Lab 2 CS160

Felix M. Diaz

University of California, Riverside

CS160

Pat Miller

10/30/2024

Abstract

In this lab report, I will explore my approach and results in attempting to speed up the computational code found in the code file hotplate.cpp for Lab 2. In addition to code, and results, I will share a graphical representation of the change in runtime across iterations of the program as I continue to explore additional means of speedup. Finally, I will share what I believe to be the Amdahl limit of this program and a justification for why I believe this to be the case.

Testing Environment

Before I dive into a deconstruction of my approach per iteration, I would like to establish some truths that will persist throughout this report. While we were given the option to SSH into *bolt.cs.ucr.edu* to make use of the shared HPC for computing, I have chosen to run my tests on a local machine running an instance of Ubuntu via WSL.

Method

To calculate runtime results for each iteration of the program, I ran every generation a total of 12 times, discarding the fastest and the slowest runs as outliers. I then averaged the total runtime of the remaining 10 runs to achieve a global average runtime for each iteration of the program at every instance. I began this process using 1 singular thread, increasing the number by 1 until I stopped making any noticeable gains in performance. At this point, I pivoted to a new starting point of 8 OR 16 threads depending on performance thus far, and began increasing by 4 threads rather than 1. This way I allowed myself to efficiently chart the performance.

I modified the hotplate program to write the results (runtime) of the program in a separate file, allowing me to easily grab the 12 values, average them, and then use a simple plotting program to visualize the results of each run.

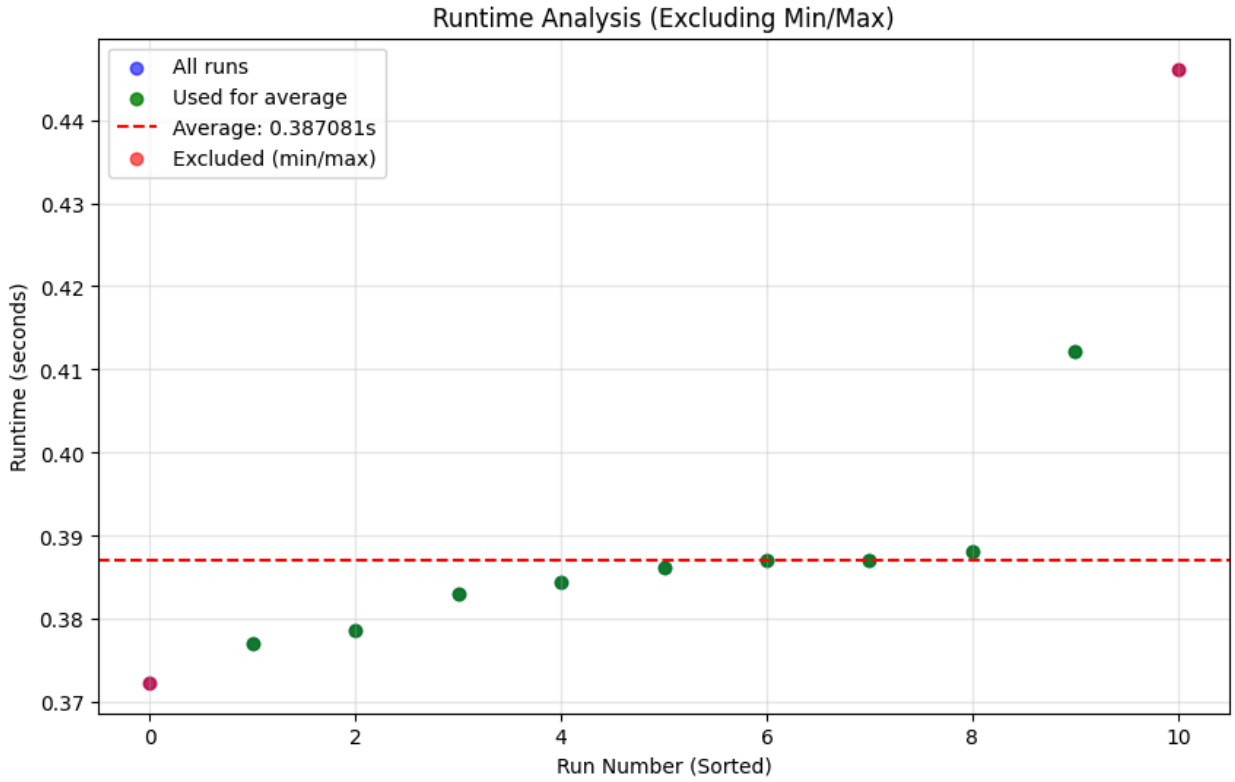
All tests were performed on a **200x200** hotplate seen in this command, with accompanying seed values:

```
./hotplate 200 10 10 10 10 10 100 100 100 10 1000 output.dat nTsk
```

Starter Code (hotplate.cpp)

Using the starter code, I chose to only graph the performance using 1 core, based on the idea that this code is not optimized for multithreading. These runs serve as more of a baseline observation for any gains made later on in the analysis.

