Parallel Computing Final Project

Bryce Harmsen

June 5, 2020

Introduction

The goal of this project is to implement a parallel program in OpenMP, MPI, or CUDA to solve either problem 3.13, 3.14, 3.15, or 3.16 from *Parallel Programming in C with MPI and OpenMP*.

Problem Description

Problem 3.15

You are given an array of n records, each containing the x and y coordinates of a house. You are also given the x and y coordinates of a railroad station. Design a parallel algorithm to find the house closest to the railroad station (as the crow flies).

Design

The parallelization of this problem is based on Foster's methodology.

Partitioning

Here we will assign a primitive task for each of the n records. Each task will hold the (x, y) coordinates of a house as well as the (x, y) coordinates of the railroad station. Each task will calculate the Euclidean distance between its assigned house and the railroad station.

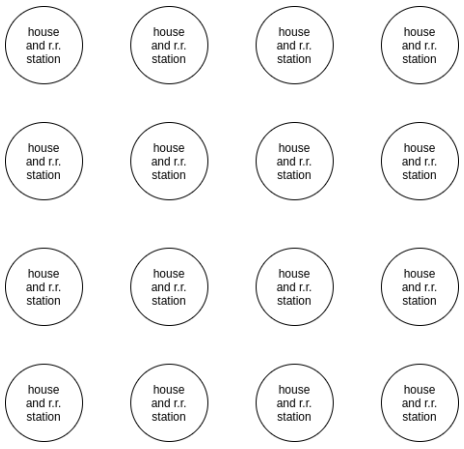

Figure 1: Partitioning into primitive tasks

Communication

During the communication step, primitive tasks will communicate their distance to another primitive task. The primitive tasks will then keep the house record with the minimum distance. This communication and minimum calculation will continue until there is only one task remaining. This task will end up with the closest house.

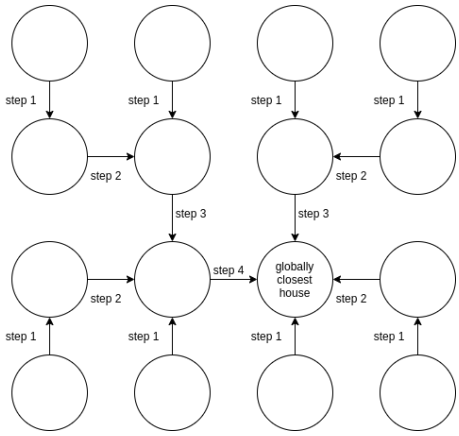

Figure 2: Communication between primitive tasks

Agglomeration and Mapping

Now we can group the houses into p balanced groups, where p is the number of available processes and each group has either $\lfloor \frac{n}{p} \rfloor$ or $\lceil \frac{n}{p} \rceil$ houses. Each process calculates the distance to the railroad station for each of its internal houses, then calculates its internal closest house by finding the house with the minimum distance to the railroad station. This minimum distance and closest house for half the processes are then communicated to the other half of the processes. The receiving processes then compare this new house to their current minimum house. Again, half of the processes pass along their minimum house to the other half of the remaining processes until only one process has a minimum house record.This will be the closest house among all processes and among all n records.

