PARALLEL ALGORITHMS FOR THE ASSIGNMENT PROBLEM

Mark DiValerio
UTEID: mad2799
EE W382V: Parallel Algorithms
Professor Vijay K. Garg
Department of Software Engineering
University of Texas at Austin
markadivalerio@gmail.com

Sanjaikanth Pillai
UTEID: se9727
EE W382V: Parallel Algorithms
Professor Vijay K. Garg
Department of Software Engineering
University of Texas at Austin
sanjaikanth@gmail.com

August 12, 2020

ABSTRACT

In this paper, we will dive into the "Assignment Problem". We will discuss the general algorithm and our approach to turning the algorithm into a parallel process. We will do some analysis of our results compared that to its serial version.

***K*eywords** Assignment Problem · Parallel · Algorithm

1 Introduction

The Assignment Problem is a common problem in software engineering, where if you have list of *agents* and a list of *tasks*, the goal is to assign each agent to a task that will best reduce the cost of assigning the agent to that task. (or maximizing profit by optimizing products to their respective stores, etc). This same problem and algorithm can also be described in bipartite graph, optimally choosing weighted edges to minimize (or maximize) the total weight.

2 Approach

We began tackling this issue by researching what known algorithms could solve the "Assignment Problem". Sequentially using a brute force algorithm to check all assignments and the resulting cost would have a time complexity of O($n!$). However the most common algorithm that solves this issue is known as the Hungarian Algorithm. Now as mentioned in the introduction 1, this algorithm can be used in both agents/tasks (aka matrix) terminology or used in bipartite graph. We went with the matrix terminology only because it was easier to put into words and describe

2.1 Hungarian Algorithm

The Hungarian Algorithm has several steps to implement. (These steps and their numbers will be references later in this paper).

1. Normalize (if unbalanced matrix)
   If the input matrix of r agents (rows) and c tasks (columns) are not equal, then you "normalize" it by adding rows or columns until they are equal. The values within the added column is equal to the maximum value of the row, and vice versa, if $c > r$, then the values in the added rows would be the maximum value in the column. (we'll call the normalized size nxn matrix)
2. Invert (if finding maximum)
   If you are trying to find the minimum value (aka the "cost"), then skip this step. Otherwise, if you are trying to find the maximum value (aka "profit"), then we invert the matrix by multiplying every value by -1, and proceed with the algorithm as normal.

3. Subtract the Row Minimum
   For each row, find the smallest value and subtract it from each element in the row
4. Subtract the Column Minimum
   For each column, find the smallest value in the column and subtract it from each element in that column.
5. Cover all Zeros with minimum number of lines
   5.1. Create an empty matrix (we'll refer to this matrix as the "cover" matrix)
   5.2. For every zero in the cost matrix, count the number of zero cost values in the column, and count the number of zero cost values in the row that are not already covered (that are 0's in the cover matrix).
   5.3. if the row's zero count is greater than the column's zero count, highlight or "cover" the row by marking the cover matrix as a -1. if column's zero count is greater, mark the cover matrix's column as a +1. If a cover matrix cell is already highlighted by another line, mark it as an intersection (2).
6. Reduce to create additional zeros.
   For all of the "uncovered" cells (the remaining 0's in the cover matrix), find the minimum value. Subtract that value from all uncovered cells. Add the minimum value to any numbers that are at the intersection of two covered lines ("2" in the cover matrix).
   Repeat the previous 2 steps of reduce and cover until the number of "covered" lines == n
7. Find the solution in the Zeros.
   The solution will be a subset of the zeros in the resulting matrix (which is anywhere between n and $2n$), find the solution by making sure every row and every column is represented by a zeros (without double-dipping). This set of zero cells' x/y (or r/c) matrix coordinates are the proper assignments.
8. Get the total cost
   Use the cells coordinates from the previous step with the matrix's original cost values to find

The serial version of this algorithm can be done in O(n^3) time complexity.

