Solutions to INF3380 problems, week 4

February 20, 2017

Exercise 1

There were some similar exercises last week; recall the general principle of giving priority to tasks along the critical line.

Graph (a)

4 workers

There is only one way to proceed, so this should be trivial. Time required is simply $10 + 9 + 8 = 27$ time units.

3 workers

Broadly speaking, there are two ways to proceed; either unlock the leftmost task (task 6) on tier 2 (e.g. by completing tasks 1, 3 and 4) **(step 1)**, then completing that one while unlocking the right task **(step 2)**, or we could unlock the rightmost task instead. Regardless, we are left with a "tail" of tasks (either task 5 or task 6, then task 7) that must be completed in serial.

However we proceed, steps **1** and **2** will take a constant 20 time units in total; completing Task 6 before 5 leaves the smallest tail, however ($6 + 8 = 14$ vs $9 + 8 = 17$ time units), so we go with that.

Total time $\underline{t = 34 \text{ TU}}$.

2 workers

There are several ways to proceed, but it seems clear that completing the first tier of tasks first makes for maximum concurrency and minimum idling, so we go with that. Total time $t = 10 + 10 + 9 + 8\underline{= 37 \text{ TU}}$.

Graph (b)

4 workers

Again, nothing very interesting happens here, perhaps other than the fact that after the first batch of tasks has completed, everything happens in serial.

Total time elapsed is $t = 10 + 6 + 11 + 7 \underline{= 34 \text{ TU}}$.

3 workers

We'll run with the idea of giving priority to tasks along the critical path, as well as tasks unlocking them. It should be clear that this takes the same amount of time as with four workers.

Total time elapsed is $t = 10 + 6 + 11 + 7 \underline{= 34 \text{ TU}}$.

2 workers

Same principle.

Total time elapsed is $t = 10 + 10 + 11 + 7 = \underline{= 38 \text{ TU}}$.

Exercise 2

	(a)	(b)	(c)	(d)
Max concurrency:	8	8	8	2
Critical path length:	4	4	7	8
Max speedup:	15/4	15/4	2	15/8
Workers needed:	8	8	3	2
Speedup 2/4/8 workers:	15/8, 15/6, 15/4	(same)	14/8, 14/7, 14/7	15/8

Exercise 3

We are tasked with proving the bound

$$\left\lceil \frac{t}{l} \right\rceil \leq d \leq t - l + 1\,, \tag{$*$}$$

for a task dependency graph with t tasks, maximal degree of concurrency d and critical path length l. That is, we must prove the two inequalities

$$\left\lceil \frac{t}{l} \right\rceil \leq d \tag{$**$}$$

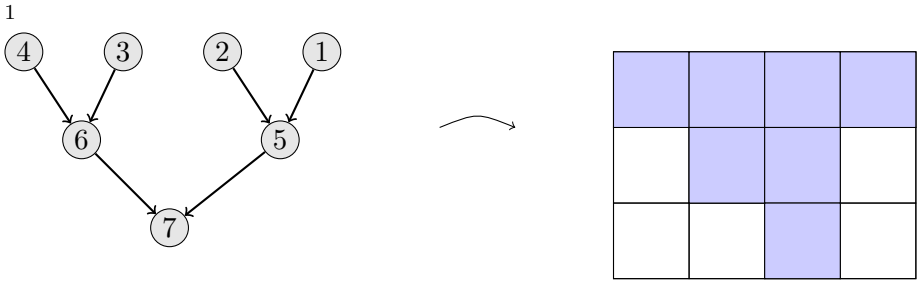
and

$$d \leq t - l + 1\,. \tag{$***$}$$

(The symbol $\lceil \cdot \rceil$, commonly referred to as the "ceiling function," means "round up.")

Geometric interpretation

[1]



This picture is not a formal proof, but gives an intuitive idea of why the inequality holds. Given a task dependency graph G with maximal degree of concurrency d and critical path length l, we map it into a rectangular block made of $d \times l$ cells as shown. For such a map to be possible, we see that there can be no more than $l \cdot d$ tasks to a graph (or we'd have to use a bigger rectangle); similarly there can be no fewer than $l + d - 1$ tasks (or we could use a smaller rectangle); these bounds trivially imply $(*)$.

Formal proof:

To show $(**)$, it suffices to note that $\frac{t}{l}$ is the *average* degree of concurrency; it is obviously bounded by the *maximal* degree of concurrency, which is d.

[1] Thanks to student Mikael Toresen for the idea.

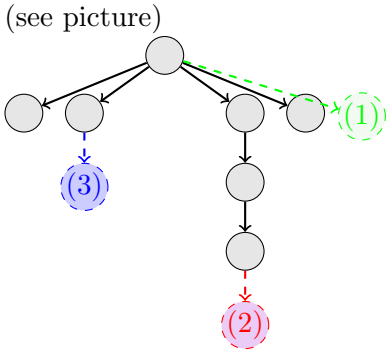
Moreover, the inequality extends to rounding up, because d is necessarily an integer.

For $(***)$, we use induction. Note first that for a single-node graph, the inequality holds trivially (with $d = t = l = 1 = t - l + 1$). For any given task dependency graph G with t nodes, we wish to show that we can remove nodes one by one until we are left with a graph $\hat{G}$, with only a single node if necessary, such that $(***)$ holds for $\hat{G}$, and that if we then reattach the nodes, one by one, $(***)$ holds for all these intermediary graphs.

