CpSc 418 **Homework 3** Due: Oct. 19, 2018, 12noon

50 points + 5 points extra credit

There are actually 61 points available (if I added them up correctly this time). You can attempt all 61 if you want. We will cap your score at 55, and treat the assignment as being out of 50 no matter which problems you attempt or skip.

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

cs418 hw3

Your submission should consist of two files:

- `hw3.py`: Python source code for your solution to question 1.
- `hw3.pdf` PDF for the written response parts of your solution.

A template for `hw3.py` is available at
http://www.ugrad.cs.ubc.ca/~cs418/2018-1/hw/3/src/hw3.py.

Please submit code that compiles without errors or warnings. If your code does not compile, we might give you zero points on all of the programming problems. If we fix your code to make it compile, we will take off lots of points for that service. If your code generates compiler warnings, we will take off points for that as well, but not as many as for code that doesn't compile successfully.

1. Branch Prediction **25 points**
   In the September 28 lecture (superscalar architectures), we noted that *branch prediction* is important to allow the processor to fetch instructions beyond a branch and find useful work. We can measure the effectiveness of branch prediction by annotating code to track branch outcomes. Computer architecture researchers use compilers that add such tracking code automatically. For this problem, you will manually add branch tracking to two, simple functions and evaluate the effectiveness of branch prediction for these two examples.

   The approach we will take in this problem is to define a `BranchPredictor` object for each branch in the code. We will have a separate `BranchPredictor` for each if-statement, for-loop, and while-loop in the code. If `br` is a `BranchPredictor`, then we invoke the method `br.branch(True)` to indicate that the branch was executed, and taken, and `br.branch(False)` to indicate that the branch was executed an not taken.

   The branch predictor itself using a 3-bit saturating counter. This is one of the earlier approaches to branch prediction that works fairly well in practice. The counter starts at 0 and can take on values from -4 to $+3$. Each time the branch is taken, the counter is incremented if its current value is less than 3. In other words, the counter *saturates* at 3. Conversely, each time the branch is not taken, the counter is decrement if its current value is greater than -4, i.e. it saturates, at -4. When a branch is encountered, the CPU *predicts* that the branch will be taken if the counter's value is greater than or equal to 0, and the branch is predicted as not taken if the counter's value is negative.

   (a) Branch prediction for `largest` – code **(4 points)**
   The function `largest(x)` finds the largest element of a list. Complete the function `largest_br(x)` that includes branch tracking.

   (b) Branch prediction for `largest` – measurement **(2 points)**
   What is the branch prediction accuracy for the for-loop and for the if-statement for lists of length 10, 100, 1000, and 10000? Use the function `randlist(length, dist)` to construct your lists with `dist=1000000000`. Average over 5 to 10 runs and present your measurements in a table. The variance is pretty high, and we'll take that into account when grading.

   (c) Branch prediction for `merge` – code **(4 points)**
   The function `merge(x)` merges two sorted lists. Complete the function `merge_br(x)` that includes branch tracking.

(d) Branch prediction for `merge` – measurement **(2 points)**
What is the branch prediction accuracy for the while-loop, for the if-statement, and for the two for-loops for merge. Report results where `list1` and `list2` are both of length 100, both of length 1000, both of length 10000. for lists of length 10, 100, 1000, and 10000? Report your measurements in a table. Report your measurements in a table. Again, we know there is variance and will take that into account when grading.

(e) Think. **(7 points)**
Answer each question below with one to three sentences.

i. **(2 points)** What pattern do you see in the predictor accuracy and final counter value for the for-loop for `largest_br`?

ii. **(3 points)** Compare the predictor accuracy for the if-statements of `largest_br` and `merge_br`. Are they similar or do they show different patterns for how they depend on the list length? Give a plausible explanation for your observations.

iii. **(2 points)** What pattern do you see in the predictor accuracy and final counter value for the for-loops for `merge_br`?