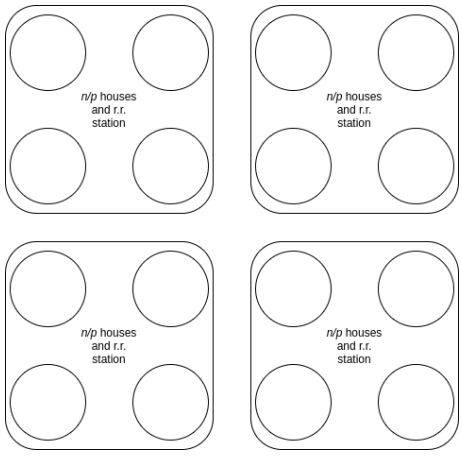

Figure 3: Agglomerations of primitive tasks assigned to processes

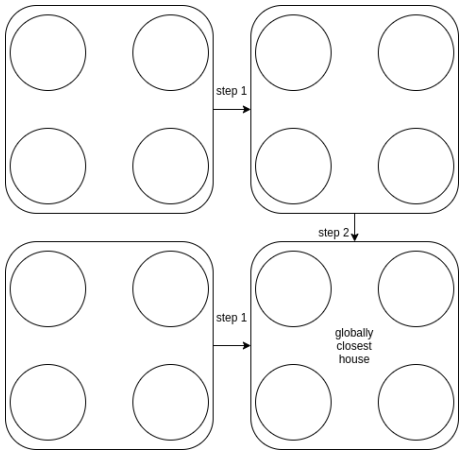

Figure 4: Mapping between processes

Implementations

While most implementations have their own improvements and structures, an improvement that is present throughout all implementations is in the `get_dist_sq` function. Because we only compare distances to each other here, the costly square root operation has been removed from the Euclidean distance formula for performance.

An Unproductive Approach with omp_critical_1

While this is the simplest implementation of threads, there are no speedups, and in fact the code is slower as the number of threads increase. The results are not shown here.

```
omp_set_num_threads(p);

#pragma omp parallel for private(i, house, dist)

for (i = 0; i < n; i++) {

  house = houses[i];

  dist = get_dist_sq(railroad, house);

  #pragma omp critical

  {

    if (dist < closest_dist) {

      closest_dist = dist;

      closest = house;

    }

  }

}
```

An Incorrect Approach with omp_critical_2

The next implementation, though still simple, is the incorrect code that generated the results for figure 5. Notice in the example below that, though there is a critical section around closest_dist and closest, there is the possibility (and if you run the code enough times it will happen) that a non-optimal value will pass the `dist < closest_dist` test, be paused at the critical section, and possibly overwrite the optimal value. This would require the non-optimal value to pass the conditional before the optimal value reassigned the `dist` variable, but still late enough that it is in the wait queue for the critical section after the optimal value. While it is rare, it does happen.

```
omp_set_num_threads(p);

#pragma omp parallel for private(i, house, dist)

for (i = 0; i < n; i++) {

  house = houses[i];

  dist = get_dist_sq(railroad, house);

  if (dist < closest_dist) { //this test should be synchronized with the assignments below

    #pragma omp critical

    {

      closest_dist = dist;

      closest = house;

    }

  }

}
```

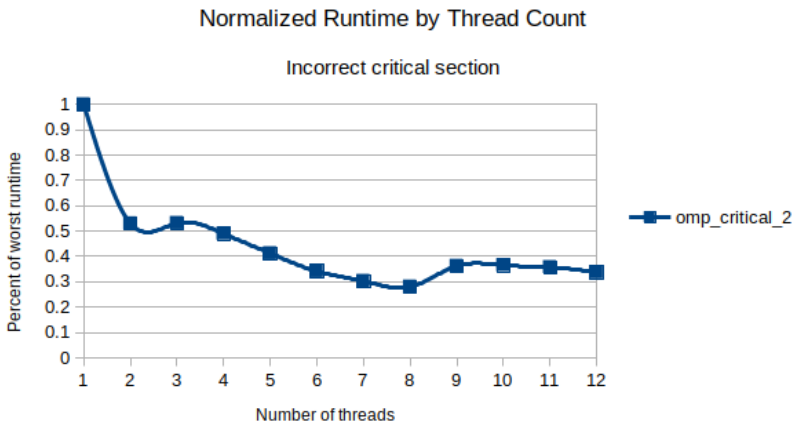

Figure 5: Incorrect critical section implementation

A Quirky Correction with omp_critical_3

The unfortunate downside of the straightforward fix for **omp_critical_2** (encompassing the `if(dist < closest_dist)` statement in a critical pragma) will bring the code to a screeching halt as mentioned above in the first implementation, producing far worse results as the number of threads increase. So the question is, can we reap the benefits of the structure shown above while still providing a correct solution? Here is an attempt to do so:

```
omp_set_num_threads(p);

#pragma omp parallel for private(i, house, dist)

for (i = 0; i < n; i++) {

  house = houses[i];

  dist = get_dist_sq(railroad, house);

  if (dist < closest_dist) { //first comparison prevents calls to critical

    #pragma omp critical

    {

      if (dist < closest_dist) { //same comparison to verify dist is still closest

        closest_dist = dist;

        closest = house;

      }

    }

  }

}
```

This quirky repetition of the `dist < closest_dist` comparison allows for often far fewer calls to the critical section while still avoiding the rare cases where a non-optimal solution is waiting on an optimal solution for access. But, as we can see in figure 11, the results do not quite measure up to any of the other correct approaches, though the speedups are noticeable in 6.

