COSC 3P93 - Project Step 3 Report - 6292064, 6366298

Dhairya Jaiswal, Nilaanjann Misra, Brock University, Canada

December 14, 2022

1 Abstract

In the last step, we created a sequential implementation of the Simplex method, a subtopic of linear programming. While there were several errors, bugs, and issues that took place when coding the method, a successful implementation was created that allowed the user to input the necessary details such as the coefficients of the objective function, the coefficients of the constraints. The algorithm in place would then compute the optimal combination of values for each variable that allows for maximum output. Moving forward, in step 3, we were tasked with using that sequential implementation and turning it into a parallel implementation using OpenMP and OpenMPI. These API specifications are primarily used for parallel programming and drastically decrease the level of work that would be needed to be done to parallelize a program from scratch. While parallelization does have its own obstacles and does raise the overall complexity of the program, the execution times on different threads/processes is the primary measure of whether or not parallelization has been implemented and conducted sufficiently.

Simplex Method for Maximization Pseudocode

Ensure:
$rows = 3;\ columns = 6; variableCount = 2;\ constraints = 2;\ temp = 0;$

$variableCount \leftarrow UserInput$
$constraints \leftarrow UserInput$

Ensure:
$resultColumn[variableCount];$
$resultRow[constraints];$
$mainMatrix[rows][columns];$
$isNegative = true;$

`Populate` $mainMatrix$

while $isNegative = true$ **do**
```
isNegative = false;
pivotElement;
pivotRow;
maxNegative = 0;
maxNegativeColumn;
```
for $i < columns$ **do**
`Find` $maxNegativeColumn$
`Find` $pivotColumn$
end for
if `There exists an unbounded solution` **then**
`End program`
end if
if $maxNegative < 0$ **then**
for $i < rows - 1$ **do**
`Find` $pivotElement$
`Find` $pivotRow$
end for
for $i < variableCount$ **do**
if $maxNegativeColumn = i$ **then**
`Populate` $resultColumn$
`Populate` $resultRow$
`Increment` $temp$ `by 1`
end if
end for
end if
for $i < columns$ **do**
`Divide entire pivot row by pivot element`
end for
for $i < rows$ **do**
`Flip sign of row element in max negative column`
for $j < columns$ **do**
if $i = pivotRow$ **then**
`break`
end if
$mainMatrix$ `is modified as per Simplex`
end for
end for
for $j < columns$ **do**
`Check if last row is non-negative`
`Adjust` $isNegative$ `accordingly`
end for
end while

2 Design Reasoning

2.1 Parallelization Choices

The algorithm above is the algorithm that was used to drive the sequential version of the Simplex method program i.e. the one done in Step 2. Now, in terms of parallelization, we can see that there are several for loops in the code. This is a prime area of access and modification for OpenMP since it has a *pragma* directive that is primarily meant for for loops. It parallelizes the loops by having each running thread deal with certain iterations. When it comes to the simplex method and areas that can be parallelized, we can break it down over several sections of the code:

1. Now one of the major changes that were made was changing the matrix to a vector from a 2D array. The primary reason for this will be discussed later. But the section of code dealing with populating the matrix can be done in a parallel manner. It's a simple nested loop implementation that can be easily parallelized using OpenMP.

2. The next area of parallelization that can be done is the section of code dealing with finding the $maxNegativeColumn$. This area of code also runs on for loops in order to figure out which column the most negative element resides in as that essentially translates into the column that our *pivotElement* resides in. However, it is imperative that variables are dealt with properly here as we don't want unnecessary modifications or accesses to the same variable by different threads causing a major error and negatively impacting the parallel behavior of the program.

3. The next major parallelization occurs in section of code dealing with figuring out *pivotRow* and *pivotElement*. These are crucial to solving problems using the Simplex method and have to undergo careful thought when parallelizing. These are responsible for later calculations that change the entire matrix upside down and modify it which means that the variables must not undergo situations where unnecessary modifications take place.

