CpSc 418

Homework 3

Due: Nov. 9, 2012, 11:59pm

5% extra credit for if solution submitted by 11:59pm on Nov. 5.

Please submit your solution using the handin program. Submit the program as

cs418 hw3

This requires you to have an account on the UBC Computer Science undergraduate machines. If you need an account, go to: `https://www.cs.ubc.ca/students/undergrad/services/account` to request one.

Your submission should consist of the following files:

`hw3.erl` – Erlang source (ASCII text). All functions requested in this assignment must be exported by this module.

`hw3.txt` – plain, ASCII text, or `hw3.pdf` – PDF.

The first file, `hw3.erl`, will be your solution to the programming part of the assignment, neatly commented.
The second file, `hw3.txt` or `hw3.pdf`, will be a plain text file of PDF file with your solution to the written part of the assignment. Handwritten solutions may be scanned and included in the hw3.pdf file.

1. **Mutual exclusion (15 points)** Figure 1 shows Dekker's mutual exclusion algorithm as presented in the October 4 lecture. A programmer decided to "simplify" the algorithm by changing the `while`-loop at line 3 to an `if`-statement – see figure 2. Here's their reasoning:

   > Now, consider the `while`-loop at lines 3–9. If the loop-body (lines 4–8) is executed by thread 0, then `turn` must be set to 0 when the while-loop at line 6 exits and execution proceeds to line 7. This means that thread 1 set turn to 0 at line 11 and will then set `flag[1]` to `false` at line 12 without entering its Nothing in lines 7–9 changes the value of `flag[1]`. This means that, `flag[1]` will be `false` (or about to be set to `false`) when thread 0 completes executing lines 7–9, and thread 0 will exit the `while` loop. Therefore, the loop body is executed at most once, and the `while`-loop can be replace by an `if`-statement.

   (a) **Counter-example trace (10 points)** Show that the modified version of the algorithm as shown in figure 2 does not guarantee mutual exclusion. In particular, show a counter-example trace like that on slide 31 of the October 4 slides (web only). Your trace should start with:

   ```
   PC₀ = PC₁ = 0;
   flag[0] = flag[1] = false;
   turn = 0;
   ```

   and end in a state with

   ```
   PC₀ = PC₁ = 8;
   ```

   See Figure 2A.

   (b) **Short explanation (5 points)** Write a short explanation (less than 50 words) of why the modified version doesn't work.

   The problem is that one thread can leave its critical section, set `turn`, and re-enter its critical section while the other thread is (suspended) at line 6. When the other thread resumes execution, it sees that `turn` has changed and enters its critical section without checking `flag`.

2. **Peterson's Algorithm (15 points)** Figure 3 shows Peterson's mutual exclusion algorithm. Peterson's insight is that when a thread tries to enter its critical section, it first sets `turn` to give the *other* thread priority. If both threads try to enter at roughly the same time, the last thread to try will set turn to give priority to the earlier thread.

thread 0:

```
PC₀= 0: while(true) {
PC₀= 1:   non-critical code
PC₀= 2:   flag[0] = true;
PC₀= 3:   while(flag[1]) {
PC₀= 4:     if(turn != 0) {
PC₀= 5:       flag[0] = false;
PC₀= 6:       while(turn != 0);
PC₀= 7:       flag[0] = true;
PC₀= 8:     }
PC₀= 9:   }
PC₀=10:   critical section
PC₀=11:   turn = 1;
PC₀=12:   flag[0] = false;
PC₀=13: }
```

thread 1:

```
PC₁= 0: while(true) {
PC₁= 1:   non-critical code
PC₁= 2:   flag[1] = true;
PC₁= 3:   while(flag[0]) {
PC₁= 4:     if(turn != 1) {
PC₁= 5:       flag[1] = false;
PC₁= 6:       while(turn != 1);
PC₁= 7:       flag[1] = true;
PC₁= 8:     }
PC₁= 9:   }
PC₁=10:   critical section
PC₁=11:   turn = 0;
PC₁=12:   flag[1] = false;
PC₁=13: }
```

Figure 1: Dekker's Algorithm

(a) **An Invariant (10 points)** Let

$$\begin{aligned} I \quad = \quad \forall k \in \{0,1\}. \quad & \texttt{flag[k]} = (3 \le \texttt{PC}_k \le 9) \\ \land \quad & ((6 \le \texttt{PC}_k \le 8) \land \neg\texttt{tmp}_k) \;\Rightarrow\; (\neg\texttt{flag[}\overline{k}\texttt{]} \lor (\texttt{turn} = k) \lor (\texttt{PC}_{\overline{k}} = 3)) \\ \land \quad & (\texttt{PC}_k = 8) \;\Rightarrow\; \neg\texttt{tmp}_k \end{aligned}$$

where $\overline{k} = 1 - k$ (i.e. it is the index of the "other" thread).

Prove that I is an invariant of th eprogram from figure 3.

Note: my version of Peterson's algorithm is a bit more pedantic than, for example, the version on wikipedia. They replace my loop at lines 4…7 with

`while(flag[`$\overline{k}$`] && (turn == `$\overline{k}$`));`

I introduced the local variables $\texttt{tmp}_0$ and $\texttt{tmp}_1$ to show that the algorithm works even if `flag[`$\overline{k}$`]` or `turn` is modified by the other thread while computing the condition for continuing the loop.

