Solution Set

CPSC 418 Midterm

October 22, 2018

Answer all of the questions: question 0 and all five of questions 1 – 5.
Graded out of **62 points**.

0. **Who are you?** (2 points)

   (a) Your name: Mark Greenstreet

   (b) Your student number: 00000000

1. **Speed-Up** (15 points)
   This problem examines the speed-up of a typical reduce problem. The input consists of N data values. The sequential version takes T_1 operations per data value. The data is available to the sequential processes at the start of execution.

   The parallel version uses P worker processes. Each worker process has its part of the data stored locally at the start of the execution. Each worker process combines N/P values. Then, these results are combined using the binary-tree approach that we have used all term. Sending a message between processors takes the same amount of time as λ operations – this is the time from the send to completing the receive. For all of the questions below:

   - $N = 131,072 = 2^{17}$.
   - $P = 16 = 2^4$.
   - $\lambda = 16,384 = 2^{14}$.
   - T_{seq} is the time to execute the sequential program.
   - T_{par} is the time to execute the parallel program with P processors.

   We will consider two values for T_1: 32 and 8. Each question specifies which value of T_1 it uses.

   (a) (**3 points**) Let $T_1 = 32$. What is T_{seq}?
   Answer: $N * T_1 = 2^{17} * 2^5 = 2^{22} = 4,194,304$

   (b) (**3 points**) Let $T_1 = 32$. What is T_{par}?

   Answer: $T_{par} = (N/P)T_1 + \lambda \log_2 P = 327,680$.
   Explanation:

   - Time at leaves: $(N/P) * T_1 = (2^{17}/2^4)2^5 = 2^{18} = 262,144$
   - Time in reduce tree: $\lambda \log_2 P = 2^{14} * 4 = 2^{16} = 65,536$
     Note: We will also accept a solution that has a "downward" pass to start the reduce and an "upward" pass to combine the results – as implemented in the course library. In this case, the time in the tree would be 2^{17} time units.
   - Total time: $T_{par} = 2^{18} + 2^{16} = 5 * 2^{16} = 327,680$.
     If you made the "two-pass" assumption, then $T_{par} = 393,216$.

   (c) (**2 points**) Let $T_1 = 32$. What is the speed-up?

   Answer: 12.8.
   Explanation:

$$\begin{aligned} SpeedUp &= \frac{T_{seq}}{T_{par}} \\ &= \frac{2^{22}}{5*2^{16}} \\ &= \frac{2^6}{5} \\ &= 12.8 \end{aligned}$$

   If you made the two-pass assumption, then $SpeedUp = 10\frac{2}{3}$.

(d) (**4 points**) With a better compiler, we get $T_1 = 8$. What is the speed-up when $T_1 = 8$?

Answer: 8.
Explanation:

$$\begin{aligned} T_{seq} &= 2^{17} * 2^3 = 2^{20} = 1,048,576 \\ T_{par} &= (2^{17}/2^4) * 2^3 + 2^{14} * \log_2 16 \\ &= 2^{16} + 2^{16} \;=\; 2^{17} \;=\; 131,072 \\ SpeedUp &= T_{seq}/T_{par} \\ &= 2^{20}/2^{17} = 2^3 = 8 \end{aligned}$$

If you made the two-pass assumption, then $T_{par} = 3 * 2^{16}$, and $SpeedUp = 5\frac{1}{3}$.
Note: the $4\times$ performance advantage with a better compiler could be roughly the difference between Erlang and C.

(e) (**1 point**) Of the four versions, sequential and parallel with $T_1 = 8$ or $T_1 = 32$, which one is the fastest (i.e. executes in the least time)?
Answer: The parallel version with $T_1 = 8$.

(f) (**2 points**) Which parallel version ($T_1 = 8$ or $T_1 = 32$) has the greater speed-up? Give a one or two sentence explanation for the trend you observe between T_1 and speed-up.

Answer: The parallel version with $T_1 = 32$ has the larger speed-up.
Explanation: When T_1 decreases, the computation is done more quickly, but the communication overhead remains the same. Thus, for smaller T_1, the overhead becomes a greater fraction of t_{par}, and speed-up is reduced.

