CM30225 Parallel Computing
Distributed Memory

Rowan Walshe

January 8, 2018

Chapter 1

Code

1.1 Sequential Version

For my sequential version of my program I used one of the programs that I created for the first part of this coursework. The main loop for this program can be seen below in Code listing 1.1. The program moves through the 2D array from the top left down to the bottom right. At each point it the array it temporally stores the value at the point, then calculates the new value based on the average of the surrounding four values. It then checks to see if the absolute difference between the new and old value is greater than the given precision. If the value at any point changes more than the given precision, then at the end of this iteration of averaging values, it must complete at least one more iteration. By checking if the endFlag is still true before calculating the absolute difference, it reduces the number of times this difference needs to be calculated, reducing overall runtime. This is illustrated in Code Snippet 1.1.

Code Snippet 1.1: Version 1 Sequential Main Loop

```
while(1) {
    endFlag = true;
    iterations++;
    for(i=1; i<sizeOfPlane-1; i++) {
        for(j=1; j<sizeOfPlane-1; j++) {
            pVal = plane[i][j];
            plane[i][j] = (plane[i-1][j] + plane[i+1][j]
                + plane[i][j-1] + plane[i][j+1])/4;
            if(endFlag && tolerance < fabs(plane[i][j]-pVal))
                endFlag = false;
        }
    }
    if(endFlag)
        return iterations;
}
```

While working on the shared memory version of this coursework, I found that by accessing data in the array using a checkerboard pattern, I was able to guarantee the result of the program would always be the same, while also reduce runtime due to a lower cache miss rate. However for a distributed memory program, accessing data in this pattern had little to no effect on the run time, as any benefit was removed due to the increased overhead required for the extra MPI calls. Therefore, I decided to use this version instead of the checkerboard one for all my testing.

1.2 MPI Version

1.2.1 Relexation Algorithm

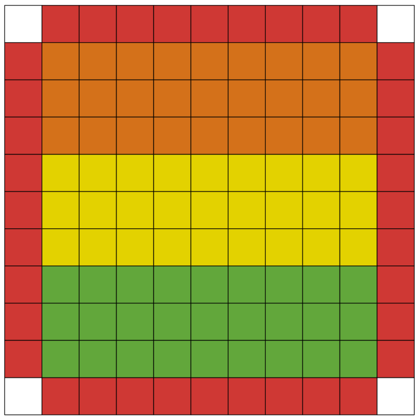
Figure 1.1: Row Division

I am using a similar method to the one I used in the shared memory coursework, in order to split the problem between MPI processes. Each MPI process is assigned a certain number of rows to work on, based on its world rank. In the example shown in Figure 1.1, an 11x11 array is split up between three MPI processes, with each process being assigned three rows. Each process frees memory for an array that contains two more rows than it is going to run the relaxation algorithm on. This is because each MPI process needs access to the values of the row above and below the first and last row that it will be doing work on. On top of this, by reducing the amount of data that each process has to keep in memory, as well as the amount of memory needed per node, it is easier to scale to run larger problem sizes as you are not limited by the amount of memory that you could reasonably fit in a shared memory system. The code for the computation that each MPI process does on its part of the array can be seen bellow in Code Snippet 1.2. This code should be recognisable from the sequential program, with the only difference being the number of rows that work is done on.

Code Snippet 1.2: MPI Relaxation Computation

```
for(i=1; i<recBot; i++) {
    for(j=1; j<sizeOfPlane-1; j++) {
        pVal = plane[i][j];
        plane[i][j] = (plane[i-1][j] + plane[i+1][j] + plane[i][j
           -1] + plane[i][j+1])/4;
        if(endFlag && tolerance < fabs(plane[i][j]-pVal))
            endFlag = false;
    }
}
```

1.2.2 MPI Communication

After each iteration, every MPI process sends the top and bottom row that it worked on to the corresponding MPI processes, and then receives updated data into the first and last row of its 2D array. The one exception to this is the MPI process with world rank zero, which only sends and receives data to and from the bottom of its array, and the last MPI process which only sends and receives data to and from the top of its array.