The invariant holds initially because both `flag` variables are false, and both program counters are zero:

$$\begin{aligned} I \quad & = \quad \forall k \in \{0,1\}. \quad && \texttt{flag[k]} = (3 \le \texttt{PC}_k \le 9) \\ & && \land \; ((6 \le \texttt{PC}_k \le 8) \land \neg\texttt{tmp}_k) \;\Rightarrow\; (\neg\texttt{flag[}\overline{k}\texttt{]} \lor (\texttt{turn} = k) \lor (\texttt{PC}_{\overline{k}} = 3)) \\ & && \land \; (\texttt{PC}_k = 8) \;\Rightarrow\; \neg\texttt{tmp}_k \\ & = \quad \forall k \in \{0,1\}. \quad && \texttt{false} = (3 \le 0 \le 9) \\ & && \land \; ((6 \le 0 \le 8) \land \neg\texttt{tmp}_k) \;\Rightarrow\; (\neg\texttt{false} \lor (\texttt{turn} = k) \lor (0 = 3)) \\ & && \land \; (0 = 8) \;\Rightarrow\; \neg\texttt{tmp}_k \\ & = \quad \forall k \in \{0,1\}. \quad && \texttt{false} = \texttt{false} \\ & && \land \; (\texttt{false} \land \neg\texttt{tmp}_k) \;\Rightarrow\; (\texttt{true} \lor (\texttt{turn} = k) \lor (0 = 3)) \\ & && \land \; \texttt{false} \;\Rightarrow\; \neg\texttt{tmp}_k \\ & = \quad \forall k \in \{0,1\}.\texttt{true} \land \texttt{true} \land \texttt{true} && \\ & = \quad \texttt{true} && \end{aligned}$$

Note: the one sentence explanation above is sufficient to get full credit. I showed the detailed derivation to help anyone who finds it helpful.

Now, I'll show that each clause of the invariant is preserved by each action of thread 0. The arguments for thread 1 are equivalent.

$\texttt{flag[0]} = (3 \le \texttt{PC}_0 \le 9)$:

- If $\texttt{PC}_0 = 2$, then performing this action sets `flag[0]` to `true` and $\texttt{PC}_0$ to 3 which establishes the clause.
- If $\texttt{PC}_0 = 9$, then performing this action sets `flag[0]` to `false` and $\texttt{PC}_0$ to 10 which also establishes the clause.

thread 0:

```
PC0= 0: while(true) {
PC0= 1:   non-critical code
PC0= 2:   flag[0] = true;
PC0= 3:   if(flag[1]) { //while changed to if
PC0= 4:     if(turn != 0) {
PC0= 5:       flag[0] = false;
PC0= 6:       while(turn != 0);
PC0= 7:       flag[0] = true;
PC0= 8:     }
PC0= 9:   }
PC0=10:   critical section
PC0=11:   turn = 1;
PC0=12:   flag[0] = false;
PC0=13: }
```

thread 1:

```
PC1= 0: while(true) {
PC1= 1:   non-critical code
PC1= 2:   flag[1] = true;
PC1= 3:   if(flag[0]) { //while changed to if
PC1= 4:     if(turn != 1) {
PC1= 5:       flag[1] = false;
PC1= 6:       while(turn != 1);
PC1= 7:       flag[1] = true;
PC1= 8:     }
PC1= 9:   }
PC1=10:   critical section
PC1=11:   turn = 0;
PC1=12:   flag[1] = false;
PC1=13: }
```

Warning: this code does not guarantee mutual exclusion!

Figure 2: Modified Dekker's Algorithm

---

step	from state					perform
	PC_0	PC_1	`flag[0]`	`flag[1]`	`turn`	
0	0	0	`false`	`false`	0	$PC_0 =$ 0: `while(true) {`
1	1	0	`false`	`false`	0	$PC_0 =$ 1: *non-critical code*
2	2	0	`false`	`false`	0	$PC_0 =$ 2: `flag[0] = true;`
3	3	0	`true`	`false`	0	$PC_0 =$ 3: `if(flag[1])`
4	10	0	`true`	`false`	0	$PC_1 =$ 0: `while(true) {`
5	10	1	`true`	`false`	0	$PC_1 =$ 1: *non-critical code*
6	10	2	`true`	`false`	0	$PC_1 =$ 2: `flag[1] = true;`
7	10	3	`true`	`true`	0	$PC_1 =$ 3: `if(flag[0]) {`
8	10	4	`true`	`true`	0	$PC_1 =$ 4: `if(turn != 1) {`
9	10	5	`true`	`true`	0	$PC_1 =$ 5: `flag[1] = false;`
10	10	6	`true`	`false`	0	$PC_0 =$ 10: *critical section*
11	11	6	`true`	`false`	0	$PC_0 =$ 11: `turn = 1;`
12	12	6	`true`	`false`	1	$PC_0 =$ 12: `flag[0] = false;`
13	13	6	`false`	`false`	1	$PC_0 =$ 13: }
14	0	6	`false`	`false`	1	$PC_0 =$ 0: `while(true) {`
15	1	6	`false`	`false`	1	$PC_0 =$ 1: *non-critical code*
16	2	6	`false`	`false`	1	$PC_0 =$ 2: `flag[0] = true;`
17	3	6	`true`	`false`	1	$PC_0 =$ 3: `if(flag[1])`
18	10	6	`true`	`false`	1	$PC_1 =$ 6: `while(turn != 1);`
19	10	7	`true`	`false`	1	$PC_1 =$ 7: `flag[1] = false;`
20	10	8	`true`	`false`	1	$PC_1 =$ 8: }
21	10	9	`true`	`false`	1	$PC_1 =$ 9: }
22	10	10	`true`	`false`	1	$PC_1 =$ 10: *critical section*

Figure 2A: Counter-example trace for modified Dekker's algorithm (question 1a).