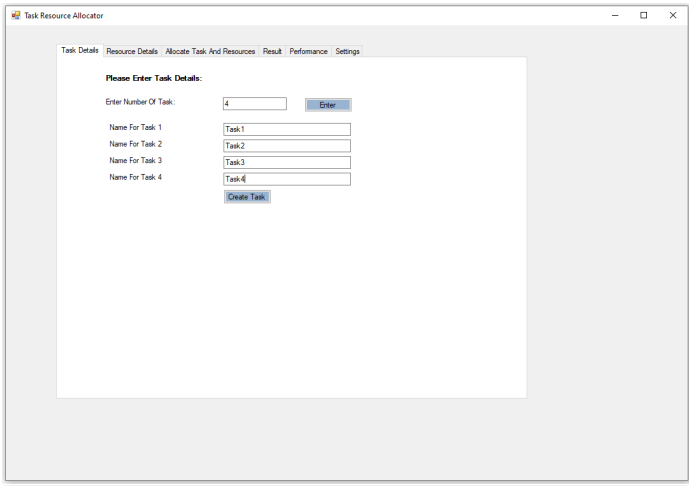
2.2 Parallel Implementation

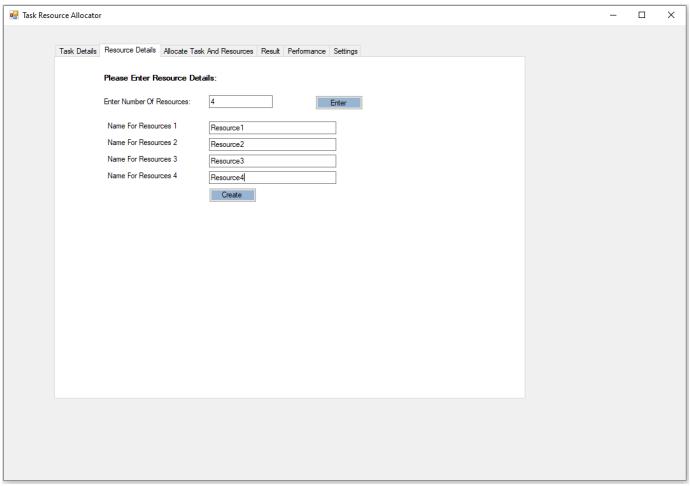
When planning to parallel the Hungarian Algorithm, we decided to go with the C# programming language as it was easier to work with in terms of how to use its parallel features. With the occasional locking and unlocking of some higher-scoped variables, it was little in terms of managing memory and avoiding race conditions.

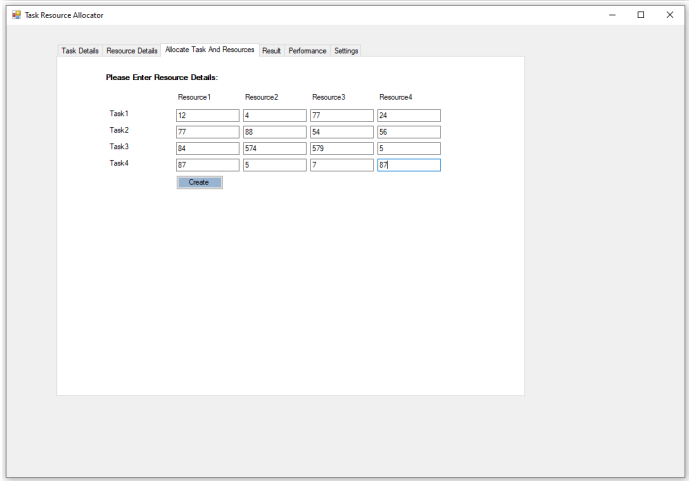
Almost every step described 2.1 is independent and almost every calculation done to a specific row/column ("cell") is independent of the calculation done to other cells within the matrix. Assuming you had an infinite number of processes/threads, you could run almost every step in O(1) time. That being said, steps 5 and 7 (Covering Zeros and Finding the solution in the zeros) are the only serial portions of the algorithm. Step 5's Covering zeros determines requires the knowledge of what has been covered and what hasn't been covered yet, and "parallelizing" the covering of the rows could alter the outcome. Step 7 (Finding the solution in the zeros) is serial because while we've greatly reduced the matrix to the possible values, we're still unsure of which zeros are included in the final result and which are not. To find this out, we have to iterate through each permutation and verify that all columns and rows are uniquely covered.