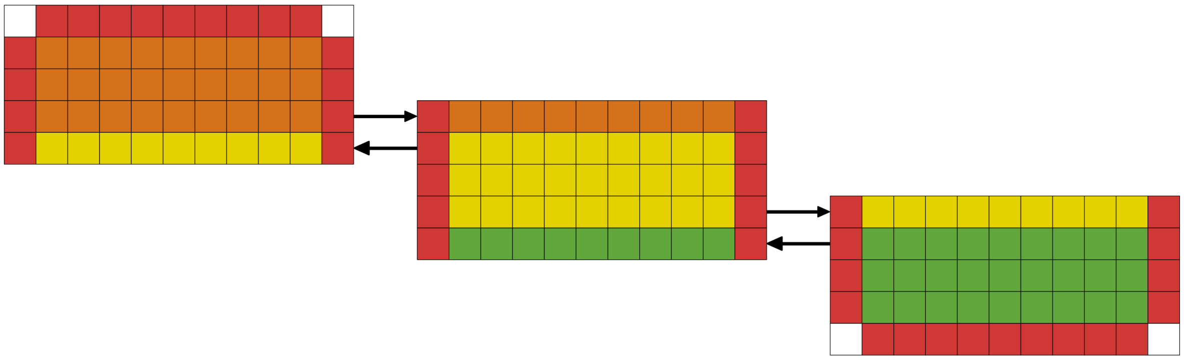

Figure 1.2: MPI Communication Pattern

Code Snippet 1.3: MPI Communication

```
// Send and Recieve data depending on the world_rank
if(world_rank==0) {
    // Only send data down to process with world_rank 1
    MPI_Isend(&plane[sendBot][1], sizeOfInner, MPI_DOUBLE, 1, 0,
       MPI_COMM_WORLD, &myRequest1);
    MPI_Recv(&plane[recBot][1], sizeOfInner, MPI_DOUBLE, 1, 0,
       MPI_COMM_WORLD, MPI_STATUS_IGNORE);
} else if(world_rank==world_size-1) {
    // Only send and recive/data to the process above i.e.
       world_rank-1
    MPI_Isend(&plane[1][1], sizeOfInner, MPI_DOUBLE, world_rank-1,
        0, MPI_COMM_WORLD, &myRequest1);
    MPI_Recv(&plane[0][1], sizeOfInner, MPI_DOUBLE, world_rank-1,
       0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
} else {
    // Send new data up
    MPI_Isend(&plane[1][1], sizeOfInner, MPI_DOUBLE, world_rank-1,
        0, MPI_COMM_WORLD, &myRequest1);
    // Send new data down
    MPI_Isend(&plane[sendBot][1], sizeOfInner, MPI_DOUBLE,
       world_rank+1, 0, MPI_COMM_WORLD, &myRequest2);
    // Receive new data from above
    MPI_Recv(&plane[0][1], sizeOfInner, MPI_DOUBLE, world_rank-1,
       0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    // Receive new data from below
    MPI_Recv(&plane[recBot][1], sizeOfInner, MPI_DOUBLE,
       world_rank+1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

I'm using `MPI_Isend()` instead of `MPI_Send()` as I hit a buffer size limit on balena when attempting to send rows over 2002 long, causing the program to hang. This is a non blocking send which means that each MPI process moves on to attempting to receive data, no matter weather or not it has actually finished sending data yet. This could cause issues, however as I use the blocking version of receive, followed by an MPI call to bring together the endFlag from each MPI process, each process never attempts to start a new iteration until all new data has finished being sent and received. To bring together the endFlag from each process I use another MPI function, `MPI_Allreduce()`, which can be seen below in Code Snippet 1.4.

Code Snippet 1.4: MPI Communication