4. The section of code where the entire row where the *pivotElement* resides is divided by the *pivotElement* itself is also handled in parallel. It's a relatively straightforward for loop thus not requiring too much attention to detail or code adjustments when compared to the sequential version.

5. The last section of parallelization that occurs is in the section dealing with modifying the whole matrix based on the Simplex method, which entails adjusting every row apart from the *pivotRow* to reach a certain state in hopes of the bottom, resultant row being non-negative.

Note: The for loop section for the $resultColumn$ and $resultRow$ have not been parallelized for reasons that will be discussed in the next section.

2.2 Design and Implementation Choices OpenMP

The primary concern when dealing with parallelizing programs in C++ is making sure that there aren't any segmentation faults occurring i.e. memory leaks, and threads are accessing shared variables in the correct, proper manner so as to not cause errors in the run or execution of the program.

1. First, we changed our 2D array setup for our matrix to a vector in order to ensure that there aren't any cases of issues regarding how much the matrix can actually hold.

2. The second thing to tackle in this program was populating the array with zeroes and then appropriately placing the slack variables in parallel (given the number of variables and constraints provided by user input). We did this using the same loops as the sequential version, except this time we employed *private* variables (alongside the loop directive in OpenMP) in order to ensure that each thread has its own instance of a variable. This was primarily done after discovering errors being caused during runs.

```
int i = 0
# pragma omp parallel for num_threads(thread_count) private(i)
for i < rows do
    for j < columns do
        Set mainMatrix[i][j] to 0.0
    end for
end for
```

```
int j = variableCount
# pragma omp parallel for num_threads(thread_count) private(i)
for i < rows do
    if j < columns − 1 then
        Set mainmatrix[i][j] to 1.0
        # pragma omp critical
        j++
    end if
end for
```

3. Our third concern was the overall overhead that running the program would cause. Starting up threads, executing threads, syncing threads et cetera all promote the overall overhead of the program thus affecting execution time immensely. And this would occur primarily considering the fact that our application worked via user input and would, realistically, deal with simple programs. Thus, we decided to ditch the idea of the user inputting every single value manually and instead populated our matrix with randomized *float* values based on the given constraints by the user i.e. the number of variables and the number of constraints thus being able to deal with large enough problems without having to enter every value manually.

```
for i < variableCount do
    Generate random float given a range (code has the range 2 - 800)
    Insert into mainMatrix[rows − 1][i]
    Flip sign of mainMatrix[rows − 1][i]
end for
```

```
for i < rows − 1 do
    for j < variableCount do
        Generate random float given a range (code has the range 2 - 800)
        Insert into mainMatrix[i][j]]
    end for
    Generate random float given a range (code has the range 2 - 800)
    Insert into mainMatrix[i][j]]
end for
```

4. In regard to the loop concerning finding the *maxNegativeColumn*, we ensured that local variables were created within the loop so that each thread has it's own copy of the variable to modify and access.

```
# pragma omp parallel for num_threads(thread_count)
for i < columns do
    Initialize variables local_maxNegative and local_maxNegativeColumn
    if mainMatrix[rows − 1][i] < local_maxNegative then
        Change local_maxNegative and local_maxNegativeColumn
    end if
    # pragma omp critical
    if local_maxNegative < maxNegative then
        Set maxNegative and maxNegativeColumn
    end if
```

end for

The second if statement is a critical section as it is updating the global variable and is handled appropriately via an OpenMP directive whose purpose is ensuring that a certain section of the code can only be accessed by one thread at a time.

5. In the same manner that the last loop section was handled i.e. by creating local variables for each thread that then later came together to be compared to adjust the global variable, the loop section regarding evaluating and figuring out the *pivotRow* and *pivotElement* is done in the same way. Furthermore, this loop also inhabits a critical section i.e. when the global variable is updated and is dealt with using an OpenMP directive.

6. The loop involving the adjustments made solely to the *pivotRow* was also parallelized in a relatively straightforward manner.