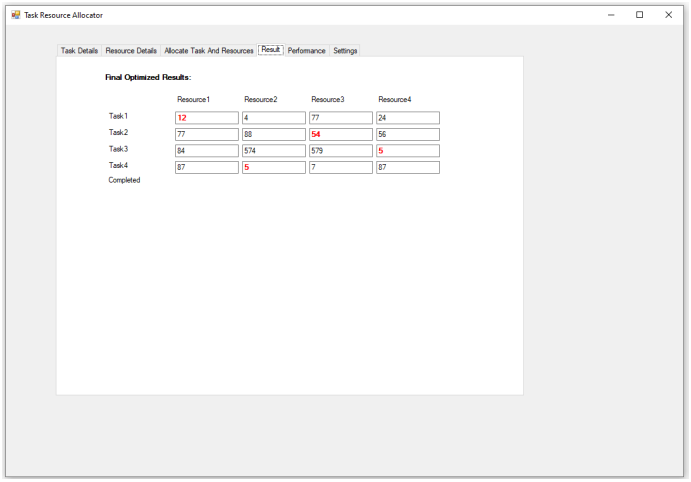
3 UI Screenshot

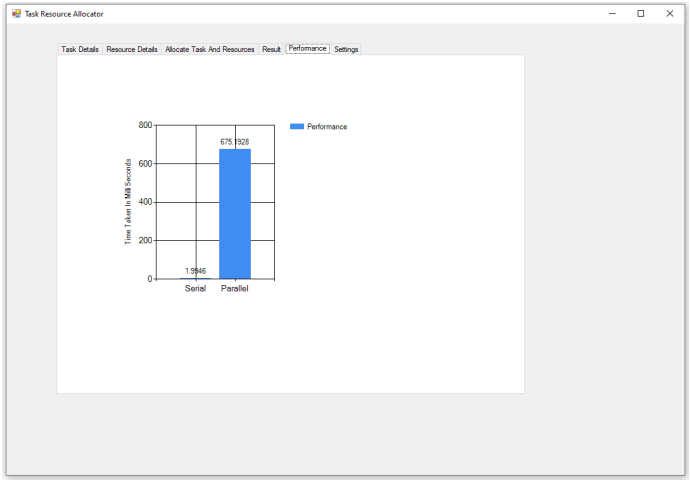
Below are the UI screenshot for determining Hungariam Algorithm.
In the first tab UI will accept task details.
In second tab UI accept resource tab.
Thir tab it accepts the cost of each matrices.
Fourth screen shows the minimum Hungarian algorithm.
Last screen shows the performance analyses of the algorithm calculation between serial and parallel algorithm.





