Parallel Algorithms for the Assignment Problem

By Mark DiValerio
and Sanjaikanth Pillai

The Assignment Problem

- The assignment problem is a common software problem that assigns each “agent” to a “task” (with at most 1 agent per task) to minimize cost, or to maximize profit.
- There are several algorithms that can solve the Assignment Problem, but the most widespread and common solution is the Hungarian Algorithm.
- It can also be used to find the most optimal weighted path in a bipartite graph

History

The Hungarian method is a combinatorial optimization algorithm that solves the assignment problem in polynomial time .

- It was developed and published in 1955 by Harold Kuhn.
- He gave the name "Hungarian method".
- Algorithm was largely based on the earlier works of two Hungarian mathematicians: Dénes Kőnig and Jenő Egerváry.

The Hungarian Algorithm - part 1

1. (Preparation) - Normalize & Invert
    a. If given an unbalanced n*m matrix, add columns or rows containing the maximum value so that it is a square matrix
    b. If finding the maximum total value (i.e. “profit” instead of the minimum “cost”), multiply every value in the matrix by -1
2. Subtract the row’s minimum from every value in the row for every row.
3. Subtract the column’s minimum from every value in the column for every column.

The Hungarian Algorithm - part 2

4. Find the minimum number of horizontal/vertical lines that would cover all the zeros in the matrix.

5. If the number of covering lines < n, Reduce to generate more zeros.

- Find the minimum “uncovered” value
- subtract the minimum value from all uncovered numbers
- Add that minimum value to any intersecting “covered” numbers, the numbers that are covered by 2 lines (one horizontal one vertical)
- Repeat steps 4-5 until the number of covering lines == n.

6. Choose n zeros that uniquely covers every row and column.
(use their row/column location with the original input matrix to get total cost/profit).

2) Row Reduction

30	25	10		10		20	15	0
15	10	20	-	10	=	5	0	10
25	20	15		15		10	5	0

(In Step 1, it’s an n*n matrix so no “normalization”. We’re also finding the minimum total value so we don’t need to invert the matrix)

3) Column Reduction

20	15	0		15	15	0
5	0	10	=	0	0	10
10	5	0		5	0	0
	-					
5	0	0				

4) Cover Zeros & 5) Shift Zeros

Cover the zeros in the matrix with the minimum number of lines.

Minimum uncovered = 5

Subtract minimum from all uncovered, add minimum to all intersections.

Repeat until # of lines == n (in this case n=3)

6) Choose solution

10	10	**0**		30	25	10
0	0	15	==>	15	10	20
0	**0**	0		25	20	15

Choose a set of zeros that will uniquely cover all rows and columns

(if there are multiple, any would work). Use original matrix to get total cost

(0,2) + (1,0) + (2,1)

10 + 15 + 20 = 45

(0,2) + (1,1) + (2,0)

10 + 10 + 25 = 45

Parallelizing Hungarian Algorithm

- Because almost every step is matrix manipulation, and every manipulation's inner iterations can be done independently (Step 1's row reduction subtracts every row with the minimum value ***of that row***), most of Hungarian's Algorithm can be easily parallelized.
- Covering Zeros is tricky as it requires to find the minimum lines that cover the zeros. Also the last step of getting the solution from the zeros, uniquely covering all rows and columns is difficult to parallelize due to each iteration depends on the previous iteration.

Implementation

- Programming language in the C# Language version 4.7.2
- Visual Studio 2019 version 16.2.4

```
Parallel.For(<start>, <stop>, (<variable>, <state>) =>{
    //parallel code here
    lock(<obj variable>) {
        // code utilizing “locked” object
    }
    Interlocked.Increment(ref <numericalVariable>);
});
```

Applications

As part of the project we created three applications :

- Hungarian Algorithm with unit testing.
- Resource Allocator (UI for finding minimum bipartite matching)
- Performance Analyser