2. **Message-Passing Architectures** (14 points)
Consider multiplying two $N \times N$ matrices, X and Y, to produce a matrix Z. We can compute Z by computing each element separately: Element $Z(I, J)$ is the dot-product of row I of matrix X with column J of matrix Y. This algorithm performs N^3 "operations" where an operation consists of a multiplication and an addition. For all questions about "time" in this problem, we consider the time for one multiply-and-add to be one time unit. For example, a sequential implementation takes N^3 time units.

We want to compute a parallel version using P message-passing processors. The P processors are arranged as a $\sqrt{P} \times \sqrt{P}$ mesh. Each link in the mesh consists of two, directed edges. Each edge can convey one matrix element per time unit. All edges can convey data in parallel. Furthermore, the network supports *broadcast* and *multicast* messages. If processor p_0 sends the same message to processors p_1, p_2, and p_3, the network can route that one message across links that are common to the paths for these three destinations – it doesn't send the same message three times.

We first consider an algorithm where each processor stores N/P rows of X and Y in its local memory. Each processor will compute N/P rows of Z. To do so, each processor sends *all* of its rows of Y to *all* of the other processors. Likewise, each processor receives all of the other rows of Y from the other processors. As described above, this is done with broadcast messages. After receiving Y, each processor performs N^3/P operations as described above and thus requires N^3/P time for *computation*. In the rest of this problem, we consider the time for *communication*.

(a) (**1 point**) What is the bisection width of the $\sqrt{P} \times \sqrt{P}$ mesh?
Note: this mesh has a total of $\sqrt{P} * \sqrt{P} = P$ nodes.
Answer: $\sqrt{P}$.

(b) (**3 points**) How much time is required to transmit the elements of matrix Y to all processors using this mesh? For this problem, only consider the cross-section bandwidth. Because the network supports broadcast, we only need to consider how much data crosses the bisection (i.e. how many matrix elements) even though this data is delivered to many processors on each side of the bisection.

Answer: $\frac{N^2}{2\sqrt{P}}$.
Explanation:

- Y is $N \times N$ and thus has N^2 elements.

- There are $N^2/2$ elements of Y on each side of the bisection.
- The problem statement says that one element can be transfered across each link per unit time.
- $\therefore$ The total time to transfer Y across the bisection is $N^2/(2\sqrt{P})$.

(c) (**3 points**) Assume that each processor can transfer at most one element to and/or from the network per time unit. This is a constraint on the processor-to-network bandwidth. How long does it take to deliver the elements of Y to each processor?

Answer: $N^2(P-1)/P$.
Explanation:

- Each processor sends N^2/P elements of Y. This takes time N^2/P.
- Each processor receives $(P-1)N^2/P$ elements of Y. This takes time $(P-1)N^2/P$.
- From the problem statement, the data transfers to and from the network can overlap. Thus, the total time is
$$\max(N^2/P, N^2(P-1)/P) \quad = \quad N^2\left(1-\tfrac{1}{P}\right)$$
I'm assuming $P \geq 2$.

(d) (**2 points**) Which constraint, cross-section bandwidth or processor-to-network bandwidth determines the communication time?

Answer: processor-to-network. Explanation: for $P \geq 2$, $1-\frac{1}{P} > 1/(2\sqrt{P})$.

(e) (**4 points**) What is the value for P such that the communication time is equal to the computation time?

Answer:
$$\begin{aligned} \tfrac{P-1}{P}N^2 &= \tfrac{N^3}{P} \\ P-1 &= N \\ P &= N+1 \end{aligned}$$
I'd expect two or three lines of derivation in an answer.

(f) (**1 point**) If P is greater than the answer to Question 2e, which will be greater, the computation time or the communication time?

Answer: communication time.
Explanation: Computation time *decreases* as $1/P$ with increasing P, and communication time increases as $1-1/P$.

3. **Dependencies** (5 points)

(a) (**1 points**) What is a read-after-write (RAW) dependency?

Answer: Assume operation x comes before operation y in program order. Operation x uses a value produces by operation y; therefore, operation x cannot execute until after operation y finishes.

(b) (**2 points**) Dependencies can lower performance by preventing parallel execution of tasks. Consider read-after-write (RAW), write-after-write (WAW), write-after-read (WAR), and control (CTRL) dependencies. Renaming removes or reduces the impact of which of these types of dependencies?

