not independent and cannot be separated and performed with different threads. The end result is filling the table row by row or column by column cannot be made parallel.

We can alternately fill the table along diagonal lines. With this approach there is no longer a data dependency along the diagonal and all computations are independent, which is then easily converted into a parallel program using OpenMP.

Consider the initialized table for this problem. The diagonals are shown in the figure below.

		G	C	T	C	A	G	C
	0	0	0	0	0	0	0	0
A	0							
G	0							
G	0							
T	0							
A	0							
C	0							

The table is filled along the diagonals starting at the bottom left and progressing to the top right, one diagonal at a time. When filling the table in this fashion all the necessary cells have been filled from a previous diagonal, so that each cell on the diagonal is independent and can be computed in parallel. The code to determine the indices along the diagonal is a bit more complicated that simply filling by rows or columns and is shown below.

```
for (diag = 1; diag <= nr + nc - 1; ++diag) {
    diagSize = diag;
    if (diag >= nr) diagSize = nr;
    if (diag >= nc) diagSize = nr - diag - nc;

    for (k = 0; k < diagSize; ++k) {
      int r = diag - k;
      int c = k + 1;
      if (diag > nr) {
          r = nr - k;
          c = diag - (nr - 1) + k;
      }
      if (t[c - 1] == p[r - 1])
          table[r][c] = 1 + table[r - 1][c - 1];
      else
          table[r][c] = max(table[r - 1][c], table[r][c - 1]);
      }
}
return table[nr][nc];
```

In the code, nr and nc refer to the number of rows and columns respectively and there are nr + nc – 1 diagonals. The length of the diagonal within the table is then determined so that the diagonal is

completely within the bounds of the table. The inner for loop enumerates all the cell indices along each diagonal and the conditional is the same as the serial version.

Using the power of OpenMP to convert a serial function into a parallel function results in the following code.

```
      for (diag = 1; diag <= nr + nc - 1; ++diag) {
          diagSize = diag;
          if (diag >= nr) diagSize = nr;
          if (diag >= nc) diagSize = nr - diag - nc;

#pragma omp parallel for num_threads(nThreads) shared(table, diag, diagSize, nc, nr)

          for (k = 0; k < diagSize; ++k) {
            int r = diag - k;
            int c = k + 1;
            if (diag > nr) {
                r = nr - k;
                c = diag - (nr - 1) + k;
            }
            if (t[c - 1] == p[r - 1])
                table[r][c] = 1 + table[r - 1][c - 1];
            else
                table[r][c] = max(table[r - 1][c], table[r][c - 1]);
            }
      }
      return table[nr][nc];
```

The OpenMP pragma allows the user to control the number of threads that will be spawned along each diagonal and the shared construct is not necessary but is considered good style.

Since the reformulated function to fill the table without any data dependences is dramatically different from the formulation that fills the table by rows, a serial version of the function was also tested and compared to the row filling and the parallel diagonal functions.

Results

The program was executed on a laptop with an i7-8805H CPU operating at 2.6 GHz and 16 GB of RAM using the Windows operating system. There were two datasets representing two different genomes searching for two different genes. One dataset contained a text string of 11240 characters and a pattern gene of 1719 characters and is referred to as the pML104Cas9 data set. The other dataset contained a text string of 57253 characters and a pattern gene containing 4173 characters and is referred to as the cheetobroGene22 dataset. The run-times were recorded for the serial row, serial diagonal, and parallel diagonal table filling function and is show below.

```
pML104Cas9 dataset
Serial time rows: 0.333
Serial time diagonals: 0.525
Parallel time: 0.228  number of threads: 2

Serial time rows: 0.339
Serial time diagonals: 0.529
Parallel time: 0.138  number of threads: 4
```

```
Serial time rows: 0.329
Serial time diagonals: 0.536
Parallel time: 0.092  number of threads: 8


cheetobroGene22 dataset
serial time rows: 4.022
Serial time diagonals: 8.300
Parallel time: 4.241  number of threads: 2

serial time rows: 4.007
Serial time diagonals: 8.414
Parallel time: 1.749  number of threads: 4

serial time rows: 4.020
Serial time diagonals: 8.408
Parallel time: 1.069  number of threads: 8
```

All three functions generated exactly the same table for the same input data set. The execution time for the serial diagonal function is significantly greater than the serial row filling function, most likely because the row filling function better utilized the cache memory where the serial diagonal function does not use the cache. In any case, the parallel version always executed faster than the serial row function and as more threads were used the program performance improved.

The program was ported to a UNIX server and the same two data sets executed with the same number of threads as the results are shown below.

```
pML104Cas9 dataset
Serial time rows: 0.091
Serial time diagonals: 0.188
Parallel time: 0.152  number of threads: 2

Serial time rows: 0.084
Serial time diagonals: 0.189
Parallel time: 0.106  number of threads: 4

Serial time rows: 0.083
Serial time diagonals: 0.205
Parallel time: 1.583  number of threads: 8

cheetobroGene22 dataset
Serial time rows: 1.116
Serial time diagonals: 3.285
Parallel time: 1.800  number of threads: 2

Serial time rows: 1.114
Serial time diagonals: 3.241
Parallel time: 1.021  number of threads: 4

Serial time rows: 1.126
Serial time diagonals: 3.221
Parallel time: 6.725  number of threads: 8
```

The program executed significantly faster on the two data sets. The parallel version did not improve on the execution time over the serial version where the rows were filled. Presumably, this is because the UNIX server had a processor with a significantly larger cache and filling the table by rows better utilized the cache. The UNIX server also showed the program execution time dramatically increased when eight threads were requested.