AM 225: Homework #1

Kevin Li 15 February 2019

Preliminaries

The code is found in the code folder; all of the code is mine, except for two external libraries. The first external library (called `fmt`) is a pretty-format library similar in spirit to Python's format function e.g.,

```
fmt::print("Hello, {}.\n", "Chris");
```

instead of more verbose `std::cout` statements. The second external library (called `cleantype`) allows for easy printing of STL containers. Clearly, neither of these libraries are essential to the logic of the problems.

Inside the code folder, type `make` to generate executables. You will need a C++14 compatible compiler; on my Linux system, I use GCC 7.3.0. The executables all have a `.bin` extension, i.e.,

```
<name>.bin,      for example: one.bin, two-a.bin, etc.
```

The source code is named in a self-explanatory way (e.g., `problem4c.cpp`). There is a small utility class called `timer` which prints out the execution timer in milliseconds.

1 Problem 1

1.1 Part (a)

According to the simulations, the expected winnings (in dollars) is 21.8308 per round. Therefore, the game is worth playing (for a risk-neutral agent). The running time (in milliseconds) are

```
threads time    profit
1       28786.8 21.8335
2       14410.8 21.8276
4       7203.43 21.826
```

This is good: we have roughly linearly scaling with the number of threads. (The third column is redundant.)

1.2 Part (b)

The following lemma is easily proven by induction.

Lemma. Let $U_1, \ldots, U_n$ be independent random variables uniformly distributed on the unit interval $[0,1]$. For any $t \in [0,1]$

$$\mathbf{P}(U_1 + \cdots + U_n \leq t) = \frac{t^n}{n!}. \tag{1}$$

We draw strictly more than n numbers in the lottery game if and only if $U_1 + \cdots + U_n \leq 1$. This occurs with probability $1/n!$, so that the expected number of draws[1] is

$$\sum_{k=0}^{\infty} \frac{1}{n!} = e. \tag{2}$$

The expected winnings therefore $100e - 250$ dollars. This matches with our answer in **Part (a)**.

[1] Recall that for a non-negative integer-valued random variable X,

$$\mathbf{E}X = \mathbf{P}(X > 0) + \mathbf{P}(X > 1) + \mathbf{P}(X > 2) + \ldots.$$

2 Problem 2

Note. See the files `grid.h` and `grid.cpp` for this problem. The files `problem2a,b` just calls the code in the grid class.

2.1 Part (a)

Run the `two-a.bin` executable to see a print out of the grid. For convenience, I piped that to the file `problem2a.txt`.

2.2 Part (b)

The raw data could be generated by running `two-b.bin`. I calculate $p(n,T)$ manually in Excel, and the result is saved in `problem2.csv`. The attached `R` code generates the graphics in **Part (c)**.

2.3 Part (c)

See Figure 1 on page 3. We see two patterns

1. For fixed n, increasing the number of threads generally decreases thread efficiency. This makes sense: since I lay out my grid in a one dimensional array, there is some degree of false sharing, where multiple threads need to access the same cache line at the same time. There is of course also a penalty for starting threads in the first place.
2. For fixed T (number of threads), thread efficiency increases as n grows. This also makes sense: there is more parallelism to exploit in larger problems (the fixed cost of spawning a thread becomes less significant). To some degree, the false sharing effect is also alleviated in larger grids (i.e., fewer percentage of accesses exhibit false sharing).

3 Problem 3

Note. The code in `problem3.cpp` and `problem3.h` include the helper functions. The code for the parts are in `problem3a.cpp`, `problem3b.cpp`, and so on.

3.1 Part (a)

See the text file `primes.txt` for the list of primes; there are 17984 of them.

3.2 Part (b)

The long division code is in `problem3.cpp`, under the function `long_division`. The file `problem3b.cpp` does a small example that I checked in Mathematica.

3.3 Part (c)

In the base $B = 2^b$, the number $2^n - 1$ may be represented as

$$2^n - 1 = (2^r - 1)(B^q) + (B-1)B^{q-1} + (B-1)B^{q-2} + \cdots + (B-1)B + (B-1) \tag{3}$$

where $0 \le r < B$ and $q \ge 0$ are the unique integers satisfying

$$n = bq + r. \tag{4}$$

My long division algorithm uses `uint64` as the underlying integer representation and the algorithm itself requires that $B^2 - 1$ fits inside the presentation. Therefore, I chose $B = 2^{32}$, i.e., $b = 32$.

As expected, there are no prime factors of the Mersenne prime $2^{82589933} - 1$. (The problem `three-c.bin` prints every prime factor, and its produces nothing.)