Answer: WAW and WAR.
Explanation: Assume operation x comes before operation y in program order. With a WAW or WAW hazard, operation x reads or writes a register (or variable) that is written by operation y. With renaming, a fresh register (or variable) is allocated for y's write which avoids having y interfere with x. The dependency is removed.

(c) (**2 points**) Branch-prediction removes or reduces the impact of which of these types of dependencies?

Answer: CTRL.
Explanation: Branch-prediction allows instructions that follow a branch to be fetched and executed before the branch is resolved. If the prediction is correct, the results from these speculated instructions can be used. If the prediction turns out to be wrong, the processor rolls-back its state to the mispredicted branch and resumes execution along the correct path.

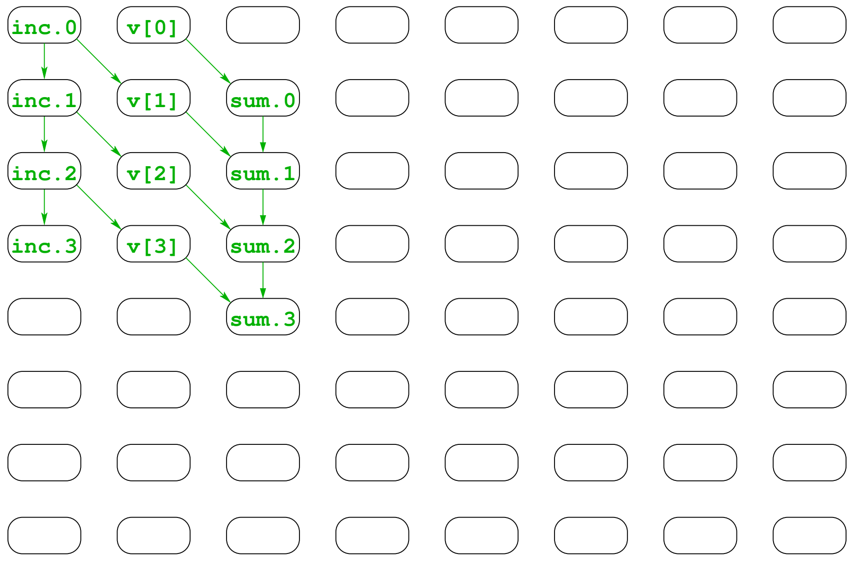
4. **The Work-Span Model** (18 points)

(a) (**4 points**) Consider the following code (in C) for computing the sum of the elements in a vector:

```
// vec_add0(v, n) – compute the sum of the elements of v.
// n is the number of elements of v.
void vec_add0(double *v, int n) {
  int i;
  double sum = 0.0;
  for(i = 0; i < n; i++)
    sum += v[i];
  return(sum);
}
```

Draw a work-span graph for this computation when `n==4` by labeling the vertices, and drawing arrows. All edges should go from a earlier row down to a later row, and each operation should be put in the earliest row allowed by the dependencies. For your convenience, I've provided labels you can use for the vertices. The template has more rows and columns of "vertex boxes" than needed for a correct solution. Only draw edges for RAW dependencies.

label	operation
`v[0]`	load `v[i]`, when `i=0`
`sum.0`	`sum+=`, when `i=0`
`inc.0`	`i++`, when `i=0`
`v[1]`	load `v[i]`, when `i=1`
`sum.1`	`sum+=`, when `i=1`
`inc.1`	`i++`, when `i=1`
`v[2]`	load `v[i]`, when `i=2`
`sum.2`	`sum+=`, when `i=2`
`inc.2`	`i++`, when `i=2`
`v[3]`	load `v[i]`, when `i=3`
`sum.3`	`sum+=`, when `i=3`
`inc.3`	`i++`, when `i=3`



(b) (**1 point**) What is the work for the Work-Span graph from Question 4a?

Answer: 12.
Explanation: one unit of work for each vertex in the graph – i.e. each operation that got a label in the template for the graph.

(c) (**1 point**) What is the span for the Work-Span graph from Question 4a?

Answer: 5.
The critical path goes: `v[0]` → `sum.0` → `sum.1` → `sum.2` → `sum.3` It has five operations; this, the span is five.

(d) (**1 point**) What is the maximum speed-up for the Work-Span graph from Question 4a?