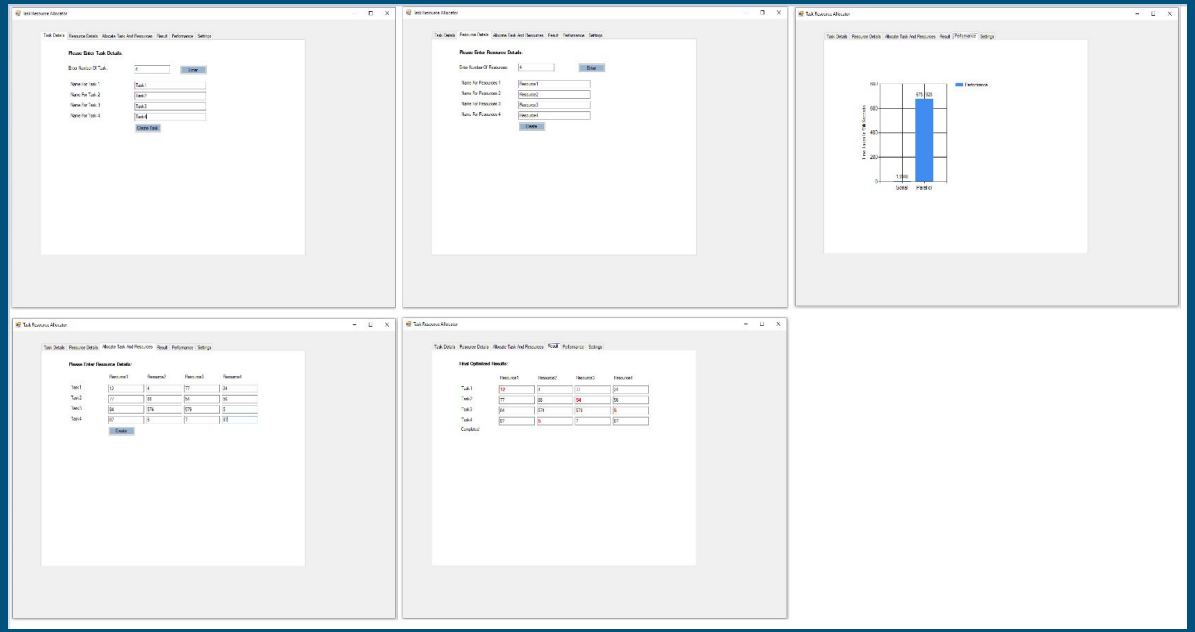
Hungarian Algorithm

```
Matrix:
 49  52  72  82  22  54  77  77   6  52
  9  38  51  11  39  16  77  93  37   3
 60  25  71  30  50  60  69  28  44  62
 34  54  81  14  10  89  69  21  29  69
 85  90  73  18  76  41   4  87  26  10
 17  64  67  72  48  71  81  89  79  62
 32  96  80  44  65   2  57  37  48  97
 94   3  64  15  18  17  99  53  54   9
  3  82  49  81  45  70  34  56  13  86
  8  80  42  67  37  36  57  29  53  74
```

```
Matrix (10x10)
 53  44  30  70  14  46  69  66   0  44
 18  35  14   4  36  13  74  87  36   0
 47   0  12   1  25  35  44   0  21  37
 36  44  37   0   0  79  59   8  21  59
 93  86  35  10  72  37   0  80  24   6
  0  35   4  39  19  42  52  57  52  33
 42  94  44  38  63   0  55  32  48  95
103   0  27   8  15  14  96  47  53   6
  0  67   0  62  30  55  19  38   0  71
 12  72   0  55  29  28  49  18  47  66

Matrix (10x10)
  1   0   1   0   0   0   0   0   1   0
  2  -1   2  -1  -1  -1  -1  -1   2  -1
  2  -1   2  -1  -1  -1  -1  -1   2  -1
  2  -1   2  -1  -1  -1  -1  -1   2  -1
  2  -1   2  -1  -1  -1  -1  -1   2  -1
  1   0   1   0   0   0   0   0   1   0
  2  -1   2  -1  -1  -1  -1  -1   2  -1
  2  -1   2  -1  -1  -1  -1  -1   2  -1
  1   0   1   0   0   0   0   0   1   0
  1   0   1   0   0   0   0   0   1   0

Minimum Uncovered Cell is (0,4) = 14
Matrix (10x10)
 53  30  30  56   0  32  55  52   0  30
 32  35  28   4  36  13  74  87  50   0
 61   0  26   1  25  35  44   0  35  37
 50  44  51   0   0  79  59   8  35  59
107  86  49  10  72  37   0  80  38   6
  0  21   4  25   5  28  38  43  52  19
 56  94  58  38  63   0  55  32  62  95
117   0  41   8  15  14  96  47  67   6
  0  53   0  48  16  41   5  24   0  57
 12  58   0  41  15  14  35   4  47  52

2,7 = 0 : horizontal
3,3 = 0 : horizontal
5,0 = 0 : vertical
9,2 = 0 : vertical
0,4 = 0 : horizontal
1,9 = 0 : horizontal
4,6 = 0 : horizontal
6,5 = 0 : horizontal
7,1 = 0 : horizontal
8,8 = 0 : horizontal
4487
Array:
   0 is assigned to    4 with cost of   22 for a total cost of   22
   1 is assigned to    9 with cost of    3 for a total cost of   25
   2 is assigned to    7 with cost of   28 for a total cost of   53
   3 is assigned to    3 with cost of   14 for a total cost of   67
   4 is assigned to    6 with cost of    4 for a total cost of   71
   5 is assigned to    0 with cost of   17 for a total cost of   88
   6 is assigned to    5 with cost of    2 for a total cost of   90
   7 is assigned to    1 with cost of    3 for a total cost of   93
   8 is assigned to    8 with cost of   13 for a total cost of  106
   9 is assigned to    2 with cost of   42 for a total cost of  148
```