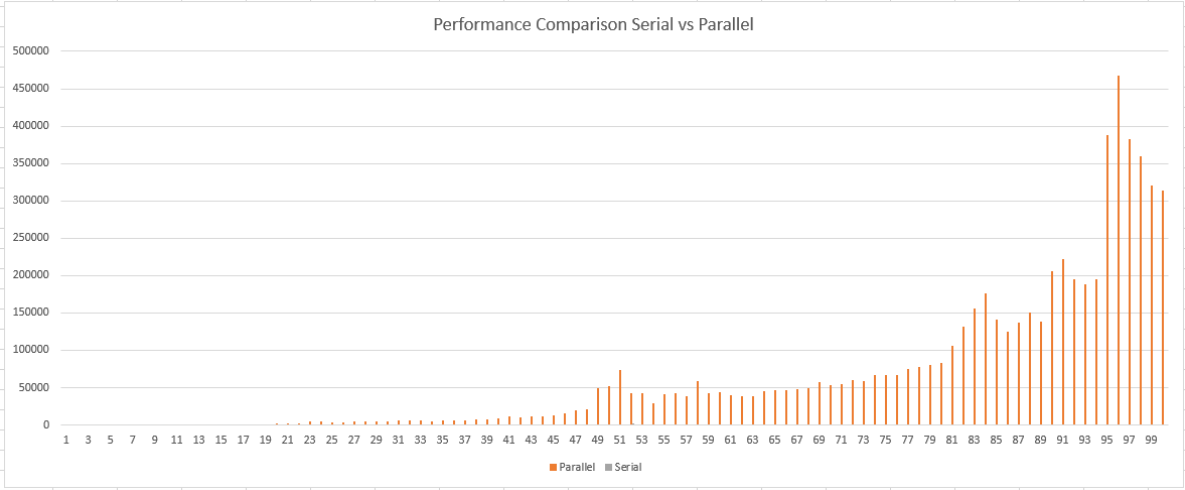






4 Analysis and Results

4.1 Results



The data may be seen here:
`https://drive.google.com/file/d/1ZjQ_cphMRNs5sRsMXRX_H908EO7Ix3kr/view?usp=sharing`

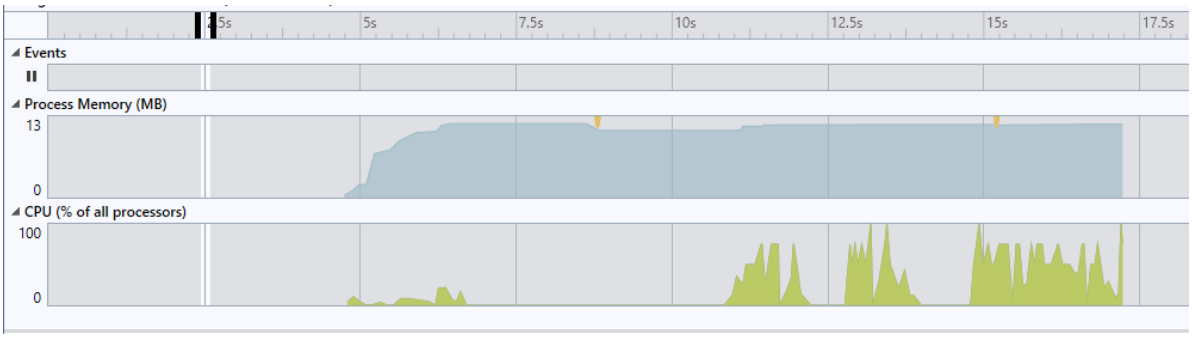
4.2 Analysis

Unfortunately, our parallel version of Hungarian's Algorithm actually performed much worse than the serial version of Hungarian's Algorithm. This is most likely due to the alteration of Step 5-7. We had some difficulties trying to convert "find the minimum number of lines that would cover the most zeros" into a parallel algorithm.

Our second thought was that this was due to the language itself. Several online articles suggested that the splitting and merging of parallel threads within C# would actually take longer time if the code within the for loop was simple. However we attempted several other resources in other languages. This included C++ serial vs CUDA graphics processing. These tests also confirmed out findings, that serial version was faster than parallel. We attempted to investigate why but we could not pinpoint the cause. We suspect that the step of covering the zeros with the minimum number of lines was the major cause though, as verbose timestamps would support this.

4.3 Performance Analyses

We did performance analyses for the parallel implementation of Hungarian Algorithm. We noticed that the implementation took 12.2 MB maximum memory. Max CPU used was 100% for some of the duration, but all other time it was consistently lower.