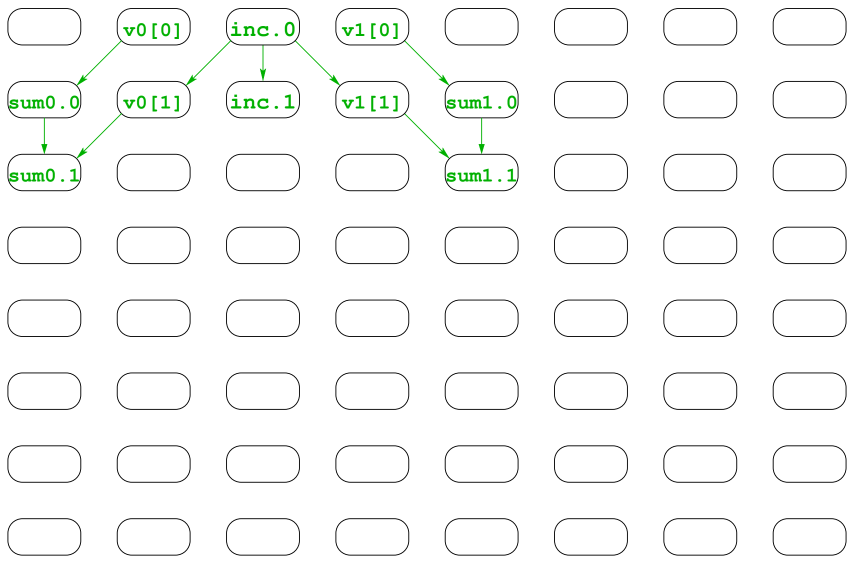
Answer: 2.4
Explanation: $T_\infty = \mathit{Work}/\mathit{Span} = 12/5 = 2.4$.

(e) (**4 points**)

```
// vec_add1(v, 1) – compute the sum of the elements of v.
// Assume: n is even.
void vec_add1(double *v0, int n) {
  int i, n2 = n/2;
  double sum0 = 0.0, sum1 = 0.0;
  double *v1 = &v0[n2];
  for(i = 0; i < n2; i++) {
    sum0 += v0[i];
    sum1 += v1[i];
  }
  return(sum0+sum1);
}
```

Draw a work-span graph for this computation when `n==4` (thus `n2=2`) by labeling the vertices, and drawing arrows. All edges should go from a earlier row down to a later row, and each operation should be put in the earliest row allowed by the dependencies. For your convenience, I've provided labels you can use for the vertices. The template has more rows and columns of "vertex boxes" than needed for a correct solution. Only draw edges for RAW dependencies.

label	operation
v0[0]	load v0[i], when i=0
v1[0]	load v1[i], when i=0
sum0.0	sum0+=, when i=0
sum1.0	sum1+=, when i=0
inc.0	i++, when i=0
v0[1]	load v0[i], when i=1
v1[1]	load v1[i], when i=1
sum0.1	sum0+=, when i=1
sum1.1	sum1+=, when i=1
inc.1	i++, when i=1


$$\begin{tikzcd}
& v0[0] \arrow[dl] & inc.0 \arrow[dl] \arrow[d] \arrow[dr] & v1[0] \arrow[dr] & \\
sum0.0 \arrow[d] & v0[1] \arrow[dl] & inc.1 & v1[1] \arrow[dr] & sum1.0 \arrow[d] \\
sum0.1 & & & & sum1.1
\end{tikzcd}$$

(f) (**1 point**) What is the work for the Work-Span graph from Question 4e?
Answer: 10

(g) (**1 point**) What is the span for the Work-Span graph from Question 4e?
Answer: 3

(h) (**1 point**) What is the maximum speed-up for the Work-Span graph from Question 4e?
Answer: $3\frac{1}{3}$

(i) (**4 points**) I tried running the `vec_add0` and `vec_add1` functions on my laptop with `n==1000` and 1,000,000 trials (a total of 1,000,000,000 additions). Using `vec_add0`, the total time was 1.03 seconds. Using `vec_add1`, the total time was 0.53 seconds. Why is `vec_add1` almost twice as fast as `vec_add0`? A short answer with two or three sentences is sufficient.