```
MPI_Allreduce(MPI_IN_PLACE, &endFlag, 1, MPI_INT, MPI_LAND,
    MPI_COMM_WORLD);
```

The MPI_op MPI_LAND makes the Allreduce perform a logical AND1 between the endFlag of all of the MPI processes. Therefore if any one of the endFlags is false, then all of them become false, and do another iteration.

1.2.3 Result Output

Initially when the program had finished, in order to output the full plane, I was initially gathering all the data on to the MPI process with world rank 0, from which it would become trivial to either print the result to stdout, or write it to a file. In order to call an MPI_Gatherv function, the root node had to calculate how many bytes it would be receiving from each MPI process, as well as the offset from the start of the receiving buffer it was to write the data to. The code for this can bee seen below in Code Snippet 1.5

Code Snippet 1.5: How to Gather Data onto One MPI Process

```
int sizeOfInner = sizeOfPlane-2;
int rowsPerThreadS = sizeOfInner/world_size+1;
int rowsPerThreadE = sizeOfInner/world_size;
int remainingRows = sizeOfInner - world_size * rowsPerThreadE;

int tempNumRows;
int* recvcounts = malloc((unsigned int)world_size * sizeof(int));
int* displs = malloc((unsigned int)world_size * sizeof(int));

// Calculate recvcounts and displs
for(int i=0; i<world_size; i++) {

    if(i < remainingRows) {
        tempNumRows = rowsPerThreadS;
    } else {
        tempNumRows = rowsPerThreadE;
    }

    recvcounts[i] = tempNumRows * sizeOfPlane;
    if(i < remainingRows) {
        displs[i] = (i * rowsPerThreadS) * sizeOfPlane;
    } else {
        displs[i] = (i * rowsPerThreadE + remainingRows) *
            sizeOfPlane;
    }
}

MPI_Gatherv(&subPlane[1][0], (numRows-2)*sizeOfPlane, MPI_DOUBLE,
    &plane[1][0], recvcounts, displs, MPI_DOUBLE, 0, MPI_COMM_WORLD
    );
```

However in my final version I wrote some code in order to be able to output the results to a file without having to gather all of the data in to a single process. While my implementation, which can be seen below in Code snippet 1.6, uses the standard C library for file input output, I also could have used a number of MPI file writing functions such as `MPI_File_write_at()` in order to have all of the MPI processes to write to the file at the same time, without having to use a barrier.

Code Snippet 1.6: Write Results to File

```
FILE* file;
char* file_name;
// Create the file name, so it is easy to see at a glance the
    world_size and problem size that the results came from
asprintf(&file_name, "%d-%d.result", world_size, sizeOfPlane);

for(int i=0; i<world_size; i++) {
    // If it is this processes turn, write out to the file
    if (i == world_rank) {
        // If this is world_rank 0 then create a new file,
            otherwise append to the existing file
        file = fopen(file_name, world_rank == 0 ? "w" : "a");