Starting Reduction with omp_reduction_1

Now we start relying on the OpenMP implementation of reduction. As of OpenMP 3.0, min reduction is available (though the user-defined min reduction is still possible in OpenMP 2.x). This is all rather straightforward, though there is still no implementation of the explicit agglomeration described in the parallelization through Foster's methodology as described above. Notice, though, that there is an unfortunate need for a final loop back through all of the houses to determine which house owned the closest distance that was found. Had this problem only been to find the closest distance, this code might have given the best performance, but note that the problem asks us for "a parallel algorithm to find *the house* closest to the railroad station". Because of the type limitations to reduction, we cannot pass the whole `double * houses[i]` array through the reduction. Otherwise, we could perform the reduction comparison on the `dist = houses[i][2]` variable and allow the reduction reassignment to pass the value of `houses[i]` to the shared variable `double * closest`, preventing the need for the final `for` loop.

```
omp_set_num_threads(p);
```

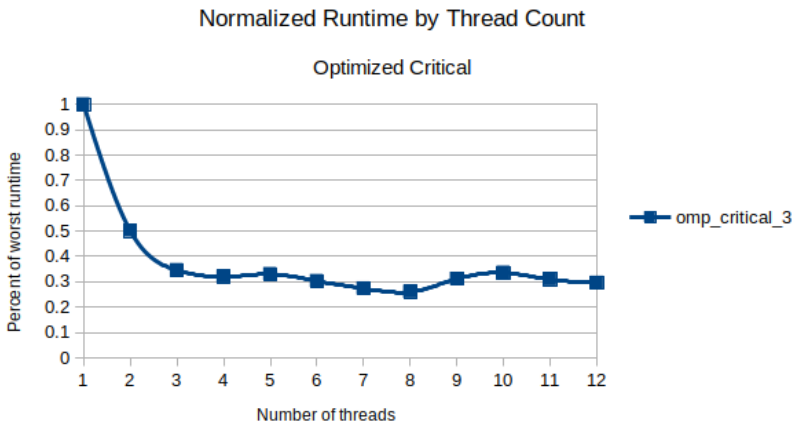

Figure 6: Optimized critical section

```
#pragma omp parallel for private(i, dist) reduction(min:closest_dist)
for (i = 0; i < n; i++) {
  dist = get_dist_sq(houses[i], railroad);
  houses[i][2] = dist;
  if (closest_dist > dist)
    closest_dist = dist;
}
//a necessary evil of finding the closest house, not the shortest distance
for (i = 0; houses[i][2] != closest_dist; i++) {}
return houses[i];
```

Improving the Search with omp_reduction_2

Next, we see some small gains by parallelizing the `for` loop from the previous reduction. While this parallelization allows us to split the search among multiple threads, we cannot exit the search loop early, so the gains depend on where the optimal house is within the `houses` array. If the optimal house is beyond position $\frac{n}{p}$, then this parallelized `for` loop will reach it in less iterations; if the optimal house is between indices 0 and $\frac{n}{p}$, then the early termination of the search loop in **omp_reduction_1** will be faster. Below is only the improved for loop. The rest of the code is the same as the last example.

```
omp_set_num_threads(p);
#pragma omp parallel for
for (i = 0; i < n; i++) { //a game of chance
  if (houses[i][2] == closest_dist)
  closest_i = i;
}
```

