Parallel Longest Common Subsequence

Harlan Iverson

Mar 23, 2011

Abstract

The approach I took to parallel-izing LCS was to think of the length grid as a pipeline and process blocks from top left to bottom right. Each processor is responsible for a single row of the grid and process its blocks from left to right. In order to process a block, it must wait for the block above it to be processed. The key idea is that rather than processing an entire row at a time, lower rows can be processed while higher ones are not completed (in a staggered fashion). Blocks are created of width $w = m/p^2$and height $h = n/p$. The choice of m and n being horizontal or vertical is arbitrary in my implementation, but their relative lengths may make one a better choice than the other because of spatial locality.

Overview

Each of the three implementations (OpenMP, MPI and pthreads), rely on a set of shared grid and LCS block processing code. This allows the length and traceback to be performed in a partitioned fashion that is inependent of the concurrency mechanism. See Figure 1 for a high level illustration.

```
typedef struct cell_t {
        int len;
        enum { none = 0, up, left, upleft } dir;
} cell_t;
typedef struct grid_t {
        int m;
        int n;
        const char* s1;
        const char* s2;
        cell_t* cells;
} grid_t;
typedef struct backtrack_state_t {
        int i;
        int j;
        int pos;
        char* res;
} backtrack_state_t;
void lcs_length_block(const grid_t* grid,
        const int y, const int x,
        const int h, const int w);
void lcs_backtrack_block(const grid_t* grid,
        backtrack_state_t* state,
        int min_i, int min_j);
```

The lcs_length_block function performs the normal lcs_length algorithm but only on a limited range of the matrix. The lcs_backtrack_block function does the same, but it resembles a re-entrant function with

```
          <---    len(s1) = m   --->
                      x          x+w
+-------+-------+-------+-------+-------+
| * * * | * * * | * * * |       |       |         A
| * * * | * * # | # # # |       |       |         |
+-------+-------+-------+-------+-------+ y       |
| * * * | * * # | 0 0 0 |       |       |    len(s2) = n
| * * * | * * # | 0 0 0 |       |       |         |
+-------+-------+-------+-------+-------+ y+h     |
|       |       |       |       |       |         V
|       |       |       |       |       |
+-------+-------+-------+-------+-------+
```

Figure 1: The logical view of the LCS length matrix, created for use in the parallel representation. When finding the length a block, the cells immediately to the top and left of it must be processed first. Tracing back an LCS happens from bottom right to top left, and may be done in parallel by row.

the state passed between processes. This is important because each process may not have access to the entire grid (namely in a message passing environment).

Both models can be thought of in terms of *producer-consumer*. In the message passing version, this is realized through blocking receives and non-blocking sends; while in the shared memory versions it is realized more explicitly through a queue with a conditional variable which has blocking consume and non-blocking produce calls (ignoring concurrency lost by small critical sections).

Message Passing

The general procedure for a process is:

1. Do a *blocking* receive from the process of the previous rank in order to get the row above what it is responsible for. Because of the way lcs_length_block works it only needs a single row of elements directly above the block it is processing.
2. It then runs lcs_length_block.
3. When that completes, it *asynchronously* sends the bottom row of its block to the process of next rank.

Once the LCS length matrix is completely calculated, each process only has a limited piece of it–namely, that of the row it is responsible for.

Backtrack happens in reverse, with the process of the last rank kicking it off from the bottom right corner to the top left corner of its row. Once it has finished this, it forwards the backttrack state to the process of previous rank to resume execution where it left off. Once the root process finishes it reverses the string and the algorithm is complete.

Shared Memory

Both shared memory mechanisms (OpenMP and pthreads) are essentially identical. The general procedure is for each thread to process all blocks in a row from left to right. Each process has a queue driven by a *conditional variable.*

1. It starts by *consuming* an offset from its queue to begin processing; it may not begin processing until there is an item to be consumed.

2. Once it has an offset, it proceeds to call lcs_length_block using the block it is responsible for.
3. Once it has completed, it ends by *producing* that same offset into the queue for the next thread to process.

Backtrack happens in an essentially serial way, since only one thread is working at a time and there is no data exchange. In theory, if each thread had only a limited piece of the memory such as in the message passing implementation (this may have cache benefits in multi-core systems), using a parallel backtrack would be beneficial–therefore one was implemented. It works in exactly the same way as in the message passing model, but again, uses a queue with a conditional variable to realize the producer-consumer model.

As in message passing, backtrack happens in reverse. The last ranking thread processes a portion of the LCS matrix that directly corresponds to the rows it was responsible for computing. Once it reaches a boundary, it passes control to the next thread via a produce call.

Producer-Consumer in Shared Memory

Producer-consumer was realized in both shared memory implementations via a construct I wrote for my Operating Systems class last year (CSCI 4061, here at the U), prodcon_queue_t. Essentially it is an API that allows non-blocking queue insertion and blocking queue extraction using a conditional variable, in a thread-safe way.

```
typedef struct queue {
        int capacity;
        int size;
        int head;
        int tail;

        void** items;
} queue_t;
typedef struct prodcon_queue {
        pthread_cond_t empty_cv;
        pthread_cond_t full_cv;
        pthread_mutex_t mx;
        queue_t q;

        int consume_all;
} prodcon_queue_t;
int queue_produce( prodcon_queue_t* q, void* item );
void* queue_consume( prodcon_queue_t* q );
```

Performance

The shared memory implementations have essentially identical performance, and experience speedup as number of processes increase. The MPI implementation does not experience speedup for $p = 2$, but does beyond that. See Table 1.

small	**1**	**2**	**4**	**8**
Pthreads	0.03	0.02	0.01	0.01
OpenMP	0.03	0.02	0.01	0.01
MPI	0.03	0.02	0.01	0.01
medium	**1**	**2**	**4**	**8**
Pthreads	0.45	0.31	0.20	0.14
OpenMP	0.45	0.31	0.20	0.14
MPI	0.45	0.48	0.21	0.16
large	**1**	**2**	**4**	**8**
Pthreads	2.93	2.03	1.24	0.87
OpenMP	2.92	2.04	1.25	0.88
MPI	2.92	2.96	1.85	1.08

Table 1: Performance of LCS using various models and #s of processes. Each is averaged over three trials using the respective data set on parallel-X.cselabs.umn.edu. Note: these times include the traceback step (I had collected them before reading directions more closely, and it was tedius, sorry).