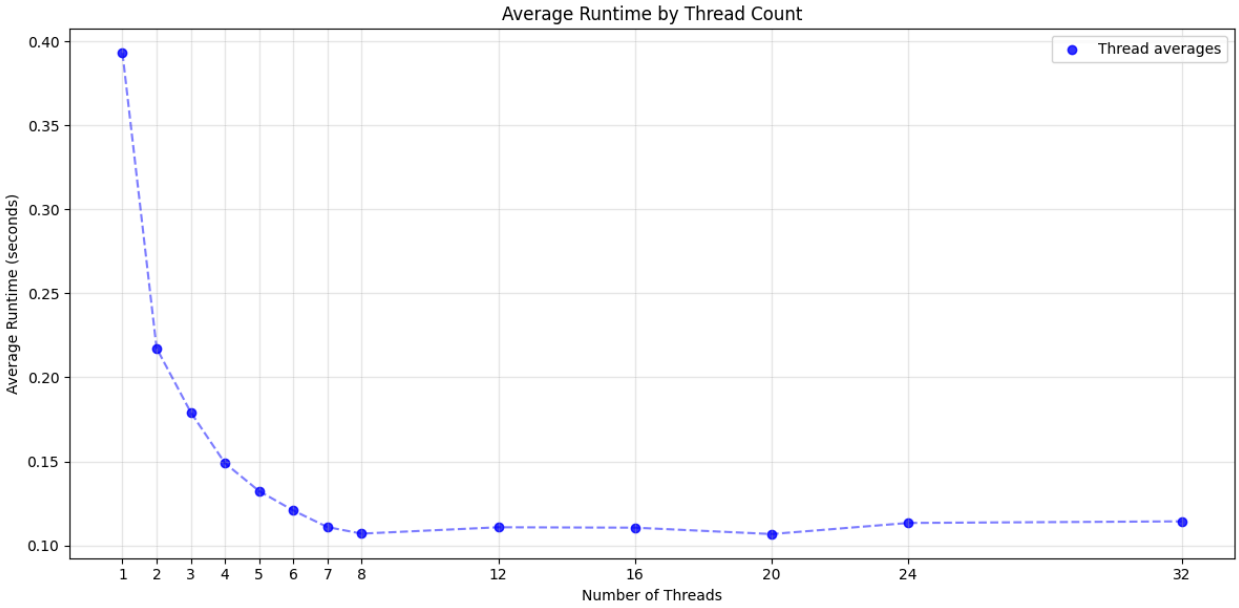
```
felixmdm@LAPTOP-1R6H2O21:~/code_linux/cs160/Lab2$ g++ hotplate.cpp -o hotplate
felixmdm@LAPTOP-1R6H2O21:~/code_linux/cs160/Lab2$ ./hotplate 200 10 10 10 10 10 100 100 100 10 1000 output.dat 1
1 0.377015
felixmdm@LAPTOP-1R6H2O21:~/code_linux/cs160/Lab2$ ./hotplate 200 10 10 10 10 10 100 100 100 10 1000 output.dat 1
1 0.388068
felixmdm@LAPTOP-1R6H2O21:~/code_linux/cs160/Lab2$ ./hotplate 200 10 10 10 10 10 100 100 100 10 1000 output.dat 1
1 0.38612
felixmdm@LAPTOP-1R6H2O21:~/code_linux/cs160/Lab2$ ./hotplate 200 10 10 10 10 10 100 100 100 10 1000 output.dat 1
1 0.384489
felixmdm@LAPTOP-1R6H2O21:~/code_linux/cs160/Lab2$ ./hotplate 200 10 10 10 10 10 100 100 100 10 1000 output.dat 1
1 0.372388
felixmdm@LAPTOP-1R6H2O21:~/code_linux/cs160/Lab2$ ./hotplate 200 10 10 10 10 10 100 100 100 10 1000 output.dat 1
1 0.378661
felixmdm@LAPTOP-1R6H2O21:~/code_linux/cs160/Lab2$ ./hotplate 200 10 10 10 10 10 100 100 100 10 1000 output.dat 1
1 0.446113
felixmdm@LAPTOP-1R6H2O21:~/code_linux/cs160/Lab2$ ./hotplate 200 10 10 10 10 10 100 100 100 10 1000 output.dat 1
1 0.412152
felixmdm@LAPTOP-1R6H2O21:~/code_linux/cs160/Lab2$ ./hotplate 200 10 10 10 10 10 100 100 100 10 1000 output.dat 1
1 0.382962
felixmdm@LAPTOP-1R6H2O21:~/code_linux/cs160/Lab2$ ./hotplate 200 10 10 10 10 10 100 100 100 10 1000 output.dat 1
1 0.387163
felixmdm@LAPTOP-1R6H2O21:~/code_linux/cs160/Lab2$ ./hotplate 200 10 10 10 10 10 100 100 100 10 1000 output.dat 1
1 0.387996
felixmdm@LAPTOP-1R6H2O21:~/code_linux/cs160/Lab2$ ./hotplate 200 10 10 10 10 10 100 100 100 10 1000 output.dat 1
1 -0.255446
felixmdm@LAPTOP-1R6H2O21:~/code_linux/cs160/Lab2$
```