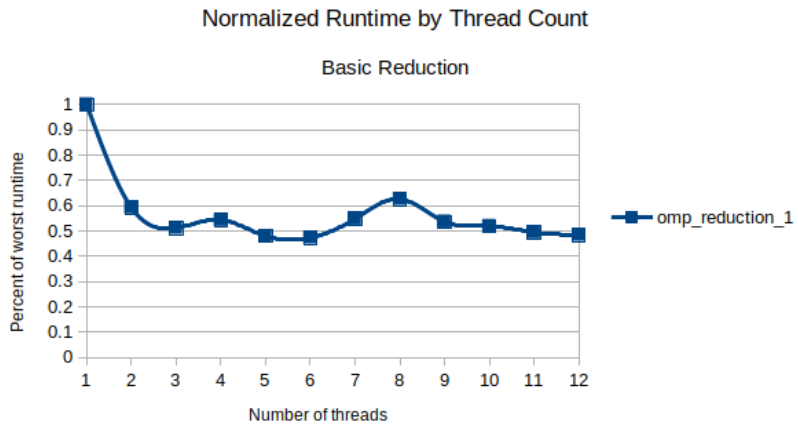

Figure 7: Basic reduction

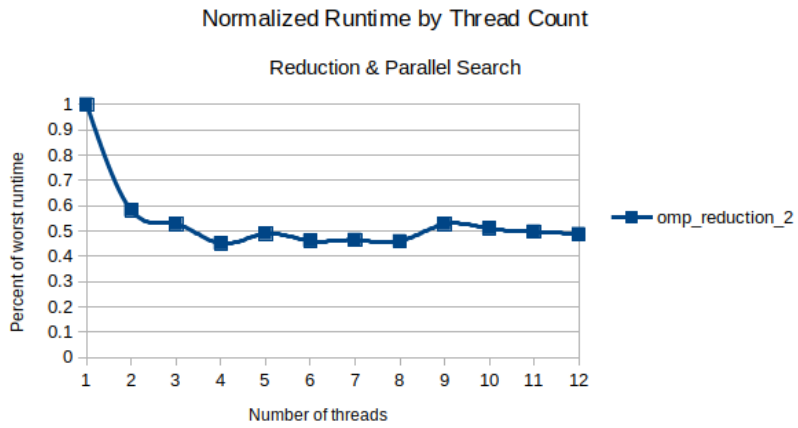

Figure 8: Reduction and parallelized search

Implementing Foster's Methodology with omp_reduction_3

We see Foster's methodology of agglomeration and mapping more clearly in this snippet, with a `for` loop for each process, and a distributed agglomeration of house records for each process. The mapping takes advantage of the binomial tree reduction within a single process as well as during inter-process communication. This also uses the `for` loop improvement for the search afterward.

```
omp_set_num_threads(p);

#pragma omp parallel for private(i, j, dist) \

reduction(min:internal_closest_dist) \

reduction(min:closest_dist)

for (i = 0; i < p; i++) {

  for (j = i * n / p; j < (i + 1) * n / p; j++) {
```

```
    dist = get_dist_sq(houses[j], railroad);
    houses[j][2] = dist;

    if (internal_closest_dist > dist) //single process reduction
      internal_closest_dist = dist;
  }

  if (closest_dist > internal_closest_dist) //map reduction across multiple processes
    closest_dist = internal_closest_dist;
}

#pragma omp parallel for private(i)
for (i = 0; i < n; i++) {
  if (houses[i][2] == closest_dist)
    closest_i = i;
}

return houses[closest_i];
```