So let G_k be a graph with k nodes, maximal degree of concurrency d_k and critical path length l_k, such that $(***)$ holds for G_k. Now, if we attach a node, exactly one of three possibilities will hold:

1. Maximal degree of concurrency increases by one, $d_{k+1} = d_k + 1$,
2. Critical path length increases by one, $l_{k+1} = l_k + 1$.
3. Neither d nor l increase, $d_{k+1} = d_k$, $l_{k+1} = l_k$.

(see picture)



Now it's just a matter of checking each case.
Case 1. Add 1 to both sides of $(***)$,

$$d_{k+1} = d_k + 1 \leq (k + l_k - 1) + 1 = (k + 1) - l_{k+1} + 1\,.$$

$\triangle$

Case 2. We add and subtract 1 to the rhs of $(***)$:

$$d_{k+1} = d_k \leq k - l_k - 1 + 1 + 1 = (k + 1) - l_{k+1} + 1\,.$$

$\triangle$

Case 3. Adding 1 to the rhs of $(***)$,

$$d_{k+1} = d_k \leq k - l_k + 1 \leq (k+1) - l_{k+1} + 1\,.$$

$\triangle$

$\square$

Exercise 4

We'll assume the house coordinates is stored in a `double` array `house_coords[2 * n]`, such that `house_coords[2 * i` and `house_coords[2 * i + 1]` give the x and y coordinates for house i respectively. For reference, a basic serial implementation would be

```
double dist(double x1, double y1, double x2, double y2) {

  return sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2));
}

double compute_mindist(double *houses_coords, int n,
                       double station_coords[2]) {
  double currentdist, mindist, x, y, sx, sy;

  sx = station_coords[0]; sy = station_coords[1];


  mindist = dist(house_coords[0]->x, house_coords[1], sx, sy);
  for (int i = 1; i < n; i ++) {

    x = house_coords[2 * i];
    y = house_coords[2 * i + 1];
    currentdist = dist(x,y, sx, sy);
    mindist = MIN(mindist, currentdist);

  }
  return mindist;
}

int main(int narg, char **argv) {

  // obtain data

  // mindist = compute_mindist(etc);

}
```

To implement this in `MPI`, assuming each process sits on its own portion of the data, an obvious solution is that each process calls the `compute_mindist` function above to find its local minimum, and an all-to-one reduction is performed to find the global minimum. This is just a single line of code, something along the lines of

```
MPI_Reduce(&local_min, &global_min, 1, MPI_DOUBLE, MPI_MIN, 0,
    MPI_COMM_WORLD);
```

(check MPI documentation for an explanation). Here, process 0 is the process that ends up with the global minimum.

Distributing. In exercises like these, it's likely that at the start, process 0 sits on the entire data set, though `MPI` does have functions for reading and writing to file in parallel. In this case, the easiest way to distribute data is via `MPI_Scatterv`. For a demonstration see the next exercise; for short problems like this, scattering data would probably be at least half of the code.
Extra. It's very possible we don't want just to find the shortest distance, but also which house it is that gives that distance (i.e. given the original `houses` array of structs, we want to find `i` such that `houses[i]` is the closest house to the railway). In serial programming, that would be just a few extra lines of code, but in parallel the problem becomes...actually, still quite easy. The problem occurs often enough that `MPI` has its own reduction routine for it, called `MINLOC`.

Exercise 5

Note

(The string search algorithm itself isn't really part of the scope of the course, so just look up Boyer-Moore or Boyer-Moore-Horspool on the internet for an example of an implementation.)

We'll assume process 0 initially has the pattern of size l, the text of size n, and that there are p processes. Roughly speaking, the algorithm goes as follows:

1. Compute how much of the text to give to each process.
2. Broadcast the pattern and distribute (scatter) the text in blocks.
3. It's possible the patterns occur at the "seams" where one process' portion ends and the next process' begins, so find some brilliant fix to this.

4. Have each process search through its portion of the text and count occurrences of the pattern, by means of, say, Boyer-Moore.

5. Sum the local results via an all-to-one reduction.

For step (1), a fair distribution is, as we saw last week

$$n_j = \left\lfloor \frac{n}{p} \right\rfloor + e_j \,,$$

where e_j is 0 or 1 depending on the remainder of n/p. However, we must have some overlap between the texts to be able to search at the seams, so we'll give process 1 the $l-1$ last characters from process 0's portion, and so on. If l is small, the distribution is still mostly fair, so we leave it unchanged; otherwise (or just for the sake of pedantry) we could give process 0 a slightly bigger portion of the text to search through.

See `ex5_dist.c` for a demonstration of the distribution.

Exercise 6

If we're interested only in the first instance of a pattern, the nature of the problem seems to change quite drastically. If we tried the distribution from Exercise 5, and let's assume the pattern appeared with some frequency, such that, say, Process 1 found it quite early, then processes 1, 2, 3, ..., $P-1$ should terminate, or they'd be busy performing useless work. That leaves us with process 0 doing all the remaining work. As well, we wish to keep communication to a minimum.

A better idea is a block-cyclic distribution (see lecture slides or curriculum book pp. 122-123), with occasional communication (all-to-all) to check if there's a point in continuing. This increases initialisation overhead, and if the individual blocks are small enough and the pattern big enough, the overlaps will take a relatively large amount of memory. On the other hand, bigger blocks will lead to the same problems as we started with.

So we're left with the task of the task of balancing these factors, as well as determining how often processes should communicate. But that's why we have philosophers.