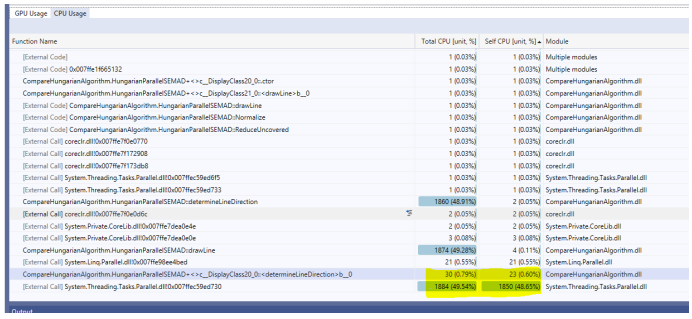


Below are the CPU usage statistics . Below image shows CPU statistics sorted by Total CPU by functions. Total CPU is the time taken by the function as well as time taken by the functions called by the given function.

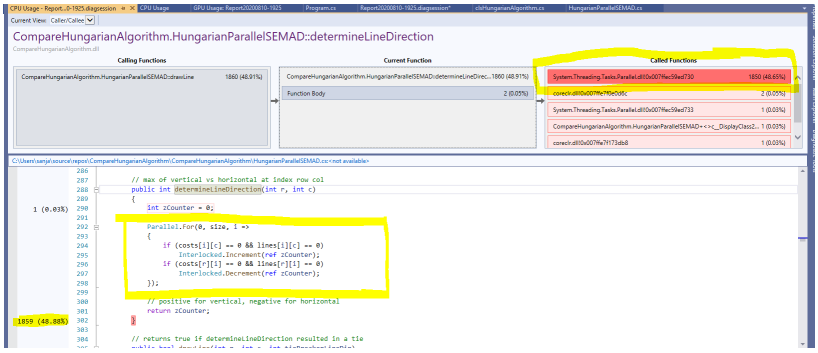
GPU Usage | CPU Usage

Function Name	Total CPU [unit,...	Self CPU [unit, %]	Module
dotnet.exe (PID: 13976)	3803 (100.00%)	0 (0.00%)	dotnet.exe
CompareHungarianAlgorithm.Program::Main	1889 (49.67%)	0 (0.00%)	CompareHungarianAlgorithm.dll
CompareHungarianAlgorithm.HungarianParallelSEMAD::Run	1884 (49.54%)	0 (0.00%)	CompareHungarianAlgorithm.dll
[External Call] System.Threading.Tasks.Parallel.dll!0x007ffec59ed730	1884 (49.54%)	1850 (48.65%)	System.Threading.Tasks.Parallel.dll
CompareHungarianAlgorithm.HungarianParallelSEMAD::CoverZeros	1876 (49.33%)	1 (0.03%)	CompareHungarianAlgorithm.dll
CompareHungarianAlgorithm.HungarianParallelSEMAD::CoverZeros	1875 (49.30%)	0 (0.00%)	CompareHungarianAlgorithm.dll
CompareHungarianAlgorithm.HungarianParallelSEMAD::drawLine	1874 (49.28%)	4 (0.11%)	CompareHungarianAlgorithm.dll
CompareHungarianAlgorithm.HungarianParallelSEMAD::determineLineDirection	1860 (48.91%)	2 (0.05%)	CompareHungarianAlgorithm.dll
CompareHungarianAlgorithm.HungarianParallelSEMAD+<>c__DisplayClass20_0::<determin...	30 (0.79%)	23 (0.60%)	CompareHungarianAlgorithm.dll
[Unwalkable]	25 (0.66%)	0 (0.00%)	Multiple modules
[External Call] System.Linq.Parallel.dll!0x007ffe98ee4bed	21 (0.55%)	21 (0.55%)	System.Linq.Parallel.dll
CompareHungarianAlgorithm.HungarianParallelSEMAD+<>c__DisplayClass15_0::<getCols...	15 (0.39%)	0 (0.00%)	CompareHungarianAlgorithm.dll
CompareHungarianAlgorithm.HungarianParallelSEMAD+<>c__DisplayClass14_0::<getRows...	12 (0.32%)	0 (0.00%)	CompareHungarianAlgorithm.dll
CompareHungarianAlgorithm.HungarianParallelSEMAD+<>c__DisplayClass11_0::<Run>b__0	5 (0.13%)	0 (0.00%)	CompareHungarianAlgorithm.dll
CompareHungarianAlgorithm.HungarianParallelSEMAD+<>c__DisplayClass12_0::<Normaliz...	4 (0.11%)	0 (0.00%)	CompareHungarianAlgorithm.dll
CompareHungarianAlgorithm.HungarianParallelSEMAD::.ctor	4 (0.11%)	0 (0.00%)	CompareHungarianAlgorithm.dll
CompareHungarianAlgorithm.HungarianParallelSEMAD::getRowsMins	3 (0.08%)	0 (0.00%)	CompareHungarianAlgorithm.dll
CompareHungarianAlgorithm.HungarianParallelSEMAD::Normalize	3 (0.08%)	0 (0.00%)	CompareHungarianAlgorithm.dll
CompareHungarianAlgorithm.HungarianParallelSEMAD::SubRowsMin	3 (0.08%)	0 (0.00%)	CompareHungarianAlgorithm.dll
[External Call] System.Private.CoreLib.dll!0x007ffe7dea0e0e	3 (0.08%)	3 (0.08%)	System.Private.CoreLib.dll
[External Call] coreclr.dll!0x007ffe7f0e0d6c	2 (0.05%)	2 (0.05%)	coreclr.dll