`# pragma omp parallel for num_threads(thread_count)`

for $j < columns$ **do**

`Set` $mainMatrix[pivotRow][i]$ `=` $mainMatrix[pivotRow][i]$ `/` $pivotElement$

end for

7. The final calculation step before checking if there are any more negatives left in the last row is dealt with in a similar manner as the last point and is kept relatively the same when compared to the sequential code (no parallelization done).

8. Now, we decided to ditch the whole process involving figuring out the result of the variables i.e. *resultRow* and *resultColumn* primarily because of the fact that the problems can get enormous when trying to see significant change in parallel execution times. As such, the only thing that is printed out at the end is the Z value i.e. the value associated with the final output. And based on the way the code is setup, the final output's value is an easy method to check if the parallelization of the program runs uniformly and as expected.

2.3 Design and Implementation Choices OpenMPI

As this is essentially a reskin of the OpenMP program, we felt that a higher level description of the design details was more appropriate.
Our primary concern for designing the OpenMPI algorithm was to ensure that it entailed a significant performance increase in contrast to the sequential algorithm while maintaining a relative familiarity to the OpenMP implementation, as that implementation was created first. This was done in order to ensure consistent code was sustained and to avoid having to redesign an algorithm from scratch. In doing so, many design decisions from the OpenMP implementation still apply here.

1. The main issue we ran into as soon as we began programming, was that OpenMPI does not allow us to send vectors like OpenMP does. As it only allows contiguous memory to be broadcasted, we were forced to regress to a standard 2D array, which entailed a handful of issues we faced during the OpenMP implementation. After running a thorough analysis on the issue at hand, we found that equalizing the main matrix to allocated contiguous memory using a pointer to a pointer allowed the program to access the matrix elements using only a single pointer and pre-determine memory requirements. This not only fixed our segmentation fault issue but also improved performance to an extent throughout the program.

2. As with the OpenMP implementation, our second concern was how to handle user input in an efficient way. We were able to realize a more efficient solution than the obvious choice of using MPI_Send for each input. We were able to do this by having one process (of rank 0) take the input and broadcast it to all other processes using MPI_Bcast. As mentioned before, this was a consistent theme throughout the implementation and it greatly increased performance.

3. We decided to take a data-parallelism based approach when building this program. This was achieved using a block-partitioning technique. The problem size is broken down into chunks of size N, where N is the number of MPI processes allocated to execution. The columns of the matrix are divided among the processes for parallel processing. This allows each process to operate on a subset of the columns, reducing the amount of data each process needs to handle and improving performance.The variable count is divided among the processes for parallel processing. This allows each process to operate on a subset of the variables, reducing the amount of data each process needs to handle and improving performance. The rows of the matrix are also divided among the processes for parallel processing. This allows each process to operate on a subset of the rows, reducing the amount of data each process needs to handle and improving performance.

4. $MPI_Allreduce$ was used multiple times throughout the program as opposed to Reduce since we decided to use a data-parallelism based approach. This allowed us to warrant operations on local variables with synchronized data.

5. We used MPI_Gather after every pivot row operation to update the processes' main matrix

6. Since each process is handling different portions of data, we had to make sure data consistency was preserved. At the end of each iteration of the main while loop, $MPI_Barrier$ was used to allow for each process to finish it's current iteration before starting a new one.

3 Analysis

In terms of analysis, as per the earlier mentioned point, we realized that overcoming the overhead barrier would be essential in showing decrease in time as the number of threads increase which is why we implemented a method where the user can input an arbitrary number of variables and constraints (however large), and then fill the matrix appropriately so as to not have to do it manually and make the process more automated in nature.

3.1 Set 1:

The first set of parameters for our analysis tests were conducted on 2000 variables, and 2500 constraints.