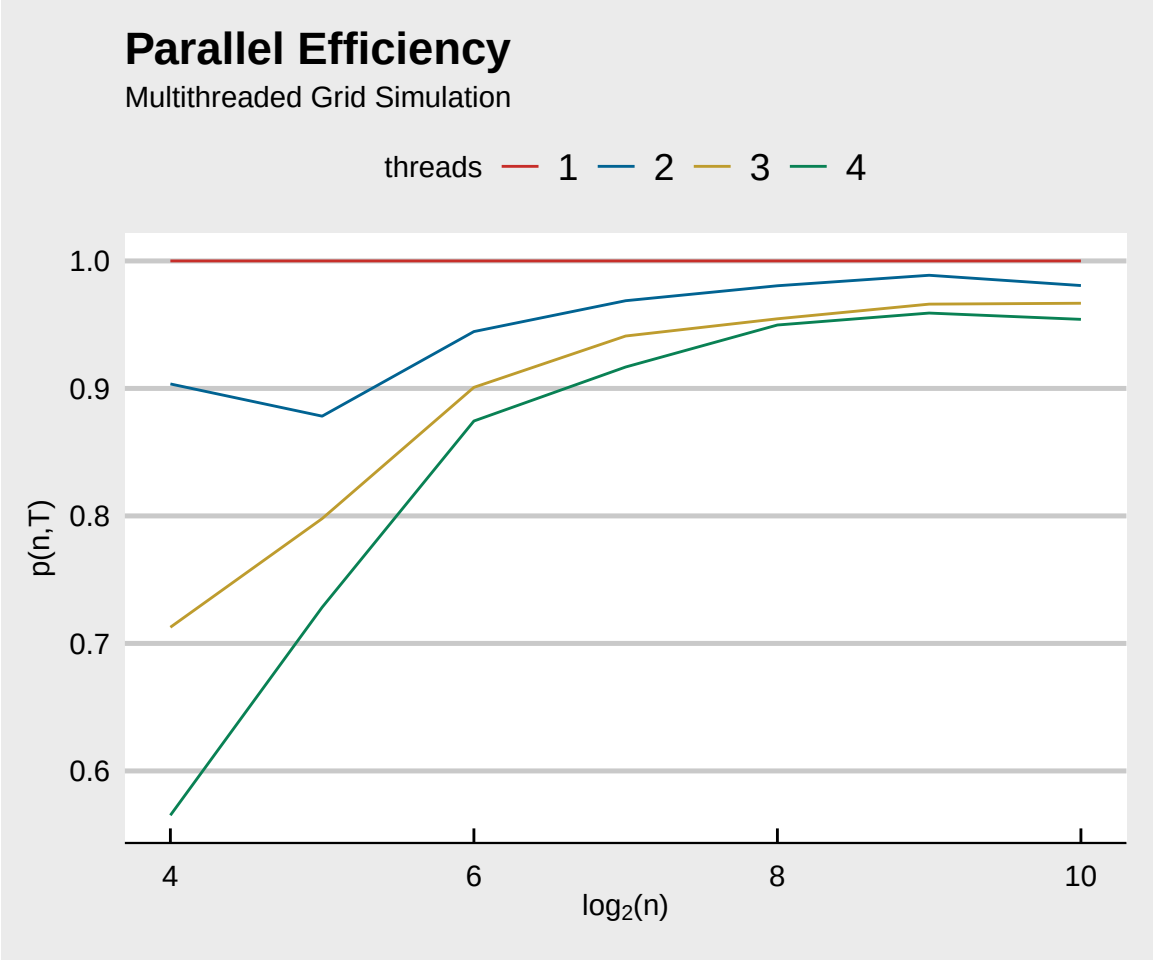

Figure 1: Problem 2c–Parallel efficiency.

3.4 Part (d)

My program finds prime factors 3, 5, and 41201 as the prime factors of N below one million. This does take a long time to run, because my long division code is inefficient. (On my fairly quick six-core desktop, I waited about 9 minutes.)

4 Problem 4

Note. I made some minor changes to Algorithm 1 to make it more efficient. The issue is that the given algorithm is too slow to finish in Part (d) in a reasonable amount of time.

The exact modification is that each the board is actually *copied* in every recursive call, thus there is no need to remove the guess j; the sudoku board simply goes out of scope. Moreover, every time a new guess is entered, there is some bookkeeping in the background to make sure that board is as reduced as possible. Specifically, this means: if a newly entered guess in square j forces the cell k to admit only one legal value, then the cell k is filled in. If this causes another cell ℓ to admit only one legal value, then ℓ is filled in as well, and so on.

The code is found in `sudoku.cpp` and `sudoku.h`. Note that `ChronoCube` is my GitHub handle, hence the namespace name.

4.1 Part (a)

According to the timing produced by `four-a.bin`, solving the puzzle once takes approximately 0.07 milliseconds. Pretty quick!

4.2 Part (b)

There are 283576 solutions, as verified. (Run the problem `four-b.bin`.)

4.3 Part (c)

I timed **Part (d)** instead of **Part (c)**, since **Part (c)** finishes so quickly (one tenth of a second) that launching parallel threads overwhelms actual computation.

The parallel algorithm only parallelizes in the top level of the tree. I tried to use OpenMP tasks[2] to parallelize the entire tree; it turns out that simply parallelizing the top part of the tree yields better performance.

The timings are 10 seconds; 5.9 seconds; and 3.9 seconds for 1, 2, and 4 threads (respectively). (The speed gains are not very impressive.)

Note. The second argument to `count_solutions_parallel` controls the number of threads. Simply change the call in **problem4c.cpp** and recompile and use the shell builtin to time.

4.4 Part (d)

There are 4347232 solutions to the puzzle, as per the verification in **Part (c)**.

[2]Using the task construct of OpenMP seems to be required here because the function we are parallelizing is recursive.