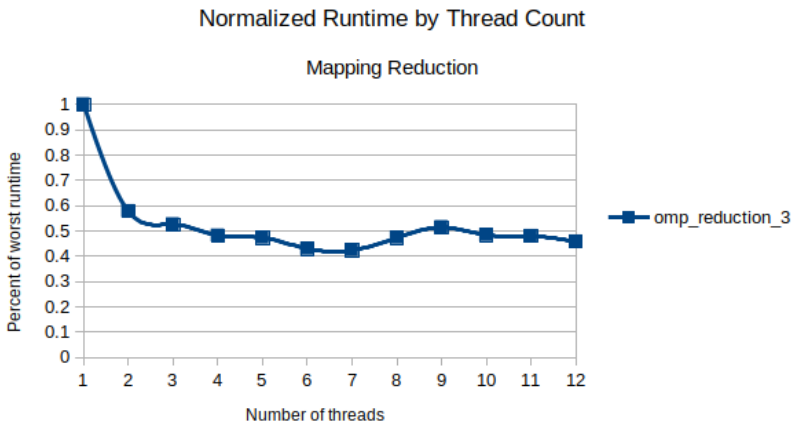

Figure 9: Agglomeration and mapping reductions

Quirky Foster's with omp_critical_4

Finally, we can put all the improvements together. The code below uses the agglomeration from **omp_reduction_3**, but there is no longer a need for a final searching `for` loop because of the use of the quirky `for` loop to handle inter-process communication mapping. Also, now that there are at most p instances where this quirky critical section is accessed (and due to uniform randomness of the house samples, on average around $\frac{p}{2}$ critical section entries), this critical section has a smaller time footprint than the search `for` loop used in the reduction code snippets.

```
omp_set_num_threads(p);
#pragma omp parallel for private(i, j, dist, internal_closes_dist, internal_closest)
for (i = 0; i < p; i++) {
  internal_closest_dist = __DBL_MAX__;
  for (j = i * n / p; j < (i + 1) * n / p; j++) {
    dist = get_dist_sq(houses[j], railroad);
    houses[j][2] = dist;

    if (internal_closest_dist > dist) { //no need for sync, all in one process
      internal_closest_dist = dist;
      internal_closest = houses[j];
    }
  }

  if (closest_dist > internal_closest_dist) {
    #pragma omp critical
    {
      if (closest_dist > internal_closest_dist) {
        closest_dist = internal_closest_dist;
        closest = internal_closest;
      }
    }
  }
}

return closest;
```

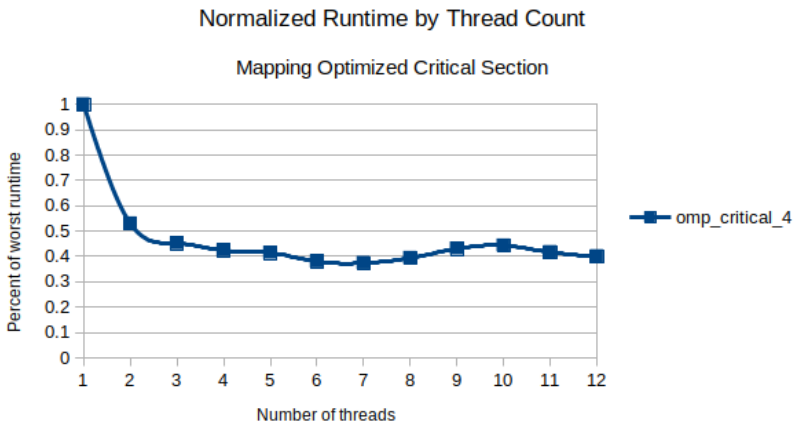

Figure 10: Agglomeration and mapping with optimized critical section

Benchmarking

We turn our attention now to benchmark results for different implementations of the parallelization described above. For the compared results in figure 11, we see each of the benchmark series plots come from a different implementation. Notice first that best run times, provided by **omp_critical_2**, come from the incorrect critical section code. Unfortunately, the quirky attempt to correct the error in **omp_critical_3** provided sub-par results. The more explicitly defined implementations of the agglomeration and mapping described above, **omp_reduction_3** and **omp_critical_4**, showed two of the best performances, with the quirky critical section showing the best overall times (assuming we ignore the incorrect implementation). Though **omp_critical_4** did not utilize binomial tree mapping, the results were still favorable in comparison to the all other correct implementations.

