CpSc 418

Homework 3

Due: Feb. 17, 2017, 11:59pm
Early Bird: Feb. 15, 2017, 11:59pm

140 points.

Please submit your solution using the handin program. Submit your solution as
cs418 hw3
Your submission should consist of three files:

- `hw3.c`: C source code for the coding parts your solution.
- `hw3.erl`: Erlang source code for the coding parts your solution.
- `hw3.pdf` or `hw3.txt`: PDF or plain-text for the written response parts of your solution

Templates for `hw3.c`, `hw3.erl`, and `hw3_test.erl` will be available soon at:
`http://www.ugrad.cs.ubc.ca/~cs418/2016-2/hw/3/code.html`.

Please submit code that compiles without errors or warnings. If your code does not compile, we might give you zero points for the problem. Points will be deducted for compiler warnings.

1. **Twin Primes** (65 points)
   Integers p_1 and p_2 are called twin primes if both p_1 and p_2 are prime, and $p_2 = p_1 + 2$. The twin prime conjecture is that there are an infinite number of twin primes – as of writing this problem statement, this conjecture remains open. However, the sum of the reciprocals of all of the twin primes is bounded (the bound is known as "Brun's constant"). The function `sum_inv_twin_primes(N)` in the template file `hw3.erl` is a sequential computation of the sum of reciprocals of the twin primes for which $p_2 \leq N$.

   In this problem, you will implement a parallel version of `sum_inv_twin_primes(N)`.

   (a) (20 points) Computing the primes that are `=< N` accounts for most of the time when computing `sum_inv_twin_primes(N)` for large `N`. Implement `primes(W, N, DstKey)` that computes the primes that are less than or equal to `N`. The computation should be performed in parallel using the workers of worker tree `W`. The result is a list of primes that is distributed across the workers of `W` and associated with `DstKey`. As an example,

   ```
   W = wtree:create(N_workers).  % e.g. N_workers=16
   hw3:primes(W, N, primes).  % e.g. N=1000
   lists:append(workers:retrieve(W, primes)) == hw3:primes(N). % should be true
   ```

   Hints: You need a way to compute the list of primes in each worker. You *could* do this with `wtree:scan` where `Leaf1` and `Combine` are trivial, for example and the real work is done in `Leaf2`

   ```
   wtree:scan(W,
      fun(_) -> ok end, % Leaf1
      fun(ProcState) -> % Leaf2
         wtree:put(ProcState, DstKey, MyPrimes) % Your job: figure out how to compute MyPrimes
      end,
      fun(_,_) -> ok end, % Combine
      ok % Acc0
   ).
   ```

   This seems like a rather twisted use of `wtree:scan`. A more direct approach is to use
   `workers:update(W, Key, Fun, Args)`
   where `Fun(ProcState, Arg)` is called in each process and returns the value to be associated with `Key`. `Args` must be a list with one element for each worker process. The `N`th worker of `W` is called with `Arg = lists:nth(N, Args)`. For example:

```
workers:update(W, DstKey,
   fun(ProcState, Arg) ->
      workers:put(ProcState, DstKey, MyPrimes) % Your job: figure out how to compute MyPrimes
   end,
   Args
).
```

You may find the function `misc:intervals(Lo, Hi, List)` to be handy:

```
W = wtree:create(6).
Args = misc:intervals(0, 99, W). % Args = [{0,16},{16,32},{32,48},{48,65},{65,82},{82,99}]
```

(b) (10 points) Measure the speed up of your parallel implementation of `primes` with `N=1000000` for `N_workers <- [4,8,16,32,64,128,256]`. Please use the `time_it:t` function for your timing measurements and run your code on `thetis.cs.ubc.ca`. What is the best speed-up your implementation achieves?

Note: by default, `time_it:t(Fun)` executes `Fun()` enough times to take a total of about one second of elapsed time. For the sequential version, this may be just one trial. If the reported average run-time is greater than 0.1 seconds, you can use

```
time_it:t(fun() -> hw3:primes(W, N, DstKey) end, 10).
```

to ensure that the mean and standard deviation are calculated based on 10 runs. If the average run time for either the sequential or parallel versions is more than two seconds (with N=1000000), fix the efficiency problem in your code.