Threaded Code (hotplate_threaded.cpp)

Results. For the first iteration of the threaded approach, I will not be graphing every individual run per thread, but rather graph the average of every run per thread using the same

method seen above for the starter code. More specifically, I will run the code using 1 thread 12 times, average it, and continue upwards as I've outlined in the methods section. I'll save each of these averages in a list, and then graph these averages to represent the change in runtime as threads change. This way we can begin to visualize at what point exactly performance begins to drop off and estimate when the law of diminishing returns begins to set in and it becomes redundant to keep throwing threads at our program, in its current state, at least.



Based on the graph, I noticed that it became redundant to add additional threads somewhere around the **8-12** thread mark.

Threading Strategy. For the most part, this code is very similar to the starter code. The major changes that enabled me to utilize multiple threads were changes made in the task objects themselves. Instead of having 1 process compute the entirety of the grid, I tried to have multiple threads that would assign themselves to compute **portions** of the grid. To do this, I had to modify the task object to specify how this portion would be computed (the start and ending index).

Calculation. In order to calculate the starting index and the ending index of the subset of indices that any one task will occupy, I used a method similar to the first questions on HW 1. For reference, below is my response,

> When n is divisible by p, all cores perform equal work. So we can represent the first index that will be assigned to core 'k,' my_first_i, and the last index assigned to core 'k,' in a given array of tasks as my_last_i, such that:
>
> elements_per_core = n/p
> my_first_i = k * elements_per_core
> my_last_i = (k + 1) * elements_per_core - 1
>
> where my_first_i points to the starting location for a given core based on the number of tasks, and my_last_i points to the starting location of the next core, minus 1.
>
> In the event that the n is not divisible by p, then we would want to handle the overflow. We can do so by evenly calculating the 'overflow' jobs and redistributing them across the initial cores.
>
> # assuming that we are looping over and counting what core we are accessing
>
> elements_per_core = n/p
> overflow_jobs = n % p
>
> my_first_i = k * elements_per_core + (1 if cores < overflow_jobs else overflow_jobs) + (-1 if cores == 0 else 0)
> my_last_i = k+1 * elements_per_core + (0 if cores < overflow_jobs else overflow_jobs - 1)
>
> for the starting index, we make sure that we calculate first k * elements_per_core to gauge roughly what the starting index would be, disregarding the overflow. Then, if we still have overflow jobs to distribute, we ensure that the starting index of the jobs is 1 more than it otherwise would, in order to account for the extra job the core before it would be performing (Excluding the starting core).
>
> For the ending index, we calculate it as per usual, except, to ensure that we account for the extra job the one preceding it would calculate, we omit the -1 if there is still overflow, otherwise add the number of overflow jobs -1 to the ending index.

Using a similar strategy, inside the *execute():* member function of the *Task* object, before the stencil is applied to the object, I define the starting and ending index as, follow,

```
int start_row = (N / ntasks) * id + 1;
```

```
int end_row = (id == ntasks - 1) ? N - 1 : start_row + (N / ntasks);
```

We can define the starting row by dividing *N* (the size of the grid) into *ntasks* (the number of tasks we would like to create), and then multiply this by the ID of the tasks we've created + 1. This technique maps it to an index of the **flattened grid** that is our NxN hotplate. The *ending index* of this subset of work is similarly calculated by first checking if we are at the last task to be processed, and if so, setting the ending index equal to this final boundary. Otherwise, we use a similar division strategy as seen in the *starting index* calculation; however, this time it is offset by the *starting index*.