Initially: $\mathtt{PC}_0 = \mathtt{PC}_1 = 0$; $\mathtt{flag[0]} = \mathtt{flag[1]} = \mathtt{false}$; $\mathtt{turn} = \mathtt{0}$.

thread 0:

```
PC0= 0: while(true) {
PC0= 1:   non-critical code
PC0= 2:   flag[0] = true;
PC0= 3:   turn = 1;
PC0= 4:   do {
PC0= 5:     tmp0 = flag[1];
PC0= 6:     tmp0 = tmp0 && (turn == 1);
PC0= 7:   } while(tmp0);
PC0= 8:   critical section
PC0= 9:   flag[0] = false;
PC0=10: }
```

thread 1:

```
PC1= 0: while(true) {
PC1= 1:   non-critical code
PC1= 2:   flag[1] = true;
PC1= 3:   turn = 0;
PC1= 4:   do {
PC1= 5:     tmp1 = flag[0];
PC1= 6:     tmp1 = tmp1 && (turn == 0);
PC1= 7:   } while(tmp1);
PC1= 8:   critical section
PC1= 9:   flag[1] = false;
PC1=10: }
```

Figure 3: Peterson's Mutual Exclusion Algorithm

- All other statements leave the value of $\mathtt{flag[0]}$ unchanged and don't change the value of $(3 \le \mathtt{PC}_0 \le 9)$. Thus, these statements maintain the invariant.

$((6 \le \mathtt{PC}_0 \le 8) \land \neg\mathtt{tmp}_0) \;\Rightarrow\; (\neg\mathtt{flag[1]} \lor (\mathtt{turn} = 0) \lor (\mathtt{PC}_1 = 3))$:

- If $\mathtt{PC}_0 \le 4$ or $\mathtt{PC}_0 \ge 8$, then performing the action establishes $(\mathtt{PC}_0 \le 5) \lor (\mathtt{PC}_0 \ge 9)$ which means that $(6 \le \mathtt{PC}_0 \le 8)$ and the clause is established.
- If $\mathtt{PC}_0 = 5$, then after executing the statement, $\mathtt{PC}_0 = 6$ and $\mathtt{tmp}_0 = \mathtt{flag[1]}$ which establishes the clause.
- If $\mathtt{PC}_0 = 6$, then I'll consider two cases:

  If, $\mathtt{tmp}_0$ holds before executing the statement,
  then $\mathtt{tmp}_0 = (\mathtt{turn} = 1)$ after executing the statement. So, if $((6 \le \mathtt{PC}_0 \le 8) \land \neg\mathtt{tmp}_0)$ holds after executing the statement, we can conclude $\mathtt{turn} \ne 1$ after executing the statement.
  Now (blush, blush), I see that I should have either included a clause $(\mathtt{turn} = 0) \lor (\mathtt{turn} = 1)$ in the invariant or written the current clause as

  $$((6 \le \mathtt{PC}_0 \le 8) \land \neg\mathtt{tmp}_0) \;\Rightarrow\; (\neg\mathtt{flag[1]} \lor (\mathtt{turn} \ne 1) \lor (\mathtt{PC}_1 = 3))$$

  I'll claim that $(\mathtt{turn} = 0) \lor (\mathtt{turn} = 1)$ is "obvious," and I'll instruct Mike to give extra credit to anyone who spots this technicality.
  With this extra assumption, we now get that if $((6 \le \mathtt{PC}_0 \le 8) \land \neg\mathtt{tmp}_0)$ holds after executing the statement, so does $\mathtt{turn} \ne 1$ and therefore $\mathtt{turn} = 0$ which establishes the clause.
  On the other hand if $\mathtt{tmp}_0$ holds after executing the statement, the implication is satisified because $((6 \le \mathtt{PC}_0 \le 8) \land \neg\mathtt{tmp}_0)$ is false.

  If, $\mathtt{tmp}_0$ does not hold before executing the statement,
  then the right side of the implication, $\neg\mathtt{flag[1]} \lor (\mathtt{turn} \ne 1) \lor (\mathtt{PC}_1 = 3)$, must have held before executing the statement. Because the statement doesn't change any variables in the right side of the implication, it continues to hold after the statement is executed.
- If $\mathtt{PC}_0 = 7$, then again there are two cases depending on $\mathtt{tmp}_0$.

  If $\mathtt{tmp}_0$ holds before executing the statement, then $\mathtt{PC}_0 = 4$ after executing the statement, and the clause is established.

  If $\mathtt{tmp}_0$ does not hold before executing the statement, then $\mathtt{PC}_0 = 8$ after executing the statement, and no other variables in the clause are changed. Thus, the clause continues to hold after executing the statement.
- I've covered all values for the $\mathtt{PC}_0$. This finishes the argument for this clause.

$(\mathtt{PC}_0 = 8) \;\Rightarrow\; \neg\mathtt{tmp}_0$:

- If $\mathtt{PC}_0 \ne 7$ before executing a statement of thread 0, then $\mathtt{PC}_0 \ne 8$ after executing the statement, and the clause is established.
- If $\mathtt{PC}_0 = 7$ before executing a statement of thread 0, then there are two cases depending on the value of $\mathtt{tmp}_0$.