Answer: The execution time for `vec_add0` is determined by the chain of dependencies for accumulating `sum`. `vec_add1` breaks this into two, independent chains of half the length. This roughly cuts the span in half.
Geeky superscalar details: A superscalar processor can perform the load, increment, and floating-point add instructions in parallel. The floating point add operation takes several (probably 3 or 4) clock cycles but the floating point unit is pipelined, allowing a throughput of one add per cycle. One might guess that putting more instructions in the loop-body for `vec_add1` is reducing the loop overhead. I tried `'gcc -O3 -S vec_add.c'` and looked at the assembly code. `gcc` unrolls four copies of the for-loop body for `vec_add0` producing four floating point adds and a total of eight instructions per iteration. `gcc` unrolls two copies of the for-loop body for `vec_add1` producing four floating point adds and a total of nine instructions per iteration. Thus, `vec_add1` is performing *more* instructions than `vec_add0`, and the speed-up is due to the greater parallelism due to the computations for `sum0` and `sum1` being independent.

5. **Three fibs** (11 points)
The `fib1(N)` and `fib2(N)` functions below both compute the N^{th} Fibonacci number:

```
fib1(0) -> 0;
fib1(1) -> 1;
fib1(N) when is_integer(N), N > 1 -> fib1(N-1) + fib1(N-2).

% fib2a is a helper function for fib2
fib2a(0) -> {0,1};
fib2a(N) ->
  {FibN_1, FibN_2} = fib2a(N-1),
  {FibN_1 + FibN_2, FibN_1}.

fib2(N) when is_integer(N), N >= 0 ->
  {Ans, _} = fib2a(N),
  Ans.
```

Using `time_it:t` I got the following timing data:

N	time for `fib1(N)`	time for `fib2(N)`	time for `fib3(N)`
10	5.2 μs	2.2 μs	2.0 μs
15	39.5 μs	2.2 μs	2.1 μs
20	435.7 μs	2.4 μs	2.3 μs
25	5.0 ms	2.5 μs	2.4 μs
30	53.8 ms	2.6 μs	2.5 μs
35	579.1 ms	2.7 μs	2.6 μs
40	6.6 s	2.9 μs	2.7 μs
50	timeout	3.1 μs	2.9 μs
100	timeout	4.8 μs	4.3 μs
200	timeout	9.4 μs	8.4 μs
300	timeout	14.3 μs	12.8 μs
400	timeout	18.9 μs	17.3 μs
500	timeout	25.1 μs	21.7 μs

(a) (**1 point**) Which is faster, `fib1` or `fib2`?
Answer: `fib2`

(b) (**2 points**) Give a one or two sentence explanation for why one is faster than the other. Your answer should address the big-$\mathcal{O}$ difference between `fib1` and `fib2`.

Answer: `fib1(N)` is exponential time – it runs in time `O(fib1(N))`. `fib2(N)` is linear time – it runs in time `O(N)`.

(c) (**2 points**) Is `fib2a` head recursive or tail recursive?

Answer: head recursive.
Explanation: `fib2a` performs operations on the result of the recursive call (adding `FibN_1+FibN_2` and constructing a tuple) to produce its return result.

(d) (**5 points**) Write a third implementation of computing the Nth Fibonacci number, call it `fib3`. `fib3` should call a helper function `fib3a`. `fib3` should have the same big-$\mathcal{O}$ complexity as `fib2`, and `fib3a` should be head-recursive if `fib2a` is tail-recursive, and vice-versa.
Hint: I included the run-times for my implementation of `fib3`.

```
% Answer:
fib3a(0, FibN_1, _) -> FibN_1;
fib3a(N, FibN_1, FibN_2) ->
  fib3a(N-1, FibN_1 + FibN_2, FibN_1).

fib3(N) when is_integer(N), N >= 0 ->
  fib3a(N, 0,1).
```

(e) (**1 point**) True, false, or poetry (any non-empty answer gets full credit): "For loops are evil."

[✓ True] – Mark said it.
[✓ False] – I'm an independent thinker.
[✓ Other] – please justify an answer of "Other" with a limerick or haiku.

for loops in Python
imperative, sequential
confusion

I think that my code is the best.
My functions and classes sure nest.
Each **for-loop** is great.
There's nothing to hate.
I hope that we all pass this test.