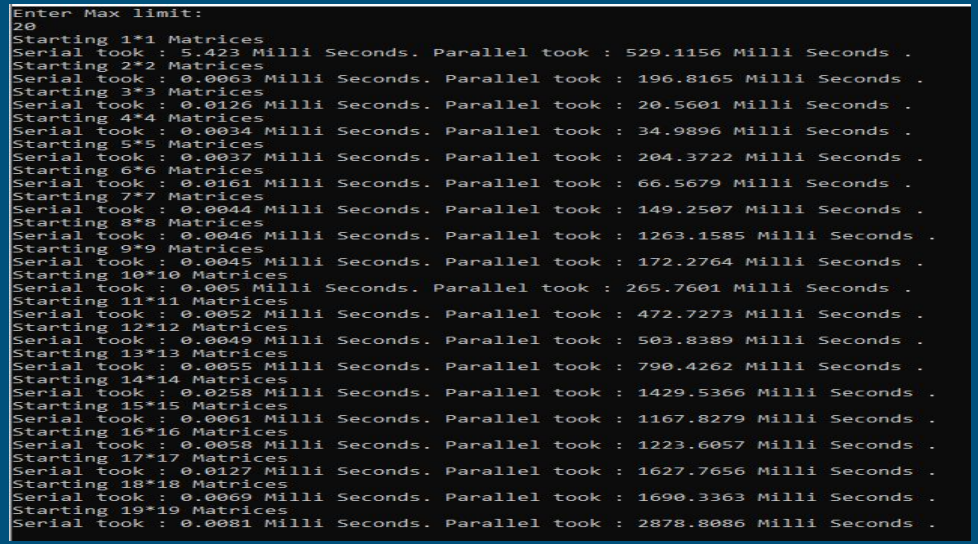
Resource Allocator

Resource Allocator is UI application for finding Minimum Bipartite Matching.