  If $\mathtt{tmp}_0$ holds before executing the statement, then $\mathtt{PC}_0 = 4$ after executing the statement, and the clause is established.

  If $\mathtt{tmp}_0$ does not hold before executing the statement, then $\mathtt{PC}_0 = 8$ and $\mathtt{tmp}_0 = \mathtt{false}$ after executing the statement which establishes the clause.

Now, I'll show that executing a statement of thread 0 doesn't violate any of the clauses of I for the other thread.

$\texttt{flag[1]} = (3 \leq \texttt{PC}_1 \leq 9)$: Actions of thread 0 don't modify any variables appearing in this clause. Thus, the clause is maintained.

$((6 \leq \texttt{PC}_1 \leq 8) \land \neg\texttt{tmp}_1) \;\Rightarrow\; (\neg\texttt{flag[0]} \lor (\texttt{turn} = 1) \lor (\texttt{PC}_0 = 3))$
We need to consider actions of thread 0 that can change $\texttt{flag[0]}$ from $\texttt{false}$ to $\texttt{true}$, change $\texttt{turn}$ from 1 to anything else, change $\texttt{PC}_0$ from 3 to anything else.

- Setting $\texttt{flag[0]} = \texttt{true}$: this is the statement at $\texttt{PC}_0 = 2$. Performing the statement sets $\texttt{PC}_0 = 3$ which establishes this clause.
- Setting $\texttt{turn} \neq 1$: the only statement of thread 0 that modifies $\texttt{turn}$ is the one at $\texttt{PC} = 3$ which sets $\texttt{turn}$ to 1 and establishes this clause.
- Setting $\texttt{PC}_0 \neq 3$: The only statement that changes $\texttt{PC}_0$ from 3 to anythinge else is the one at $\texttt{PC} = 3$ which sets $\texttt{turn}$ to 1 and establishes this clause.

$(\texttt{PC}_1 = 8) \;\Rightarrow\; \neg\texttt{tmp}_1$: Actions of thread 0 don't modify any variables appearing in this clause. Thus, the clause is maintained.

(b) **Mutual Exclusion (5 points)** Prove that the invariant, I, from part (a) ensures mutual exclusion. In other words, show

$$I \;\Rightarrow\; \neg((\texttt{PC}_0 = 8) \land (\texttt{PC}_1 = 8))$$

Showing $I \;\Rightarrow\; \neg((\texttt{PC}_0 = 8) \land (\texttt{PC}_1 = 8))$ is equivalent to showing $\neg((\texttt{PC}_0 = 8) \land (\texttt{PC}_1 = 8) \land I)$. In the derivation below, I just propagate the consequences of both program counters being 8 through the clauses of I to show $\neg((\texttt{PC}_0 = 8) \land (\texttt{PC}_1 = 8) \land I)$. Expanding the definition of I yields:

$$
\begin{array}{lllll}
 & & \multicolumn{2}{l}{\neg((\texttt{PC}_0 = 8) \land (\texttt{PC}_1 = 8) \land I)} & \\
 & \neg\forall k \in \{0,1\}. & & (\texttt{PC}_k = 8) & \\
 & & \land & \texttt{flag[k]} = (3 \leq \texttt{PC}_k \leq 9) & \\
 & & \land & ((6 \leq \texttt{PC}_k \leq 8) \land \neg\texttt{tmp}_k) \;\Rightarrow\; (\neg\texttt{flag[}\bar{k}\texttt{]} \lor (\texttt{turn} = k) \lor (\texttt{PC}_{\bar{k}} = 3)) & \\
 & & \land & (\texttt{PC}_k = 8) \;\Rightarrow\; \neg\texttt{tmp}_k & \%\ \text{bring } \texttt{PC}_0 = 8 \text{ and } \texttt{PC}_1 = 8 \text{ inside } \forall \\
\equiv & \neg\forall k \in \{0,1\}. & & (\texttt{PC}_k = 8) & \\
 & & \land & \texttt{flag[k]} = \texttt{true} & \\
 & & \land & \texttt{true} \land \neg\texttt{tmp}_k) \;\Rightarrow\; (\neg\texttt{flag[}\bar{k}\texttt{]} \lor (\texttt{turn} = k) \lor \texttt{false} \qquad\qquad) & \\
 & & \land & \texttt{true} \;\Rightarrow\; \neg\texttt{tmp}_k, & \%\ \text{propagate } \texttt{PC}_k = 8 \\
\equiv & \neg\forall k \in \{0,1\}. & & (\texttt{PC}_k = 8) & \\
 & & \land & \texttt{flag[k]} = \texttt{true} & \\
 & & \land & \texttt{true} \;\Rightarrow\; (\neg\texttt{true} \lor (\texttt{turn} = k)) & \\
 & & \land & \neg\texttt{tmp}_k, & \%\ \text{propagate } \texttt{flag[}k\texttt{]} \text{ and } \neg\texttt{tmp}_k \\
\equiv & \neg\forall k \in \{0,1\}. & & (\texttt{PC}_k = 8) & \\
 & & \land & \texttt{flag[k]} = \texttt{true} & \\
 & & \land & \texttt{turn} = k & \\
 & & \land & \neg\texttt{tmp}_k, & \%\ \text{boolean algebra} \\
\equiv & \neg\texttt{false}, & & & \%\ (\texttt{turn} = 0) \land (\texttt{turn} = 1) \;=\; \texttt{false} \\
\equiv & \texttt{true}. & & &
\end{array}
$$

3. **Mesh Networks (25 points)**

(a) **2-dimensional meshes (5 points)** Let $m > 0$ be an integer, and consider a 2D mesh network consisting of $N = m^2$ processors. For simplicity, assume that m is even. Each processor can be identified with a tuple, (i, j), where $0 \leq i, j < m$, and processor (i, j) has links to

processor $(i + 1, j)$, if $i < m - 1$;
processor $(i - 1, j)$, if $i > 0$;
processor $(i, j + 1)$, if $j < m - 1$;
processor $(i, j - 1)$, if $j > 0$.