        // Unless this is the first or last MPI process, do not
            write out the first or last line of the array
        int startingRow = 1;
        int endingRow = numRows - 1;
        // If this is the first or last MPI process, then also
            write out the first or last line of the array
            respecively
        if(world_rank == 0) {
            startingRow = 0;
        } else if(world_rank == world_size-1) {
            endingRow = numRows;
        }
        // Write out data from the array
        for(int j=startingRow; j<endingRow; j++) {
            for(int k=0; k<sizeOfPlane; k++)
                fprintf(file, "%f, ", subPlane[j][k]);
            fprintf(file, "\n");
        }
        fclose(file);
    }
    MPI_Barrier(MPI_COMM_WORLD);
}
// Additional information about how the program ran
if(!world_rank) {
    file = fopen(file_name, "a");
    fprintf(file, "\nThreads: %d\n",world_size);
    fprintf(file, "Size of Pane: %d\n", sizeOfPlane);
    fprintf(file, "Iterations: %lu\n", iterations);
    fprintf(file, "Time: %Lfs\n", toSeconds(start, end));
    fclose(file);
}
```

Chapter 2

Correctness Testing

I was able to confirm that for small test sizes my final program was correct by manually calculating the result by hand and comparing my result with that of my program. For larger problem sizes, I wrote the final result of the program to a file, as well as the result one iteration before, to another file. I was then able to use a short Python script to compare the two files, to make sure that there was never a difference greater than 0.0001 between respective values in the two files.

