Parallel GPU Implementation for the Optimal Assignment Algorithm

Joseph Greshik
Department of Computer Science
Colorado School of Mines
Golden, CO, USA
jgreshik@mines.edu

Allee D. Zarrini
Department of Computer Science
Colorado School of Mines
Golden, CO, USA
adzarrini@mines.edu

Abstract

This paper implements the well-known solution to the optimal assignment problem, also known as the Hungarian algorithm, on GPU infrastructure. Using optimization techniques such as intelligent data structure design and memory copy optimization alongside multithread model parallelism, we propose a GPU-based solution to the difficult-to-parallelize iterative Hungarian algorithm $O(n^3)$ implementation in order to achieve speedup relative to the optimized CPU implementation. The serial implementation is done in C++ and the parallel implementation is done using host C++ code with CUDA code for the GPU implementation. The model and data-level parallelization of the algorithm is executed on an Nvidia Tesla K20m GPU.

CCS Concepts • **Computing Methodologies → Modeling and Simulation**; *Parallel computing methodologies*;

Keywords Combinatorial optimization, Strong polynomial optimization

ACM Reference Format:
Joseph Greshik and Allee D. Zarrini. Fall 2019. Parallel GPU Implementation for the Optimal Assignment Algorithm. In *Proceedings of (CSM CSCI580).* ACM, New York, NY, USA, 7 pages.

1 Background and Related Work

The optimal assignment problem, also referred to as the Hungarian algorithm, is a crucial algorithm in the field of operations research [1]. The Hungarian algorithm solves the task assignment optimization problem in algorithm form.

Imagine you are in charge of assigning n tasks to m workers, with an associated cost value $c_{n,m}$ per task per worker. The task is to minimize $c_{n,m}$ while giving a unique job to each worker.

The cost of finishing all n jobs by an assignment is given by

$$C = \sum c_{n,m} \tag{1}$$

which is the sum of all $c_{n,m}$ job costs, dependent on the chosen job allocation. This sum, which we will denote as C, can be thought of as the total *cost* or *value* of the chosen job allocation. The Hungarian algorithm solves for the optimal value of C.

This optimization problem can be modified easily to be either a *minimum* or *maximum* matching problem, giving a wide array of applications in the field of operations research. To convert a *minimum* score-solving problem to a *maximum* score-solving problem, let $\Delta = \max(c_{ij})$ and adjust all the weights using: $c_{ij} = \Delta - c_{ij}$ and *then* solve for the maximum. A similar approach can be made by negating the original matrix edge weights (setting $c_{ij} = -c_{ij}$). In order to get results for the *minimum* matching problem we return the elements of the original cost matrix that are outputted by the *maximum* algorithm on the converted cost matrices.

$$\begin{bmatrix} 3 & 2 & 3 \\ 2 & 1 & 4 \\ 1 & 7 & 1 \end{bmatrix} \xrightarrow{c_{ij}=7-c_{ij}} \begin{bmatrix} 4 & 5 & 4 \\ 5 & 6 & 3 \\ 6 & 0 & 6 \end{bmatrix} \xrightarrow{\text{orig. mapping}} \begin{bmatrix} 3 & 2 & 3 \\ 2 & 1 & 4 \\ 1 & 7 & 1 \end{bmatrix}$$

Figure 1. The Hungarian algorithm can select the values that generate the maximum matching of the matrix using a minimum-matching approach after either applying $c_{ij} = \Delta - c_{ij}$ or a negation to the matrix edge weights.

Problem Motivation

A simple task for a small number of workers (n) and jobs (n), the Hungarian algorithm has been shown to require a time complexity of $O(n^3)$ in polynomial time to compute square $n \times n$ matrix [2].

This time complexity makes the algorithm increasingly difficult to apply to matrices with increasing values of n. Memory complexity increases quadratically with input size.

It has been proven that the Hungarian algorithm has a strictly polynomial solution [3]. This means that with the current accepted time complexity, $O(n^3)$, we can assume there exists an optimal time-complexity solution to the algorithm.

The Hungarian algorithm computational bottleneck of time complexity makes a GPU implementation the ideal choice for the computation of the optimal assignment for a data-set. Modern GPU architectures have proven to increase CPU hours at a scale that is unmatched by general-purpose CPU performance.