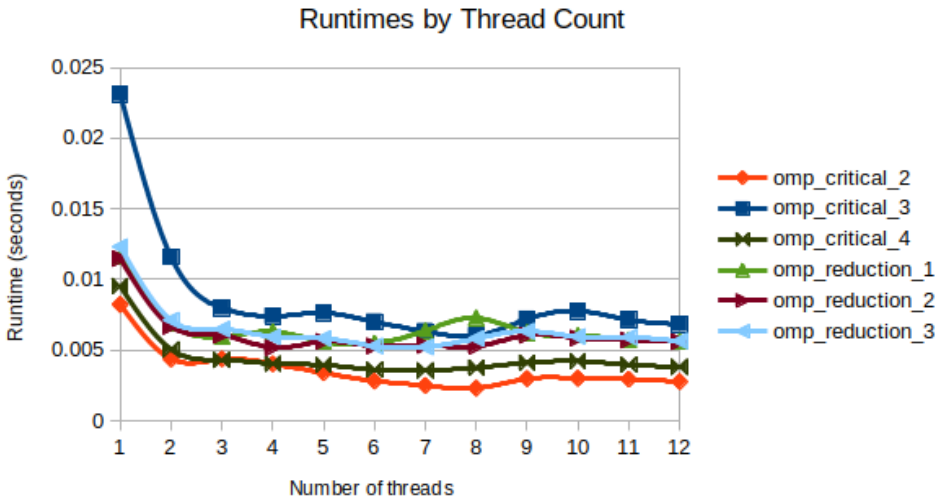

Figure 11: Comparing implementations

Theoretical Efficiency

Aside from empirical evidence, we can analyze the theoretical, asymptotic run times of the sequential and parallel algorithms. Rather than assessing each implementation we will only compare the sequential to the top performer of the parallelized implementations.

Sequential Assessment

The sequential solution need only focus on the computational time, since there is no inter-process communication. Though there might be some clever improvement to this algorithm that does not require Euclidean distances to be calculated, here we will assume that a prerequisite for finding the closest house is finding the distance from each of the n houses to the railroad station. The run time of calculating the Euclidean distances is $\Theta(n)$. While finding these distances, we can compare each new distance to the current minimum distance and, if the new distance is shorter, assign a new minimum distance and closest house. Because the discovery of the closest house can happen simultaneously with the distance calculation, the overall run time is $\Theta(n)$ for this algorithm.

Parallel Assessment

For a proper parallel algorithm assessment, we will determine the computation and communication run times. Let us look at the **omp_critical_4** implementation of the parallel algorithm. During the computation step, each process calculates the distance for all houses it is responsible for, either $\lfloor \frac{n}{p} \rfloor$ or $\lceil \frac{n}{p} \rceil$ houses. During these calculations, this algorithm finds the minimum distance and closest house in the same way that the sequential algorithm does, but for roughly $\frac{n}{p}$ houses, so the computational run time for each process is $\Theta(\frac{n}{p})$. During communication, the distance of each of the p closest houses found by each thread will be compared to the global closest distance, and will enter the critical section if the individual thread has a house closer to the railroad station than the current global closest house. In the worst case, each thread enters the critical section, making this a sequential chain of updates to the global closest house variable. This sequential process would take p iterations of the critical section of code. Since the critical section of code runs in $\Theta(1)$, we can say that the worst time for this communication is $O(p)$. Thus the overall run time of this algorithm is $\Theta(\frac{n}{p}) + O(p)$. This implies that an increase in processes (threads) will reduce computation time but increase communication time, as expected.

Discussion

A few talking points remain about the individual algorithm empirical results and how each implementation responded to an increase in parallelism. But first, let us review the theoretical efficiency. Note that as n increases this algorithm tends toward $\Theta(n)$, while an increase in p results in a run time tending toward $O(p)$. This gives overall a linear run time. This implies that though an increase in parallelism might provide a speedup of some constant factor, the parallel algorithm is in the same efficiency class as the sequential algorithm. With this perspective, we turn our attention to figures 5 through 10. Notice that the incorrect critical section made significant gains, achieving nearly a 4x speedup. In fact, a similar speedup was achieved by the quirky critical section implementation as well, as shown in figure 6. While most of the reductions only gained approximately 2x speedups, the proper implementation of Foster's algorithm from figure 9 made the most gains, with 6 or 7 threads running in 40% of the time of the single thread execution. The best correct algorithm shown in figure 10 achieved nearly a 3x speedup. Also, most algorithms peaked in performance at 8 threads. As you might have guessed, the implementations were run on a 4 core computer. While gains can be made from the basic omp parallel for loop, the best gains come from a more structured, balanced assignment of work to the available threads.