(c) (5 points) Choose the `N_workers` according to the value that got the highest speed-up with `N=1000000`, and measure the speed up of your implementation of `primes` for
`N <- [10,100,1000,10000,100000,1000000]`.
Please use the `time_it:t` functions as described above. How big does `N` need to be to get at least 90% of the peak speed-up?

(d) (15 points) Now, implement a parallel version of `sum_inv_twin_primes`. The function should be `sum_inv_twin_primes(W, SrcKey)` where `W` is a worker tree, and `SrcKey` is the key for a list of primes that is distributed over the workers. For example,

```
W = wtree:create(16).
hw3:primes(W, 10000, primes).
BrunGuess = hw3:sum_inv_twin_primes(W, primes).
```

should set `BrunGuess` to 1.41689. . .

(e) (5 points) Measure the speed up of your implementation of `sum_int_twin_primes` with `N=1000000` for `N_workers <- [4,8,16,32,64,128,256]`. Please use the `time_it:t` functions as described above.

(f) (10 points) Briefly explain the trends you observed in questions 1b, 1c, and 1e. Connect your observations with what we have covered about parallel program performance and performance loss in class and in the readings. If you want to devise a few experiments to test your hypotheses – that's great. Please do, and report what you did and what you found. However, running more experiments than those requested above is not required for this problem.

2. **Architecture** (40 points)
   The file `http://www.ugrad.cs.ubc.ca/~cs418/2016-2/hw/2/src/hw3.c` provides two implementations of merge sort, one using lists and the other using arrays.

   (a) **(20 points)** Complete the function `main` so that the program `hw3` can be run as
   `hw3 init_data data_struct n n_trial`
   where

   `init_data` is either `random` or `ascending` – it specifies whether the data to be sorted is random or an ascending sequence.

   `data_struct` is either `list` or `array` – it specifies whether to run the list or array version of merge-sort.

   `n` is optional. If given, it must be a positive integer – it specifies the number of elements to sort. The default is 1,000,000.

   `n_trial` is optional. If given, it must be a positive integer – it specifies how many times to execute the specified sorting problem to accumulate enough time to make a meaningful measurement. The default is 1.

   Your completion of main should print the time that elapses to perform the requested sort. In particular it should print a line of the form:
   random list n=1000000 n_trial=10 t_avg=2.230e-01
   where `random` indicates that the initial data was "random"; where `list` indicates that the data structure was a list; and `t_avg=2.230e-01` indicates that the average time for sorting the data took 0.223 seconds. Please use getrusage to measure the time before and after running the sort(s) and thereby determine the execution time of the sorting function.
   Warning: This is C, not Erlang. The sort functions are destructive in the sense that the original array or list is modified by sorting. Make sure that when you run multiple trials of random data that *all* of them start with random data, not just the first. Likewise for multiple trials with ascending data.

   (b) **(4 points)** Report the elapsed time for

   i. Sorting an array of 1,000,000 random elements.
   ii. Sorting an array of 1,000,000 ascending elements.
   iii. Sorting a list of 1,000,000 random elements.
   iv. Sorting a list of 1,000,000 ascending elements.

   Please run your trials on `thetis.ugrad.cs.ubc.ca`.

   (c) **(8 points)** Which takes longer: sorting an array of ascending elements or sorting an array of random elements? By what factor? Explain why one is faster than the other; you don't need to explain the specific ratio of run times that you observed.
   Hints: think about branch prediction, cache misses, instruction level parallelism, pipelining, any other good stuff we covered in our architecture review. When in doubt, try modifying the code to test your conjecture(s) and seeing how it impacts the run time. If you do that, summarize your experiments in your explanation.

   (d) **(8 points)** Which takes longer: sorting a list of ascending elements or sorting a list of random elements? By what factor? Is the ratio bigger or smaller than for arrays? Explain your observations.
   Hints: same as the ones for arrays.