CSM CSCI580, F19, Golden, CO, USA
Fall 2019.

With a GPU implementation we propose a speedup compared to the most optimized CPU implementation of the Hungarian algorithm. GPU architecture may even allow for the finding of a solution that is above expectations according to GPU speedup over CPU due to certain leverages over temporal and spacial locality of the input data.

2 Approach and Rationale

To calculate the optimal score for an $n \times n$ matrix we use a bipartite-graph representation of job to worker matching, and we solve for a minimum-cost maximum bipartite matching.

Our implementation will use a bipartite matrix interpretation of the assignment values for a given cost matrix A. Our implementation will follow the general algorithm for sequentially solving the linear assignment problem given by equations 2, 3, 4, and 5 [1].

For the minimum-matching problem, find

$$\min \quad \Sigma_{i=1}^{n}\Sigma_{j=1}^{n} c_{ij}x_{ij} \tag{2}$$

$$\text{s.t. } \Sigma_{j=1}^{n} x_{ij} = 1 \quad \forall\, i = 1, \ldots, n; \tag{3}$$

$$\Sigma_{i=1}^{n} x_{ij} = 1 \quad \forall\, j = 1, \ldots, n; \tag{4}$$

$$\text{and } x_{ij} \in \{0, 1\} \quad \forall\, i, j = 1, \ldots, n, \tag{5}$$

where x_{ij} are the edge weights given in A, and $c_{ij} \in \{0, 1\}$ are values assigned to corresponding x_{ij} that we assign $c_{ij} = 0$ if we do not want to include the weight of edge x_{ij} in our assignment.

Finding an optimal matching will require fining an augmenting path in the $O(n^3)$ implementation, this will be the main focus of parallel speedup for the algorithm.

Hungarian By Parts

Implementation $O(n^4)$

The $O(n^4)$ implementation can be dissolved into 4 different steps. Consider the 3×3 adjacency matrix below. We have 3 workers with costs corresponding to 3 tasks represented by the adjacency matrix A, and Bipartite graph $B = (U, V, E)$. Where B is our bipartite graph, U is the left set of vertices or workers, V is the right set of vertices or tasks, and E is the set of edges or the cost.

$$A = \begin{bmatrix} 1 & 4 & 5 \\ 5 & 7 & 6 \\ 5 & 8 & 8 \end{bmatrix}$$

Figure 2. Our initial graph A we wish to calculate a minimum matching for.

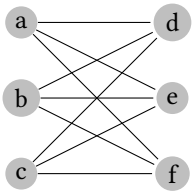

Figure 3. Matrix A represented as bipartite graph, where nodes a, b and c are the nodes represented by the rows of A and nodes d, e, and f are the columns of A. Edge weights between

Step 1. The first step in this algorithm is to do a row-wise minimization of the matrix. We define the set of Rows R and an individual row $r_k \in R$. Given a row r_k, we label the minimum element of the row $\Delta_k = \min_{k=i}(c_{ij})$. Then for each element c_{ij} where $i = k$, we set $c_{ij} = c_{ij} - \Delta_k$.

This is identical to subtracting the minimum edge weight coming out of vertex u from all the edges leaving u for all $u \in U$.

$$A = \begin{bmatrix} 0 & 3 & 4 \\ 0 & 2 & 1 \\ 0 & 3 & 3 \end{bmatrix}$$

Figure 4. Matrix A after applying row minimization.

Step 2. This step is similar to step one, but here we perform a column-wise minimization. Computing this is identical to subtracting the minimum edge weight coming into vertex v from all the edges entering v for all $v \in V$.

$$A = \begin{bmatrix} 0 & 1 & 3 \\ 0 & 0 & 0 \\ 0 & 1 & 2 \end{bmatrix}$$

Figure 5. Matrix A after applying column minimization.

Step 3. In this step, we would like to draw the minimum number of lines to completely cover all of the 0s in A. If the number of lines equal n, then confirm an optimal matching of workers to tasks.

$$A = \begin{bmatrix} 0 & 1 & 3 \\ \cancel{0} & \cancel{0} & \cancel{0} \\ 0 & 1 & 2 \end{bmatrix}$$

Figure 6. We draw the minimum number of lines to to completely cover all 0s in A after row and column minimizations.