Thread initialization and assignment. Now that we've defined the tasks that will allow us to carry out our work in a concurrent manner, we need to modify the main function of the program to reflect these changes and handle the new operations accordingly. In previous iterations of the code (the starter code), this was handled by creating an array of tasks, and an array of threads (fixed size of 1) and assigning each thread, its respective task. Instead of creating a vector of 1 fixed task,

```
std::vector<Task> tasks;
tasks.emplace_back(0,1, to,from,N,steps);
```

We create an array of threads that actually reflects our threading implementation,

```
std::vector<Task> tasks;
for (int i = 0; i < ntasks; ++i) {
 tasks.emplace_back(i, ntasks, to, from, N, steps);
}
```

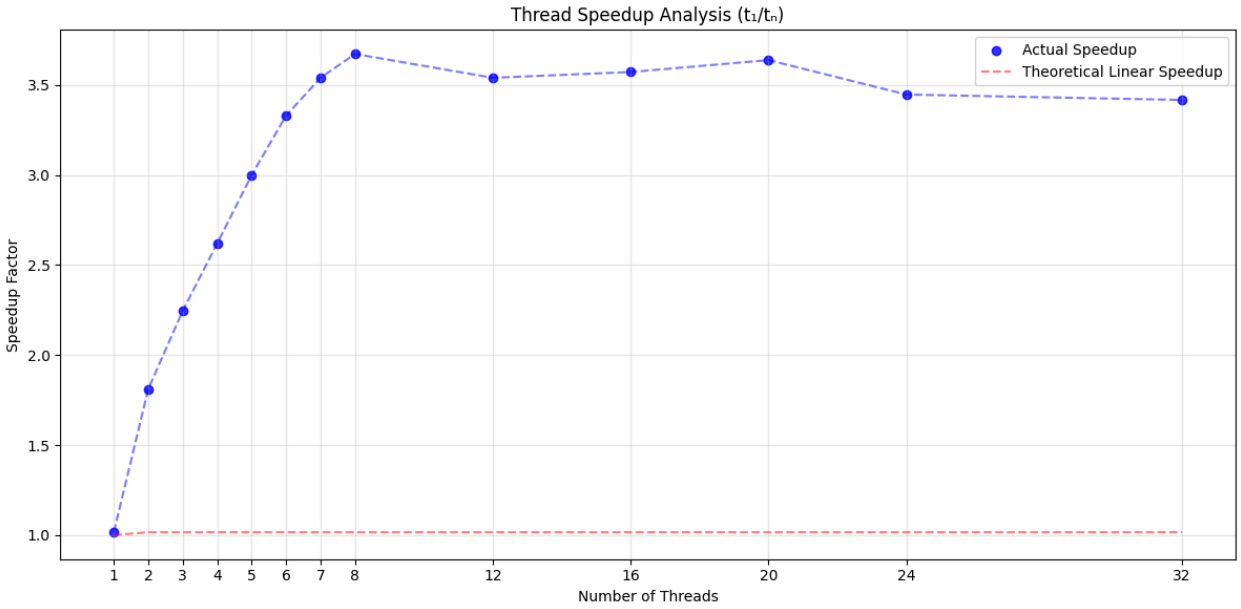
In this way, once we begin to loop from *0* to *tasks* in the code that follows, we can assign each thread an appropriate subset of tasks,

```
for(int i=0; i < ntasks; ++i) {
  int status = ::pthread_create(&threads[i], nullptr, Task::runner, &tasks[i]);
  if (status != 0) {
    ::perror("thread create");
```

```
    return 1;
  }
}
```

The remainder of the code operates the same as the starter code example, the only difference is that I included simple functionality to write the output of the run (runtime) to a separate CSV file, such that I was easily able to copy the runtime of each 12 runs to a *.ipynb* that helped me to visualize the data, averaged, and graphed across their respective xy values.

The Amdahl's Limit. From the data we've calculated so far, we can attempt to calculate the Amdahl limit to get the *theoretical* maximum speedup based on the trend we notice in the data. To estimate the speedup, I tried to get the *speedup factor*. I graphed this by plotting [(t1/tn)/x] where t1/tn is the average runtime for 1 core, which we previously found to be **~393ms**. These y values are then graphed over the respective cores where n represents the number of cores used. This yields the graph seen below,



Based on the trajectory of the program's runtime, we can estimate the maximum speedup that we might possibly achieve, which sits somewhere around 4x.

Semaphore Scheduling

During the past 2 weeks, we talked briefly about the notion of semaphores which, are signals that allow us to indicate whether a thread is open or closed, where open indicates that it is ready to work, and closed indicates that work is currently being done, or for any other reason, it is unable to take on a new job. Had I a better understanding of implementing semaphores, I would have liked to use them to speed up the threaded hotplate. The direction I was originally headed in was to use semaphores to denote whether the tasks we defined in a hotplate, are busy or not. From there, we can reassign any threads who'd completed their job, to a new job. In the process of doing so, theoretically accelerating the calculation.

Results

Overall, I saw that by using a static partitioning approach to the multithreading of this program, I was able to achieve a very tangible speedup as I increased the number of cores. Though I did not get the chance to fully implement a working dynamic scheduler, based on what I saw, I estimate that its speedup graph may look sharper than the one we observed in this report. I look forward to continuing to mess with this code and hopefully successfully implementing a dynamic scheduler. I strongly believe that using the approach I outlined in the section titled *Semaphore Scheduling*, I can achieve a much larger speedup than what I observed throughout this assignment.