3. **Message Passing Networks** (35 points)
Let's say that I want to build a machine with 10^k processing nodes; for example, a "node" might be a multi-core CPU, but we'll just consider "nodes" in this problem. Each node has a 1GByte/second network port, and the routing is done with 16-port crossbar switches. We implement a form of "fat-tree" (if you want to go beyond this homework problem, you can see [Lei85], but this problem statement should be self-contained). Here's how the connections work:

   - We start with 10 nodes and one 16-port crossbar switch. Each node is connected to one port of the crossbar. This leaves 6 crossbar ports for communication to the next level of the tree. We'll call this a 10-node cluster.
   - We then build a 100-node cluster from ten 10-node clusters. We use six 16-port crossbar switches. Each of these crossbar switches has a port for each of the 10-node clusters. This leaves six ports per crossbar switch, a total of 36 ports, for communication to the next level of the tree.
   - We continue in this fashion. Let's say that a 10^k-node cluster has $p(k)$ ports for communication with the next level. From the previous two cases, we know that $p(1) = 6$ and $p(2) = 36$. To construct a 10^{k+1}-node cluster, we use ten 10^k-node clusters, and $p(k)$ 16-port crossbar switches. Each of these crossbar switches has a port for each of the 10^k-node clusters. This leaves six ports per crossbar switch, a total of $6p(k)$ ports, for communication to the next level of the tree.

   (a) (5 points) Derive a formula for $p(k)$.
   Hints: it's simple. The description in the bullet-points pretty much gives it away.

   (b) (5 points) What is the bisection width of a machine with 10^k nodes?

   (c) (10 points) Let the nodes on the "left" side of a bisection have indices $0 \dots 5 * 10^{k-1} - 1$. Let the nodes on the "right" side have indices $5 * 10^{k-1} \dots 10^k - 1$. Consider a case where each node, i, on the left sends a 1Kbyte message to node $i + 5 * 10^{k-1}$. If we only consider the time to transfer the data across the network bisection, how long does it take to send these messages? Give a formula with k, and a specific value for the case that $k = 4$.

   For example, if $k = 4$, then there are 10,000 nodes. The left nodes have indices $0, 1, 2, \dots, 4999$, and the right nodes have indices $5000, 5001, \dots, 9999$. The scenario above has node 0 send a 1Kbyte message to node 5000; node 1 sends a 1Kbyte message to node 5001; $\dots$; and node 4999 sends a 1Kbyte message to node 9999.

   (d) (10 points) If k is even, we can arrange the 10^k nodes as a $10^{k/2} \times 10^{k/2}$ toroidal mesh, where each node has four 1Gbyte/sec links: one to its north neighbour, one to its south neighbour, one to the east, and one to the west. What is the bisection width of a toroidal machine with 10^k nodes?

   (e) (5 points) Again, we can assign indices to the nodes of the toroidal machine as described above for the fat-tree. Consider a case where each node, i, on the left sends a 1Kbyte message to node $i + 5 * 10^{k-1}$. If we only consider the time to transfer the data across the network bisection, how long does it take to send these messages? Give a formula with k, and a specific value for the case that $k = 4$.

Why?

Question 1: The parallel implementation of the prime sieve is an embarrassingly parallel problem. I want you to get experience with "easy" parallelism. It also provides a nice example for many of the performance issues that we've been covering in class.

Question 2: The obvious reason for this problem is to give you some hands-on experience with the architecture issues covered in lecture and readings. It also lets you see how we can connect analytical results with actual performance – where practice matches theory, and where it doesn't.

A second motivation is that we will be programming in C with CUDA in the second half of the term. This problem gets you measuring timings in C. It is a chance for the instructors and TAs to make sure we've got the right set-up for office hours, tutorials, and grading as we make this transition.

Question 3: The book didn't cover fat-trees, but you should see them at least once before the term is over. This gives you a chance to see some of the trade-offs when designing a network topology for parallel computing.

This problem also has a second motivation. We've got a midterm coming up. I haven't figured out a way for you to write real code on real computers using real development environments during an exam. frownie. Accordingly, I figured I should start giving some problems that can be solved without using a computer.

References

[Lei85] C. E. Leiserson. Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, 34(10):892–901, October 1985.