Below image shows CPU usage sorted by Self CPU. Self CPU is the CPU time tekn by the given function only and does not include time taken by any inner functions it called. We can see parallel execution of threads took longer cpu time.



We can see that below highlighted parallel execution took 48.65% of the total CPU time.The highlighted parallel execution blocks took more time to finish.



5 Conclusion

5.1 Expanding & Future Work

If we had more time to invest in this project, some future work could be adjusting the two serialized steps (Steps 5 and 7) into parallel. There was not much information we could find in how to accomplish covering all the zeros in a cost matrix with the least number of lines, and while our algorithm did work, it could be optimized a bit more, especially for parallel implementation. Step 7 of finding the solution after all of this reduction feels a bit untouched and unrefined as there should be other methods of finding a possible outcome rather than looking through all permutations (finding the "first" that returns a good result). Given more time, we would like to investigate better options to these steps so the algorithm as a whole could be fully implemented in parallel.

Another route we would like to have investigated is memory and performance on graphics cards using CUDA and the CUDA language. While we did find external resources implementing cuda and C++, due to the time restrictions and our personal resource limitations, we were not able to make our own implementation and investigate this branch of the hungarian algorithm further.

5.2 Conclusion

In conclusion, you can see how our C# implementation of the Hungarian Algorithm compares to its serial version. Unfortunately we could not get the performance to improve over the serial. This is most likely due to the alterations we had to do after step 5.

6 Resources

1. **Unit Test Case Source Code:**
`https://github.com/markadivalerio/parallel_algorithms/tree/master/project/project/project`

2. **User Interface Source Code:**
`https://github.com/markadivalerio/parallel_algorithms/tree/master/project/ResourceAllocator/ResourceAllocator`

3. **Performance Analyses Source Code:**
`https://github.com/markadivalerio/parallel_algorithms/tree/master/project/CompareHungarianAlgorithm/CompareHungarianAlgorithm`

4. **Powerpoint Presentation**:
`https://docs.google.com/presentation/d/1BfwLW3g7eBXLuWwIioiOOQjvmupAx5-MMKyv7cIA9R8/edit?usp=sharing`

5. **Data analysis**:
`https://drive.google.com/file/d/1ZjQ_cphMRNs5sRsMXRX_H908EO7Ix3kr/view?usp=sharing`