(f) Experiment. **(6 points)**
This is the "extra-credit" part of the problem – i.e., you could be entirely reasonable if you think this question is slightly unreasonable.

i. **(2 points)** Implement a subclass of `BranchPredictor` that predicts that the branch outcome will be the same as last time.
Note/hint: this is a 1-bit saturating counter.

ii. **(2 points)** Repeat the measurements for `largest_br` and `merge_br` using this "same as last time" branch predictor.

iii. **(2 points)** What is the most significant impact of the using the "same as last" predictor? Describe which branches are most impacted, and give a plausible explanation for why this happens.

2. Bandwidth and Scalability **(20 points)** Let's say we have some computation that takes $t_1 N \log_2 N$ time on a sequential computer for an instance of size N. Furthermore, assume there is a parallel algorithm such that for P processors, with $P \ll N$, the parallel algorithm does the following:

- The N words of data for the problem are initially distributed across the P processors such that each processor has N/P words. For simplicity, you can assume that N is a multiple of P, P^2 and anything else that is convenient.
- Each of the P processors computes function f_1 on its N/P words. This takes time $t_1(N/P)\log_2(N/P)$
- Each of the P processors sends a separate message to each of the other $P-1$ processors. Each message has size $N/(P^2)$ – ah, that's why I wrote that you can assume N is a multiple of P^2. Analysing the time for communication is the main point of this question. In other words, for every pair of processors, p_i and p_j, there is a message of size $N/(P^2)$ sent from p_i to p_j.
- After receiving these message, each processor computes function f_2 on the N/P words that it received. This takes time $t_1(N/P)\log_2 P$.

The total time for computation is the sum of the times for computing f_1 and f_2

$$\begin{array}{rcccl} T_{compute} & = & (t_1(N/P)\log_2(N/P)) + (t_1(N/P)\log_2 P) & = & t_1(N/P)(\log_2(N/P) + \log_2 P) \\ & = & t_1(N/P)\log_2 N & & \end{array}$$

In other words, we get a perfect speed-up of P if we ignore the time for communication.

Assume the processors are organized as a 2-dimensional mesh. Assume P is a perfect square; so, we have a $\sqrt{P} \times \sqrt{P}$ mesh. From lecture the cross-section bandwidth of the mesh is $\sqrt{P}$. Recall that we

find the cross-section bandwidth by finding how to partition the P processors into two, disjoint sets, A and B with $|A| = |B| = P/2$ and minimizing the number of direct edges from A to B. For the mesh in this problem, there are $\sqrt{P}$ such edges. Assume that transferring W words over one edge takes time $t_2 W$.

(a) **(4 points)** For the computation described above, how many *message* must cross from partition $|A|$ to partition $|B|$?

(b) **(2 points)** Each such message consists of $N/(P^2)$ words. How many words must cross from partition $|A|$ to partition $|B|$?

(c) **(2 points)** Given that each edge can transfer one word in t_2 time units, how many time units are required for communication phase of the computation?

Only consider the time for moving data across the network bisection. For real machines, this can be a reasonable assumption for large values of N.

(d) **(2 points)** If f_1 must be computed before starting the communication, and the communication phase must complete before computing f_2, what is the *total time* for the parallel algorithm?

(e) **(2 points)** What is the parallel efficiency for this parallel algorithm?

(f) **(4 points)** What is the largest value of P such that the parallel efficiency is at least 0.5? Give a formula in terms of N, t_1 and t_2. Then, give numbers for P when $N = 2^{20}$, and $N = 2^{32}$ and when $t_2 = t_1$, and $t_2 = 10t_1$.

(g) **(4 points)**
This is the "extra-credit" part of the problem. Solve Question 2f but assume a 3-dimensional *torus*.

i. **(1 point)** What is the cross-section bandwidth for a $\sqrt[3]{P} \times \sqrt[3]{P} \times \sqrt[3]{P}$ torus?

ii. **(1 point)** What is the speed-up as a function of N, P, t_1 and t_2?

iii. **(1 point)** What is the parallel efficiency as a function of N, P, t_1 and t_2?

iv. **(1 point)** What is the largest value of P such that the parallel efficiency is at least 0.5? Give numbers for P when $N = 2^{20}$, and $N = 2^{32}$ and when $t_2 = t_1$, and $t_2 = 10t_1$.