Assume that each link can receive one message on each incoming link and send one message on each outgoing link using one unit of time.

Show that if each processor with $i < m/2$ sends distinct messages to each processor with $i \geq m/2$, then the time to convey these messages to their destinations is $O(N^{3/2})$.

Note:

- In office hours, I was discussed this problem with a student. I realized just before the HW was due that my hints led to a proof that the time is $\Omega(N^{3/2})$m not the $O(N^{3/2})$ as requested.
- I'll send a note to Mike and give that student full-credit for a solution that shows the wrong kind of bound. Furthermore, any student who acknowledges having collaborated with the student from that office hour will also get full-credit.
- If I got it backwards, I won't take a hard-line on others who make the same mistake. There will be two points deducted for making that error in any of parts a, b, or c (but only deducted once), and an additional three points for the error in part d (because the bound for part d doesn't work if you've got it backwards in part c).
- I'm including an appendix that has the proofs for the Ω bounds. These are nice, as they show that the bounds are tight, i.e., we've got big-Θ results for each part of this problem.

We can send the messages using dimension routing:

```
// route along dimension 0
forall j0 in 0..(m-1) {
   for i0 in 0..((m/2)-1) {
      for i1 in (m/2)..(m-1) {
         for j1 in 0..(m-1) {
            processor(i0, j0) sends a message that is destined
               for processor(i1, j1) to processor(i1, j0);
} } } }

// route along dimension 1
forall i1 in (m/2)..(m-1) {
   for j0 in 0..(m-1) {
      for i0 in (m/2)..(m-1) {
         for j1 in 0..(m-1) {
            processor(i1, j0) forwards a message from processor(i0, j0) to processor(i1, j1);
} } } }
```

To determine the time for the route along dimension 0, note that for any choice of `(i0, j0)`, `processor(i0, j0)` can send the messages for the `for i1` and `for i2` loops one per cycle without any collisions in the routing. Thus, `processor(i0, j0)` sends these messages in time $O\left(\frac{m}{2} * m\right) = O(\frac{m^2}{)}$. When `processor(i0, j0)` has sent its last message, `processor(i0+1, j0)` needs to wait one cycle (for the link from `(i0, j0)` to `(i0+1, j0)` to clear), and then `processor(i0+1, j0)` can send its $\frac{m^2}{2}$ messages in $O\left(\frac{m^2}{2}\right)$ time.

Thus the total time for the `for i0` loop is $O\left(\frac{m}{2}(\frac{m^2}{+}1)\right) = O(\frac{m^3}{)}$.

A similar analysis shows that the routing along dimension 1 can be completed in $O(m^3)$ time. Thus, the total time is $O(m^3 + m^3) = O(m^3) = O(N^{3/2})$ as desired.

(b) **d-dimensional meshes (5 points)** Let $m > 0$ and $d > 0$ be integers, and consider a d-dimensional mesh network consisting of $N = m^d$ processors. For simplicity, assume that m is even. Each processor can be identified with a tuple, $(i_0, i_1, \ldots, i_{d-1})$, where $0 \leq i_k < m$, and there is a link between a pair of processors iff all but one of their indices are identical, and they differ by ± 1 in the index for which they are different. Assume that each link can receive one message on each incoming link and send one message on each outgoing link using one unit of time.

Show that each processors with $i_0 < m/2$ sends distinct messages to each processor with $i_0 \geq m/2$, then the time to convey these messages to their destinations is $O(N^{1+\frac{1}{d}})$.

We could extend the dimension routing algorithm from question 3a in the obvious way and get a time bound of $O(dm^{d+1}) = O(dN^{1+\frac{1}{d}})$. The extra factor of d arises because it takes time $O(m^{d+1})$ along each dimension.

Note that this simple approach to dimension routing only uses the links in one dimension at a time. This leads to the factor of d in the time-bound. There are two ways we could improve the routing:

- Assume that we have a large number of these data shuffling operations to do and pipeline them. When the first batch finishes routing along dimension 0, then the first batch goes on to route along dimension 1 while the second batch starts its routing along dimension 0. With this approach, we can show that the network routes these batches with a *throughput* of one batch every $O(N^{1+\frac{1}{d}})$ cycles, even though each batch has a *latency* of $O(dN^{1+\frac{1}{d}})$ cycles. This is the simplest solution and it mimics the analysis of the hypercube from class. It should get full credit (even though it doesn't quite solve the problem as stated).
- We can partition the $N^2/4$ messages that must be sent into $(d-1)$ roughly equally sized sets. For example, if a message is sent from processor $(i_0, i_1, \ldots i_{d-1})$ to processor $(j_0, j_1, \ldots j_{d-1})$ we could assign the message to partion $j_{d-1} \bmod (d-1)$. If $m \gg d$, this will produce roughly equally sized partitions. If m is not large compared with d, we could devise another partitioning scheme.