Another test I performed was setting all of the edges to one, and then running the program with a precision of zero and outputting the result to a file. Looking at the output file, all of the values in the array should be equal to one. I ran both these tests a number of times with various thread counts and problem sizes, with the result coming out as correct each time.

Chapter 3

Scalability Testing

To measure the performance of my parallel algorithms, I gathered a large range of results from balena, using various thread counts and problem sizes. I will be using a number of common metrics including speedup, efficiency, and the Karp-Flatt Metric to discuss the performance and scalability of my MPI program.

Speedup on P processors is equal to the time to run the sequential algorithm divided by the time to run the parallel algorithm on p processors. This assumes that the problem size is constant.

$$Speedup\ _p = \frac{Sequential\ Time}{Parrallel\ Time\ _p}$$

Efficiency is equal to the amount of speedup per processor. It is a measure of how efficiently my parallel algorithm is using the extra processors.

$$Efficiency\ _p = \frac{S_p}{p} = \frac{Sequential\ Time}{p \times Parrallel\ Time\ _p}$$

The Karp-Flatt metric can be used to effectively measure the sequential portion of parallel programs. Generally it is between zero and one, though may be more than one if there is slowdown, or less than zero if there is superlinear speedup.

$$e = \frac{\frac{1}{S_p} - \frac{1}{p}}{1 - \frac{1}{p}}$$

Using the value at which the Karp-Flatt metric levels off at, in conjunction with Amdahl's law, it is possible estimate the maximum speedup of a given program, or in this case problem size.

$$Max\ Speedup = \frac{1}{KarpFlatt}$$

Overhead is required to calculate isoefficiency. It effectively tells us how much extra time was added in overheads while parallelising the sequential algorithm. A smaller overhead also means that the parallel algorithm is work efficient.