In this case, there is no optimal matching yet. As the number of lines, 2, does not equal n. We create a subgraph $B_l = (U, V, E_l)$ where E_l is the subset of E such that for all edges $e \in E$, the cost of the edge c_e is 0. The resultant graph B_l is displayed.

$$E_l = \{e \mid e \in E \text{ and } c_e = 0\}$$

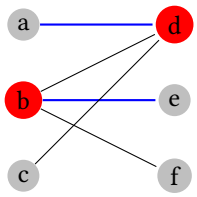

Figure 7. Initial bipartite B_l, where we only consider the 0 weight edges in A as unweighted edges. We see that the minimum vertex cover (shown in red) of B_l is $S = \{b, d\}$. We see the maximum matching (shown in blue) of B_l. Notice that the cardinality of the maximum matching is equal to the cardinality of vertex cover.

The graphical equivalent of the line-covering operation is the minimum vertex cover of the graph B_l. A horizontal line in A can be represented as a cover on a vertex $u \in U$ and a vertical line in A can be represented as a cover on a vertex $v \in V$.
The minimum vertex-cover problem is NP-hard. However, Konigs theorem states the cardinality of the minimum vertex cover of a bipartite graph is equivalent to the cardinality of its maximum matching.
Maximum matching in a bipartite graph can be solved by running max-flow. The algorithm generally utilized in this step is the Ford-Fulkerson algorithm; however, push-relabel algorithm is more commonly parallelized and would be the algorithm of choice for this step [4]. Let's define E_M as the set of edges in our maximum matching where $E_M \subseteq E_l \subseteq E$. Let's also define M as the set of vertices where the edges E_M contain endpoints. Let $Z = U \setminus M$, meaning it is the set of vertices in U, such that for all u in Z, there are no edges in E_M connected to vertex u. Next, we add to Z all vertices that can be travesed through an alternating path. An alternating path, is one such that edges alternate from being a member of E_M, and not in E_M. Once Z is clearly defined, the vertex cover S can be defined as

$$S = (U \setminus Z) \cup (V \cap Z)$$

Step 4. Once S is defined, we modify matrix A in the following ways. We find Δ where

$$\Delta = \min_{i \notin S, j \notin S} (c_{ij}),$$

and modify the every element $c_{i,j}$ of A according to the criterion

$$c_{ij} = \begin{cases} c_{ij} - \Delta & i \notin S \wedge j \notin S \\ c_{ij} & i \notin S \vee j \notin S \\ c_{ij} + \Delta & i \in S \wedge j \in S \end{cases}.$$

The resultant adjacency matrix for A is shown in figure 8. We repeat steps 3 and 4 until a solution is reached.

$$A = \begin{bmatrix} 0 & 0 & 2 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Figure 8. Matrix A after applying the delta value update.

In this scenario, a valid solution is reachable because the number of lines to cover all 0s is equal to n or equivalently, the cardinality of the maximum matching of B_l is equal to n. We choose a valid combination 0s, such that there is only one 0 per row and column. In this example, the minimum cost matching is equal to 15.

$$A = \begin{bmatrix} 0 & 0 & 2 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \longrightarrow \begin{bmatrix} 0 & 0 & 2 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \longrightarrow \begin{bmatrix} 1 & 4 & 5 \\ 5 & 7 & 6 \\ 5 & 8 & 8 \end{bmatrix}$$

Figure 9. Optimal matching for matrix A found to be $a \longrightarrow d$, $b \longrightarrow f$, $c \longrightarrow e$.

There are many opportunities for speedup through parallelization in the $O(n^4)$ implementation. Row-wise and column-wise parallelization of steps 1 and 2 is trivial. There exist non-trivial parallelization approaches for the push-relabel algorithm of max-flow running in $O(V^2E)$ time [4]. Converting from bipartite matching to vertex cover will be the main bottle-neck in the serial implementation. Step 4 can also be parallelized in a row-wise fashion.

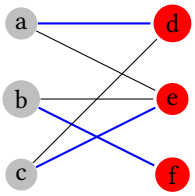

Figure 10. At the finish of the $O(n^4)$ algorithm, we find the optimal matching for the workers in A is given by the edges chosen with $c_{ij} = 0$ and unique i (row) and j (column) values.