3. Performance: rules of thumb **(16 points)**

(a) **Amdahl's Law: (4 points)**
A computation has 3% that is inherently sequential. What is the maximum possible speed-up when executed with 20 processors?

(b) **Brent's Lemma: (4 points)**
A computation has work that grows as $N^{3/2}$ and span that grows as $\sqrt{N} \log_2 N$.

i. What is the maximum possible speed-up when $N = 2^{16}$?

ii. What is the lower bound for speed-up using Brent's Lemma when $N = 2^{16}$ and $P = 1024$?

(c) **Energy and Time: (4 points)**
In the October 12 lecture, it was noted that energy and time for a computation can be traded with energy proportional to the inverse of time: if we increase time by a factor of α we can decrease energy by a factor of α, and thus power decreases by a factor of α^2. This tends to involve designing the hardware for a particular energy target, but we won't make you design a CPU for this question.

Let's say we have a task that can be completed in time T using energy E when executed sequentially on one processor. With the trade off that energy is proportional to the inverse of time, which of the following are possible?

i. **(1 point)** Complete the task in 0.5× the time, using 0.25× the energy, with 8 processors, and a parallel efficiency of 1.

ii. **(1 point)** Complete the task in 0.5× the time, using 0.25× the energy, with 16 processors, and a parallel efficiency of 2/3.

iii. **(1 point)** Complete the task in 4× the time, using 0.2 the energy, with 1 processor.

iv. **(1 point)** Complete the task in 10× the time, using 0.06 the energy, with 4 processors and a parallel efficiency of 0.5.

(d) **Moore's Law: (4 points)**
In 1965, Gordon Moore predicted that the number of transistors on a chip would double every year for the next 10 years. He was right. While the growth in transistors has continued, it hasn't quite been doubling every year.

i. **(3 points)** In 1965, it was possible to make a chip with 64 transistors. If Moore's law had continued with the rate of doubling every year, how many transistors would it be possible to put on a chip today?
Please write your answer in scientific notation. We won't worry about the month in which Moore's Law was published (it was April 1965); so, we'll accept answers that are within a factor of two of the "right" answer.

ii. **(1 point)** Today, a large chip is roughly a square that is 30mm on a side. A silicon atom has a diameter (i.e. spacing in a silicon crystal) of about 0.3nm. Transistors are formed on the surface of the chip – i.e. in one, two-dimensional layer. If a chip had as many transistors as in Question 3(d)i, what would be the ratio of the diameter of a transistor to the diameter of a silicon atom today?

Why?

Question 1: Connect ideas from the architecture lecture with how code actually executes. This question should let you see that branches can be "friendly" for branch-prediction, or not. The 3-bit saturating counter is a pretty good branch-predictor that has been used in real CPUs. Today's superscalar architectures use more sophisticated branch predictors that use hashes to track the execution path that led to the branch. Combining path-based methods with counters can achieve even higher prediction accuracies.

The "why" question that I kept asking myself was "Why Python?"

- Using an imperative language makes the branches more obvious.
- I considered using Java, and wrote my first version of the code for this problem in Java. It's about the same length as the Python code. I decided to use Python because this seems like a Python-friendly example: the code is fairly short – the template is a about 250 lines, and my solution adds about 12 more. That's a length where Python's total lack of encapsulation and other software engineering niceties isn't a big issue, but the interactivity of an interpreted language will hopefully entice you to try more experiments.

Question 2: This is a problem to make you think more about bandwidth issues and how bandwidth is crucial for scalability and performance. The computation time and communication pattern for the problem is based on a parallel sorting algorithm based on randomized quicksort.

Question 3: These are intended to be easy problems to give you some familiarity with some of the most useful ways of getting quick estimates of parallel performance and scalability. The Moore's Law question was just for fun. This is intended to be an easy problem: get these points quickly, and then work out your strategy for the rest of the assignment