  We now do dimension-routing on the first $d-1$ dimensions (indices 0 through $d-2$). For messages in partition 0, we start by routing along index 0, then index 1, and so on up to index $d-1$. For messages in partition k, we start by rouding along index k, then index $k+1$, and so on up to index $d-1$, then along index 0, then index 1, and so on up to index $k-1$. By analsis similar to that for question 3a, we get that the total time to route one partition along one dimension is $O(mN^2/(d-1))$. We can route all of the partitions in parallel, because each partition is using the links for a different dimension at any given phase of the routing. Thus, the total time for each dimension is $O(mN^2/(d-1))$. We route along $d-1$ dimensions; so the total time is $O(mN^2)$.

  We complete the routing by routing along dimension $d-1$. This time, we route all $N^2/4$ messages over the links for dimension $d-1$. This tames time $O(mN^2)$.

  The time for the complete route is $O(mN^2) = O\left(N^{1+\frac{1}{d}}\right)$.

  There are other ways of doing the routing to achieve the desired bound. A typical router will just route incoming traffice to available outgoing nodes that get the message closer to its destination. Such greedy routing probably achieves the desired bound as well, but I don't have a deriviation in mind. One could reasonably ask if a router could really take the global data-pattern into account. The answer is "yes." High-end networks for clusters (such as Infiniband) provide programmable routers that can be configured for application-specific routing patterns.

(c) **d-dimensional messages (cont.) (5 points)** Now consider a d-dimensional mesh, and let A and B be any partitioning of the processors into two sets of size $N/2$. Show that if each processor in A sends a distinct message to each processor in B, then the time to convey these messages to their destinations is $O(N^{1+\frac{1}{d}})$.

The routing method from the solution to 3b works in this more general case as well, and the same bound applies. The routing can be done in $O\left(N^{1+\frac{1}{d}}\right)$ time.

(d) **How big is a d-dimensional mesh? (10 points)** Use the result from part (c) to show that if a d-dimensional mesh of $N = m^d$ processors is implemented in our 3-dimensional universe, then the volume of the mesh of processors is $\Omega(N^{\frac{3}{2}(1-\frac{1}{d})})$. Compare this with the result for a hypercube from the October 9 lecture.

Consider any implementation of a d-dimensional mesh in our 3-dimensional universe. Choose any orientation for a plane. We can position such a plane so that $N/2$ processors are on each side of the plane. Let the processors on one side of the plane be set A as defined in problem 3c and the processors on the other side of the plane be set B. Each processor in A can send a distinct message to each procesosr in B using time $\Omega(N^{1+(1/d)})$. There are $\Theta(N^2)$ such messages. Thus,

$$\frac{\Theta(N^2)}{O(N^{1+\frac{1}{d}})} \quad = \quad \Omega(N^{1-\frac{1}{d}})$$

messages must cross the plane per unit time. This means that there must be $\Omega(N^{1-\frac{1}{d}})$ links crossing the plane. Assuming that each link has a cross-sectional area that is $\Omega(1)$, the intersection of the mesh with this plane has a diameter of $\Omega(N^{\frac{1}{2}-\frac{1}{2d}})$. Because this applies for *any* plane, an inscribing sphere for the processor has volume of $\Omega(N^{\frac{3}{2}(1-\frac{1}{d})})$.

While writing the solution, I realize that the problem asked for of $O(N^{\frac{3}{2}(1-\frac{1}{d})})$ rather than Ω. In fact a $\Theta(N^{\frac{3}{2}(1-\frac{1}{d})})$ result is possible. If a solution messes up big-O vs. big-Ω, it should still get full-credit – I made the mistake, and this isn't a theory course!

While writing this solution, I thought about "What if the mesh is arranged as a pancake (i.e. fairly thin in one dimension)?" The argument about diameter above is good enough for full-credit. Here's a way to handle "pancakes". I'll assume that anyone who is mathematically oriented enough to think of this question knows a fair amount about geometry; so, I'll write my "solution" without explaining all of the terms.

Let's define the volume of the mesh as the volume of the smallest convex polyhedron that contains the mesh. This convex polyhedron has some shortest diametrical chord. Let χ be such a chord and s be the length of this shortest chord. Now, consider any plane that contains chord χ and divides the processors of the mesh into two equally sized subsets. As argued above, the intersection of the mesh with this plane must have area $\Omega(N^{1-\frac{1}{d}})$. Because this intersection has a diametrical chord of length s, the chord perpendicular to this one must have a length of $\Omega(N^{1-\frac{1}{d}}/s)$. As this is true for any such plane, the area of the mesh when projected onto a plane that is normal to χ must be $\Omega(N^{2-\frac{2}{d}}/s^2)$, and thus the volume of the processor is $\Omega(N^{2-\frac{2}{d}}/s)$. This volume is minimized by *maximizing* s. To maximize the length of the shortest chord, we make all chords the same length. This is exactly the spherical implementation that we considered above, and the $\Omega(N^{\frac{3}{2}(1-\frac{1}{d})})$ bound applies.

Finally, one could ask about non-convex implementations. I think the previous case was technical enough; so, I won't worry about making it even more complicated.