$$T_o = pT_p - T_s$$

Isoefficiency allows me to measure how scalable my parallel algorithm is. It is generally between zero and one. The closer isoefficiency is to one, the better it scales.

$$E = \frac{1}{1 + \frac{T_o}{T_s}}$$

3.1 Time to Run

To test the scalability of my different algorithms, I measure the time that it took to complete the start up and relaxation part of the program, for various problem sizes and MPI process counts using a precision of 0.0001. Instead of using the time given in the output file from balena, I decided to use `clock_gettime()` which can be found in the standard C time library. This allowed me to measure the runtime more precisely, to within 0.000001 seconds. It also allowed me to remove overheads from the timing that could be introduced while the OS starts and cleans up after the program. In practice these overheads become less significant as the runtime increases, as it is something that only has to be done once per run.

As you can see in Figure 3.1, there is generally a large drop of in runtime as the number of processes starts to increase, which then begins to level off. As the largest problem size that finishes for process count one 2048, this is the max problem that I will being using in my analysis going forward.

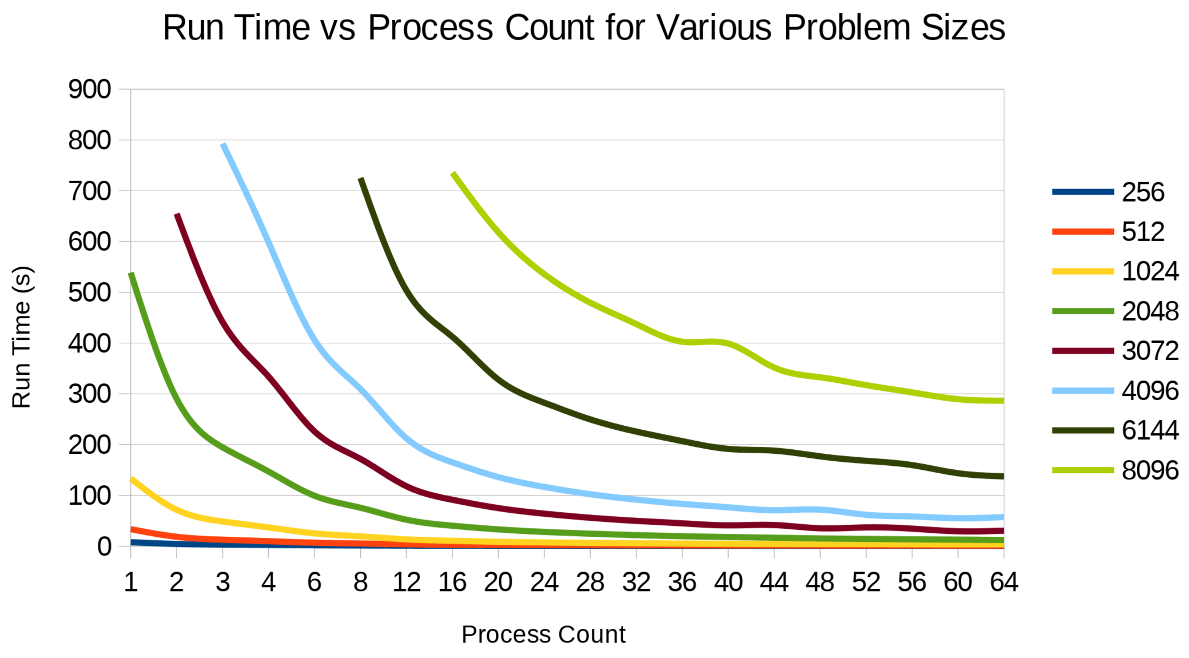

Figure 3.1: Program Runtime vs Process Count for Various Problem Sizes

3.2 Speedup

Using the timing results that I gathered, I was able to calculate speedup for various problem sizes and process counts. As you can see in Figure 3.2 the parallel algorithm starts off slower than than the sequential for all process counts, however by array size 64x64, all of them have speedup greater than one and so are finishing faster than the sequential algorithm. The parallel program is intially slower than sequential program, as overheads introduced in order to parallelise the program, add more time than is saved by running in parallel.