3.1.1 Speedup OpenMP

1 thread:
Speedup = T_1/T_P = 9.33285/9.33285 = 1

2 threads:
Speedup = T_1/T_P = 9.33285/4.82102 = 1.93587

4 threads:
Speedup = T_1/T_P = 9.33285/2.56393 = 3.64006

8 threads:
Speedup = T_1/T_P = 9.33285/2.14504 = 4.35090

3.2 Speedup OpenMPI

1 process:
Speedup = T_1/T_P = 6.8366/6.8366 = 1

2 processes:
Speedup = T_1/T_P = 6.8366/4.23266 = 1.6152

4 processes:
Speedup = T_1/T_P = 6.8366/1.69396 = 4.0359

8 processes:
Speedup = T_1/T_P = 6.8366/0.96608 = 7.0766

3.2.1 Efficiency OpenMP

1 thread:
Efficiency = Speedup / N = 1/1 = 100%

2 threads:
Efficiency = Speedup / N = 1.93587/2 = 96.8%

4 threads:
Efficiency = Speedup / N = 3.64006/4 = 91%

8 threads:
Efficiency = Speedup / N = 4.35090/8 = 54.39%

3.2.2 Efficiency OpenMPI

1 processes:
Efficiency = Speedup / N = 1/1 = 100%

2 processes:
Efficiency = Speedup / N = 1.6152/2 = 80.76%

4 processes:
Efficiency = Speedup / N = 4.0359/4 = 100.89%

8 processes:
Efficiency = Speedup / N = 7.0766/8 = 88.46%

3.2.3 Cost OpenMP

1 thread:
Cost = N * T_P = 1 * 9.33285 = 9.33285 sec

2 threads:
Cost = N * T_P = 2 * 4.82102 = 9.64204 sec

4 threads:
Cost = N * T_P = 4 * 2.56393 = 10.25572 sec

8 threads:
Cost = N * T_P = 8 * 2.14504 = 17.16032 sec

3.2.4 Cost OpenMPI

1 process:
Cost = N * T_P = 1 * 6.8366 = 6.8366 sec

2 processes:
Cost = N * T_P = 2 * 4.23266 = 8.46532 sec

4 processes:
Cost = N * T_P = 4 * 1.69396 = 6.77584 sec

8 processes:
Cost = N * T_P = 8 * 0.96608 = 7.72864 sec

3.3 Set 2:

The second set of parameters for our analysis tests were conducted on 400 variables, and 500 constraints.

3.3.1 Speedup OpenMP

1 thread:
Speedup = T_1/T_P = 0.0616897/0.0616897 = 1

2 threads:
Speedup = T_1/T_P = 0.0616897/0.0345321 = 1.78645

4 threads:
Speedup = T_1/T_P = 0.0616897/0.0218881 = 2.81841

8 threads:
Speedup = T_1/T_P = 0.0616897/0.0200901 = 3.07065

3.3.2 Speedup OpenMPI

1 process:
Speedup = T_1/T_P = 0.123218/0.123218 = 1

2 processes:
Speedup = T_1/T_P = 0.123218s/0.0809029 = 1.523

4 processes:
Speedup = T_1/T_P = 0.123218s/0.0381877 = 3.22664

8 processes:
Speedup = T_1/T_P = 0.123218s/0.0277087 = 4.4469

3.3.3 Efficiency OpenMP

1 thread:
Efficiency = Speedup / N = 1/1 = 100%

2 threads:
Efficiency = Speedup / N = 1.78645/2 = 89.32%

4 threads:
Efficiency = Speedup / N = 2.81841/4 = 70.46%

8 threads:
Efficiency = Speedup / N = 3.07065/8 = 38.38%

3.3.4 Efficiency OpenMPI

1 process:
Efficiency = Speedup / N = 1/1 = 100%

2 processes:
Efficiency = Speedup / N = 1.523/2 = 76.15%

4 processes:
Efficiency = Speedup / N = 3.22664/4 = 80.67%

8 processes:
Efficiency = Speedup / N = 4.4469/8 = 55.59%

3.3.5 Cost OpenMP

1 thread:
Cost = N * T_P = 1 * 0.0616897 = 0.06169 sec

2 threads:
Cost = N * T_P = 2 * 0.0345321 = 0.06906 sec

4 threads:
Cost = N * T_P = 4 * 0.0218881 = 0.08755 sec

8 threads:
Cost = N * T_P = 8 * 0.0200901 = 0.16072 sec

3.3.6 Cost OpenMPI

1 process:
Cost = N * T_P = 1 * 0.123218 = 0.123218 sec

2 processes:

Cost = N * T_P = 2 * 0.0809029 = 0.16181 sec

4 processes:
Cost = N * T_P = 4 * 0.0381877 = 0.15275 sec

8 processes:
Cost = N * T_P = 8 * 0.0277087 = 0.2216696 sec

3.4 Set 3:

The third set of parameters for our analysis tests were conducted on 20 variables, and 20 constraints.

3.4.1 Speedup OpenMP

1 thread:
Speedup = T_1/T_P = 0.000484098/0.000484098 = 1

2 threads:
Speedup = T_1/T_P = 0.000484098/0.000296599 = 1.63216

4 threads:
Speedup = T_1/T_P = 0.000484098/0.000415028 = 1.16642

8 threads:
Speedup = T_1/T_P = 0.000484098/0.00077883 = 0.62157

3.4.2 Speedup OpenMPI

1 process:
Speedup = T_1/T_P = 0.0001964/0.0001964 = 1

2 processes:
Speedup = T_1/T_P = 0.0001964/0.0008051 = 0.2439449

4 processes:
Speedup = T_1/T_P = 0.0001964/0.0007766 = 0.252897244398661

8 processes:
Speedup = T_1/T_P = 0.0001964/0.0012705 = 0.154584809130264

3.4.3 Efficiency OpenMP

1 thread:
Efficiency = Speedup / N = 1/1 = 100%

2 threads:
Efficiency = Speedup / N = 1.63216/2 = 81.60%

4 threads:
Efficiency = Speedup / N = 1.16642/4 = 29.16%

8 threads:
Efficiency = Speedup / N = 0.62157/8 = 7.77%

3.4.4 Efficiency OpenMPI

1 process:
Efficiency = Speedup / N = 1/1 = 100%

2 processes:
Efficiency = Speedup / N = 0.2439449/2 = 12.197%

4 processes:
Efficiency = Speedup / N = 0.252897244398661/4 = 6.32%

8 processes:
Efficiency = Speedup / N = 0.154584809130264/8 = 1.93%

3.4.5 Cost OpenMP

1 thread:
Cost = N * T_P = 1 * 0.000484098 = 0.000484098 sec

2 threads:
Cost = N * T_P = 2 * 0.000296599 = 0.000593198 sec

4 threads:
Cost = N * T_P = 4 * 0.000415028 = 0.001660112 sec

8 threads:
Cost = N * T_P = 8 * 0.00077883 = 0.00623064 sec

3.4.6 Cost OpenMPI

1 process:
Cost = N * T_P = 1 * 0.0001964 = 0.0001964 sec

2 processes:
Cost = N * T_P = 2 * 0.0008051 = 0.0016102 sec

4 processes:
Cost = N * T_P = 4 * 0.0007766 = 0.0031064 sec

8 processes:
Cost = N * T_P = 8 * 0.0012705 = 0.010164 sec

3.5 Analysis Conclusion

As per the analysis results, we can clearly see in set 3 that the problem regarding overhead takes over and causes a severe performance drop in terms of parallelization. Furthermore, we also notice that as the parameters increase in size, they tend to perform better in our analysis which also goes to support our initial remark that having a small problem size will result in the overhead taking over and bigger problem sizes are better supported in regard to our parallelized implementation of the Simplex method.
Something else to be noted is that the multi-threaded (2+ processes) performance for the OpenMPI implementation is exceptionally superior in comparison to the OpenMP implementation for sets 1 2. Adversely, we see a rather consequential downfall in performance and the reason for that is as mentioned above. The OpenMPI implementation has massive overhead when the problem size drops below a certain threshold. Comparing the two, we can make the observation that the OpenMPI implementation is definitely faster and more efficient with the trade-off being opposite performance for a small N.