4. **Reduce (38 points)** Given any solution (yours, the solution set, a friend's, etc. – but give proper attribution if it's not yours) for finding all prime numbers that are less than or equal to N,

   (a) **(10 points)** Write code to compute the sum of all primes that are less than or equal to N. You should measure the time that it takes to compute the sum given that the distributed list of primes has already been computed. Use the `wtree` module from the CpSc 418 erlang library. You should to use `wtree:create` to create a worker pool where the workers are organized as a binary tree, and `wtree:reduce` for the reduce operation to compute the sum. Note that the worker-pool returned by `wtree:create` can be used by any of the functions from module `workers` as well as those from `wtree`.

   In a bit more detail, you should write a module called `hw3` that exports the function `sum/2` where

   ```
   sum(W, Key) -> Total
   ```

   `W` is a worker pool, `Key` is the name for the distributed list of primes, and `Total` is the sum of the primes in that distributed list.

   For example, then the primes that are less than or equal to 100 are:

   ```
   2,  3,  5,  7,  11, 13, 17, 19, 23, 29, 31
   37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79
   83, 89, 97
   ```

Assume that the primes that are less than or equal to 100 are stored in a distributed list that's associated with the atom `p100` for worker pool `W`, and that `p90` is a distributed list for the primes that are less than or equal to 90.

```
hw3:sum(W, p100).
```

should print

```
1060
```

Likewise,

```
hw3:sum(W, p90).
```

should print

```
963
```

See `sum/2` in `hw3.erl`.

(b) **(5 points)** Measure the speed-up of your implementation of `sum` compared with `lists:sum` (assuming that you have already retrieved all of the primes into a single list) when running on gambier.ugrad.cs.ubc.ca with 64 worker processes.

- What is the speed-up for the value of N for which the sequential version takes 1 second?
- What is the smallest value of N for which the parallel version achieves 80% of the speed up that you reported above?

For this problem, I wrote functions:

`hw3:time_sum(Nproc, Pmax, Ntrials)`: Compute the sum of all primes that are $\leq$ `Pmax` sequentially and using `Nproc` processors. The parameter `Ntrials` tells `time_it:t` how many trials to run – if `Ntrials` is an integer, that is the number of trails to run. If `Ntrials` is a float, it runs enough trials to use (roughly) that many seconds.
The return value is a list of two tuples that give the mean and standard deviation of the run-times for the sequential and parallel versions.

`hw3:time_sum(Nproc, Pmax)`: I'm lazy. This is equivalent to `hw3:time_sum(Nproc, Pmax, 1.0)`.

`hw3:speedup_sum(Nproc, [Pmax1, Pmax2, ...])`: Run `hw3:time_sum(Nproc, Pmax, 50)` for each value of `Pmax`. Return a list of the form:

```
[ {Pmax1, SequentailTime1, SpeedUp1},
  {Pmax2, SequentailTime2, SpeedUp2},
  ...
]
```

`hw3:speedup_sum/0`: This calls `hw3:speedup_sum/3` with `Nproc=64` (as specified in the problem), and the list of values of `Pmax` that I used for this problem.

I ran trials for 46 values of `Pmax` from 10,000 to 40,000,000. The run trial with `Pmax` $= 32,000,000$ had the run-time closest to 1 second (a mean of 0.995 seconds, closer to 1 second than the measurement error). The speed-up with `Pmax` $= 32,000,000$ is 23.9.
Figure 4B shows the sequential run-time and speed-up for the various values of `Pmax` tried. The purple markers show the values with `Pmax` $= 32,000,000$, and the green dashed line is for $\mathit{SpeedUp} = 0.8 * 23.9 = 19.12$. The smallest value of `Pmax` (that I tried) for which the speed up is at least 80% of the speed-up when the sequential time was roughly 1 second is with `Pmax` $= 300,000$. This addresses the literal wording of the question.
The speed-up plot shows an "anomalous" region where the speed-up *decreases* with larger values of `Pmax`. For all values of `Pmax` (that I tried) that are greater than or equal to 10,000,000, the speed-up is greater than 19.2. So, 10,000,000 is also a reasonable answer to this question.

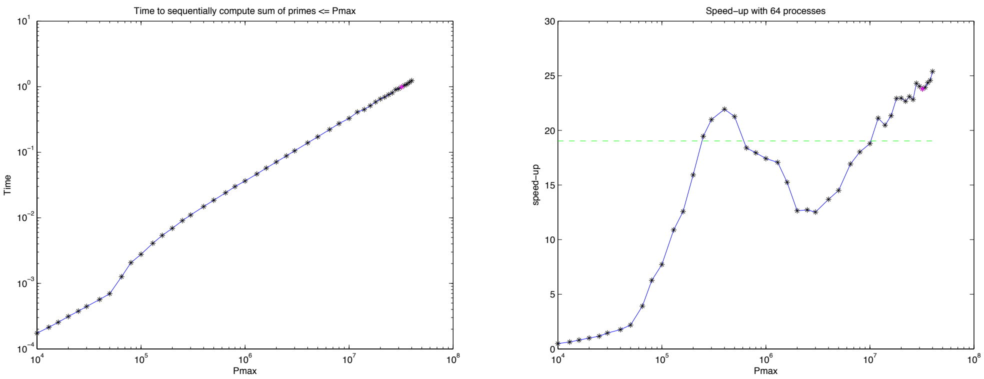

Figure 4B: Sequential runtime and parallel speed-up for sum-of-primes.