Unfortunately as there is a time limit for how long a program can run on balena for, we are unable to see at what problem size, speedup levels off at for some of the larger process counts, as I am unable to test the sequetial program at larger problem sizes. However as stated by Gustafson's Law, as the problem sizes get increased, the sequential part effectively becomes smaller. Therefore I expect that if I was to be able to test larger problem sizes, the larger thread counts would continue to see their speedup increase before eventually levelling off.

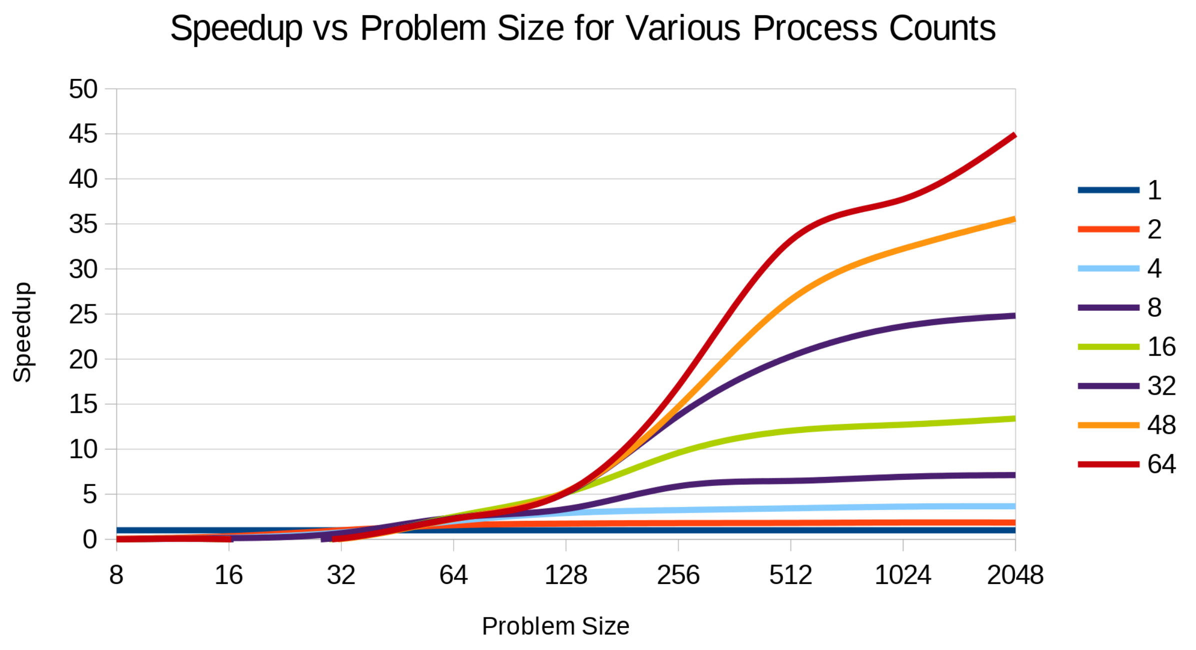

Figure 3.2: Speedup vs Problem Size for Various Process Counts

In Figure 3.3, you can see that the speedup for some of the smaller problem sizes begins to level off, even as the process count continues to increase. This is because of the Amdahl's law, which says that:

> ”Every program has a natural limit on the maximum speedup it can attain, regardless of the number of processors used”

Unfortunately as there is a node limit on balena, we are unable to see exactly where the speedup's level off at for some of the larger problem sizes.

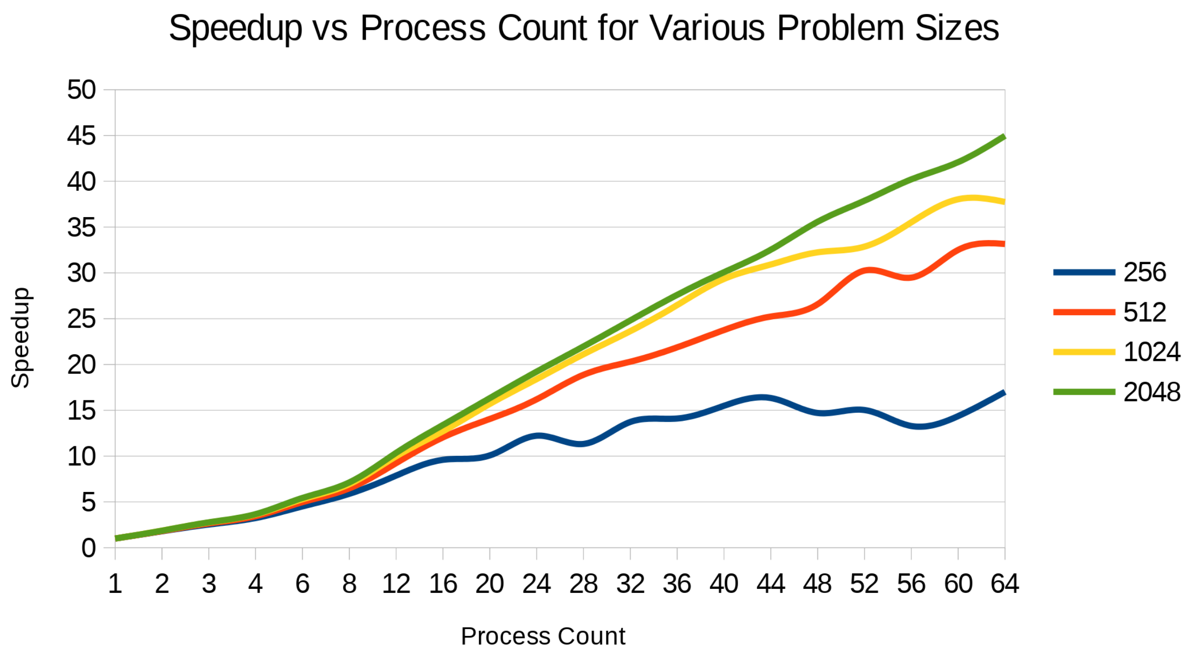

Figure 3.3: Speedup vs Process Count for Various Problem Sizes

3.3 Efficiency

As stated earlier, efficiency is a measure of how efficiently my parallel algorithm is using the extra processors. For example, if a program has an efficiency of 0.5 for a given problem size an number of processes, then half of the time that each process could be doing work, is lost to overheads. The graph in Figure 3.4 shows that my algorithm scales pretty well, when the processor count increases, efficiency