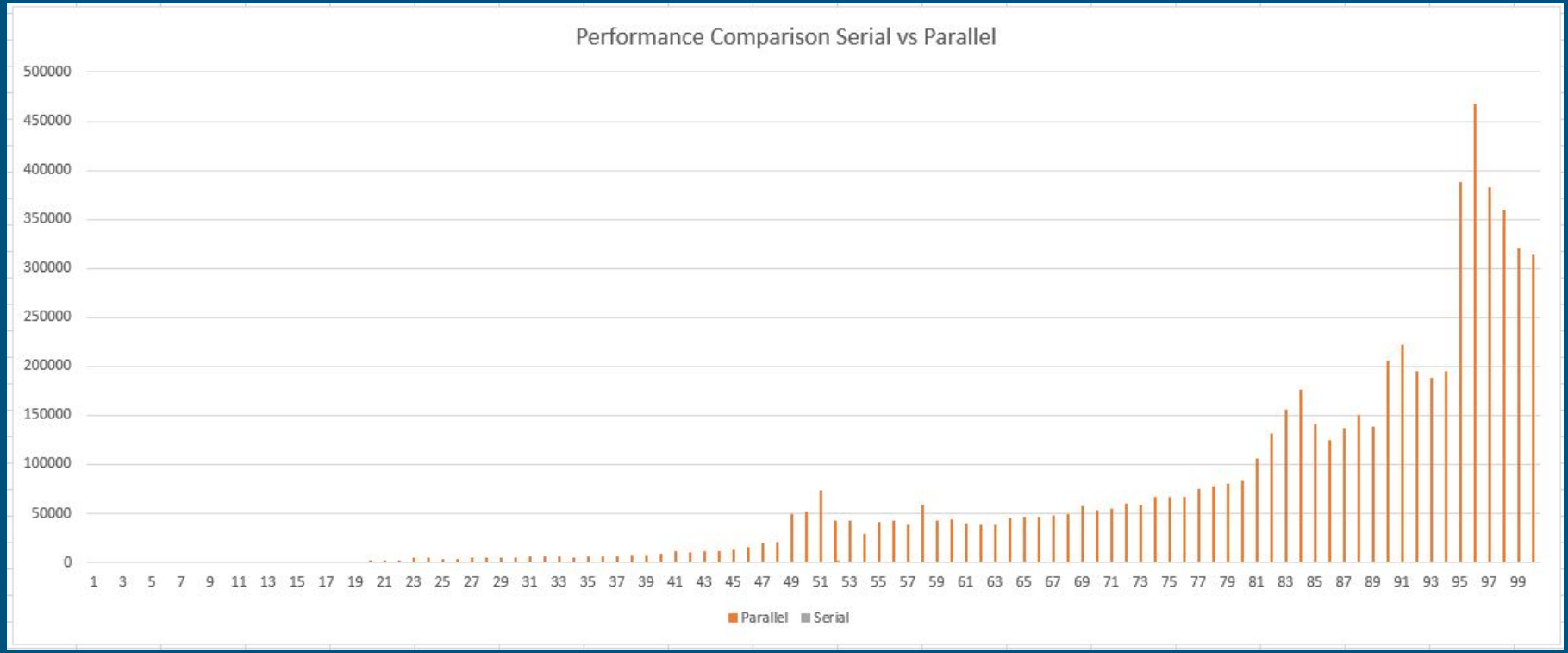


Performance Analyser

Performance Analyser is a console application that compares execution between serial and parallel implementation.



Parallel Hungarian Algorithm Results



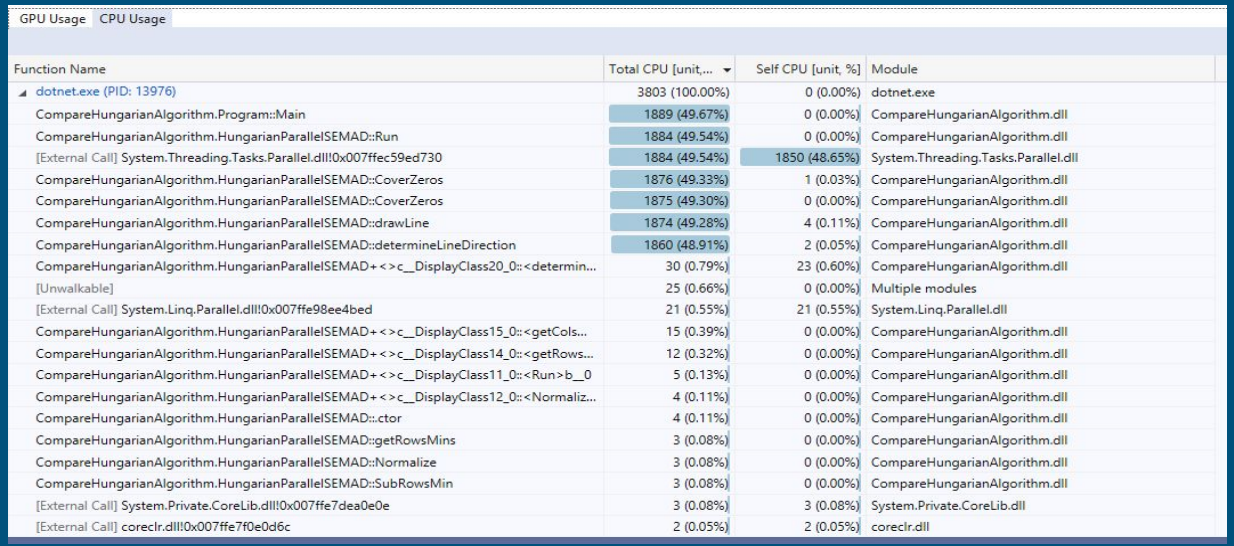
Analyses

Below are the analyses for running Hungarian Parallel implementation:

- Max Memory used:12.2 MB.
- Max CPU:100%

Analyses :CPU Usage

Below are the CPU usage by functions:
CoverZeros() is taking more CPU unit compared to rest of the core functions.



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

Demo

Questions?