What causes the dip in the speed-up plot for $400,000 < \texttt{Pmax} < 4,000,000$? Looking at the plot of sequentail time versus `Pmax`, there is an upward shift of the "line" as `Pmax` goes from about 65,000 to about 400,000. I suspect that this is because when the list of primes is too long, it doesn't fit into the L1 (per-core) cache and requires more L2 cache accesses. In particular, the first 8 data points can be fit to the line

$$\mathit{SequentialTime} \quad = \quad \texttt{Pmax} * 13.03\text{ns} + 46.63\mu\text{s} \quad rcl$$

(mean-square-error $\approx 1\%$). Likewise, the points from $\texttt{Pmax} = 400,000$ to $\texttt{Pmax} = 4,000,000$ can be fit to the line

$$\mathit{SequentialTime} \quad = \quad \texttt{Pmax} * 33.27\text{ms} + 2.13\text{ms} \quad rcl$$

The slope increases by a factor of about 2.5, presumably due to cache misses. The change of slope appears as a "small" displacement of the line because I used a log-log plot.
The parallel version hits the same per-core cache limit when all eight cores hit the limit. This should occur for a value of `Pmax` a bit more than 8 times larger than that for the sequential case ("a bit more" because the primes are less dense for larger values). This matches the plots very well.

(c) **(15 points)** Write code to find the pair of consecutive primes with the largest gap, for the primes that are less than or equal to N. If there is more than one such pair, return the first such pair.
In a bit more detail, module `hw3` should export `largest_gap/2` where

```
largest_gap(W, Key) -> {P1, P2}
```

`W` is a worker pool, `Key` is the name for the distributed list of primes, and `{P1, P2}` is the pair of primes in that distributed list with the largest gap.
Continuing the example from part (a),

```
hw3:largest_gap(W, p100).
```

should print

```
{89,97}
```

If the primes that are less than or equal to 90 are stored in a distributed list that's associated with the atom `p90`, then

```
hw3:largest_gap(W, p90).
```

should print

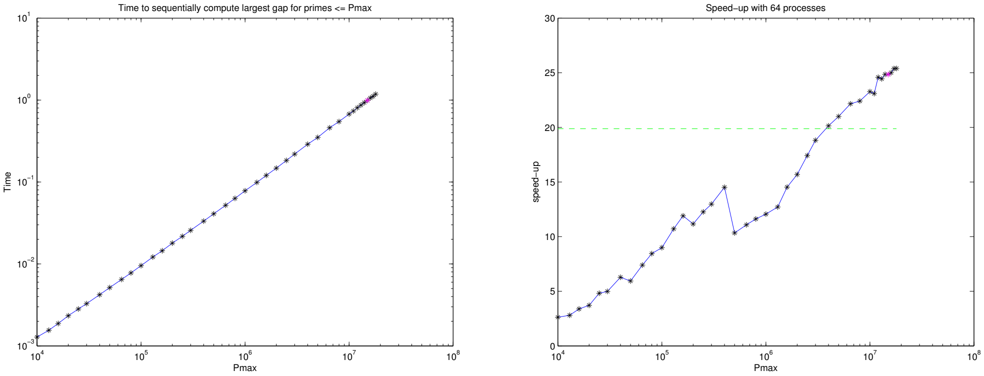

Figure 4D: Sequential runtime and parallel speed-up for largest-gap-of-primes.

`{23,29}`

There are seven pairs of consecutive primes less than 90 with a gap of 6, but `{23,29}` is the *first* (i.e. smallest) such pair.

See `largest_gap/2` in `hw3.erl`.

(d) **(8 points)** Write a sequential version of `largest_gap`. Measure the speed-up of your implementation of `largest_gap` compared with your sequential version (assuming that you have already retrieved all of the primes into a single list) when running on gambier.ugrad.cs.ubc.ca with 64 worker processes.

- What is the speed-up for the value of N for which the sequential version takes 1 second?
- What is the smallest value of N for which the parallel version achieves 80% of the speed up that you reported above?

I wrote functions `time_gap/3`, `time_gap/2`, `speedup_gap/3`, and `speedup_gap/0` in `hw3.erl` that correspond to the functions described above for measuring performance when computing the sum of the primes. I used the `largest_gap_leaf` function that I wrote for the parallel version to compute the gap sequentially.

After doing a few trial runs using `hw3:time_gap/3` to get a guess of where the sequential run time would reach one second, I ran trials for 39 values of `Pmax` from 10,000 to 40,000,000. The run trial with `Pmax` $= 15,000,000$ had the run-time closest to 1 second (a mean of 0.991 seconds over 50 trials). The speed-up with `Pmax` $= 15,000,000$ is 24.85.

Figure 4D shows the sequential run-time and speed-up for the various values of `Pmax` tried. The purple markers show the values with `Pmax` $= 15,000,000$, and the green dashed line is for $\mathit{SpeedUp} = \mathit{0.8} * \mathit{24.85} = \mathit{19.88}$. The smallest value of `Pmax` (that I tried) for which the speed up is at least 80% of the speed-up when the sequential time was roughly 1 second is with `Pmax` $= 400,000$.

As with the computation of the sum, the speed-up plot shows an “anomalous” region where the speed-up *decreases* with larger values of `Pmax`. It’s not as pronounced as with the computation of the sum. I’ll guess that this is related to caching behaviour, but this solution set is already long and late; so, I won’t explore that further now.