decreases slower the larger the problem size is. Another way thinking about this, is that efficiency increases as the problem size does, for a given process count. This follows the reasoning of Gustafson's law. As none of the lines on the graph go above one, it is easy to see that none of the programs are displaying superlinear behaviour.

3.4 Karp-Flatt Metric

Figure 3.5 shows how the Karp-Flatt metric changes as the process count increase for various problem sizes. As the process count increases, the Karp-Flatt metric decreases, until it eventually levelling off for each respective problem size. In the graph, you can see that the larger the problem size, the smaller the Karp-Flatt metric is. This is consistent with Gustafson's law, which says that as the problem size increases, the effective sequential part of an algorithm decreases.

3.5 Final Note About Graphs

One final note about the graphs is that for some of the smaller problem sizes, the line is generally not as smooth. This is because the run times are very small (less than a second), which then causes small runtime variances to create large variances on the graphs.

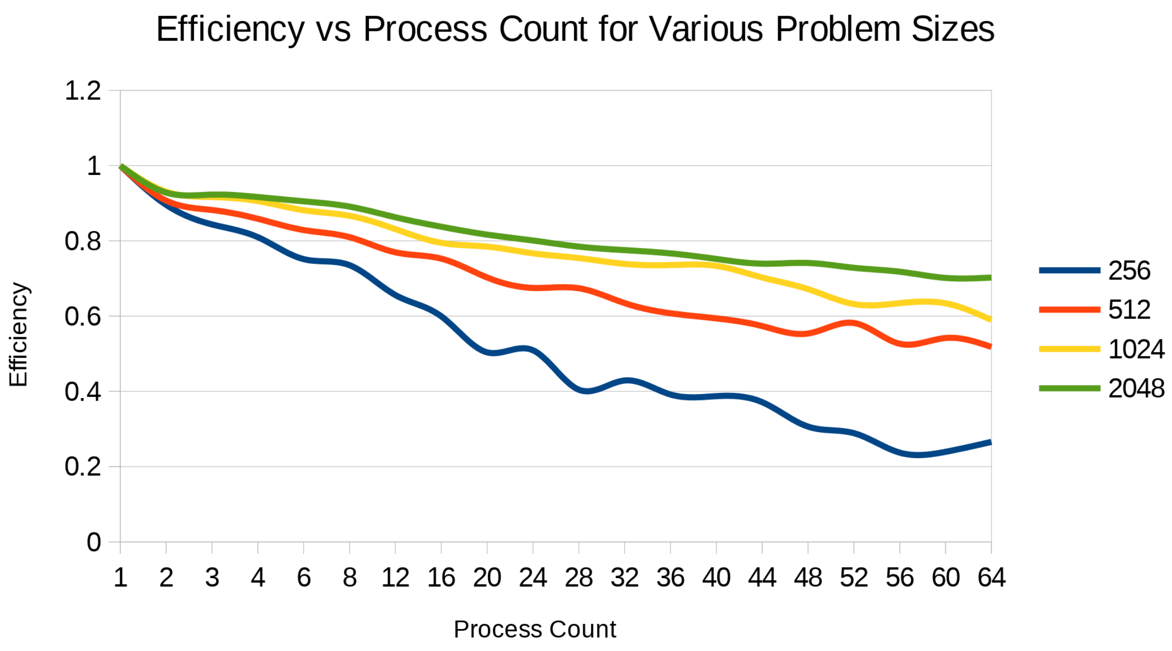

Figure 3.4: Efficiency vs Process Count for Various Problem Sizes

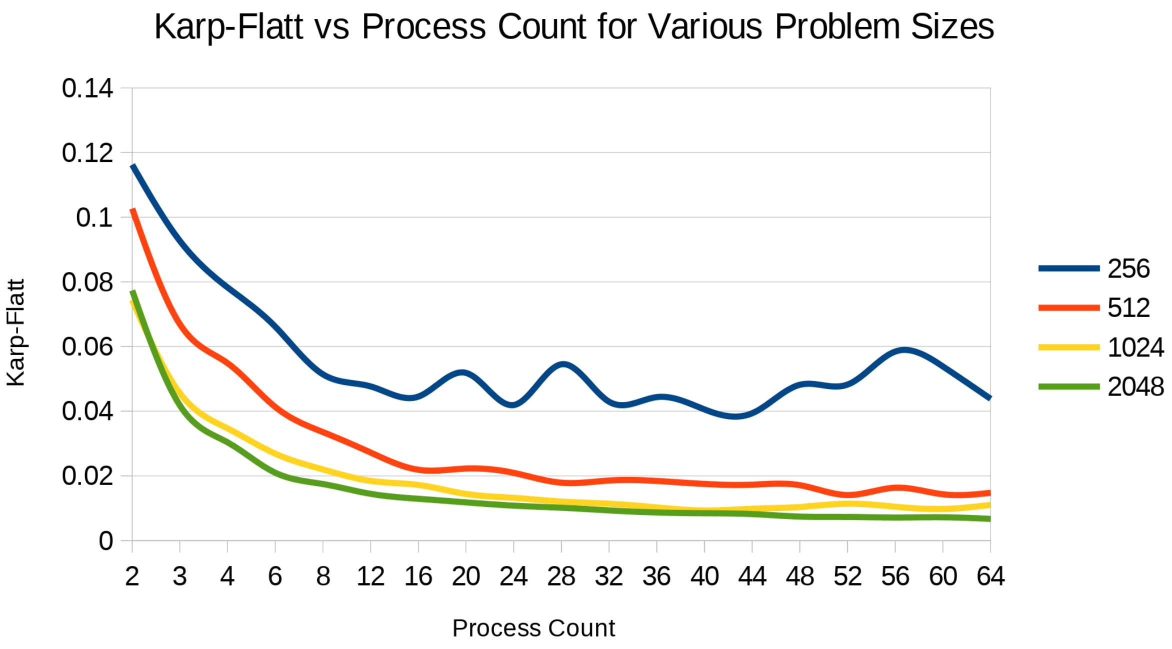

Figure 3.5: Karp-Flatt vs Process Count for Various Problem Sizes

Chapter 4

Conclusion

My current solution scales well with process count and problem size, with the max recorded speedup being around 45. My results also show that the efficiency of my solution drops as expected, as the thread count increases. The lowest Karp-Flatt value that I messured was around 0.067. Using this value in conjunction with Amdahl's law, I can predict a max speedup of my algorithm on an array size 2048 will be around 149.