Implementation $O(n^3)$

The $O(n^3)$ implementation uses a slightly different approach to solve this. Conversely to the $O(n^4)$ implementation which naturally finds the minimum matching, the $O(n^3)$ implementation finds the maximum matching. We can covert from minimum and maximum using the metric defined in section 1.

Before we begin, we will describe some notation. Once again, let the bipartite graph be $B = (U, V, E)$.

Vertex and Set Neighborhood

Let the function $J_G(v)$ represent all the vertices in the neighborhood of vertex v. A neighborhood is a set of vertices that share an edge with v. Therefore, we can define $J_G(S)$ as the set of vertices that share an edge with any vertex in S. For $v \in U, V$, we define

$$J_G(v) = \{u, (u, v) \in E\}.$$

And if we let $K \subseteq U, V$, we define

$$J_G(K) = \bigcup_{v \in K} J_G(v) \quad [5].$$

Vertex Labeling

For each vertex, we define a function $l(v)$ that takes as input a vertex v and assigns an integer to the vertex. Each vertex is assigned a label such that

$$l(u) + l(v) \leq w(u, v), \forall u \in U, \forall v \in V \, ,$$

where $w(u, v)$ is the weight of the directed edge going from u to v [5].

Equality Subgraph

$B_I = (U, V, E_I)$ is a subgraph of B, where

$$E_I = \{(u, v) \mid (u, v) \in E \wedge l(u) + l(v) = w(u, v)\} \, .$$

In other words, B_I includes only the edges in B such that the labeling of the endpoints of the edges equal the weight of the edge. If we define M_I as the perfect matching of the graph B_I, then M_I is the maximum matching of B [5].

Alternating Path

Consider a matching of B as $M \subseteq E$. A vertex is considered matched if contains an edge $e \in M$, otherwise it is exposed. A path P is alternating if its edges alternate between M and $E \setminus M$. If the first and last vertex of P are exposed, we call this an augmenting alternating path.

Alternating Tree

An alternating tree is a tree that has an exposed vertex as its root, and a condition that every path from the root is alternating.

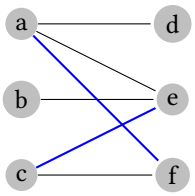

Figure 11. Matching for some subgraph of B where a, c, e, and f are matched. We call nodes b and d exposed.

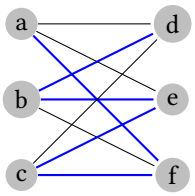

Figure 12. Example alternating path in matrix B is shown beginning at node a and ending at node d.

Algorithm

Now that we have defined our notation, we can describe how to implement the $O(n^3)$ algorithm.

Initialization

As an initial labeling, we set

$$\forall u \in U, l(u) = \max_{(u,v) \in E} (w(u, v)) \, ,$$

and

$$\forall v \in V, l(v) = 0 \, .$$

We start with an empty matching, M. We perform the the following steps in order to achieve an optimal matching for B.

Step 1. Given our initial matching M and the initial vertex labeling for U and V, there exists an exposed $u \in U$. This u will be the root of our alternating tree.

Step 2. Check if M is perfect, this is our stopping condition and determines if a solution exists. If not, set $S = \{u\}$, $T = \{\}$. If $J_{B_I}(S) = T$ then go to step 3, otherwise we go to step 4.

Step 3. In this step, we update the labeling in the following way.

$$\Delta = \min_{u \in S, v \in V \setminus T} (l(u) + l(v) - w(u, v))$$

$$l(x) = \begin{cases} l(x) - \Delta & x \in S \\ l(x) + \Delta & x \in T \\ l(x) & x \notin S, T \end{cases}$$

Step 4. Search if there exists $v \in T \setminus J_{G_I}(S)$. If v is matched in M with another vertex x, add x to set S, and add v to set T and move to step 3. If v is exposed, we augment a path from u to v and go to step 2.

The recursive $O(n^3)$ algorithm halts when a perfect matching M is found in step 2 of the algorithm. The algorithm will always halt given a bipartite graph $B = (V, U, E)$ where $|V| = |U|$ [5].

Parallel Implementation

The serial implementation of the Hungarian algorithm is a non-trivial endeavor. Our first attempt at serial implementation was the $O(n^4)$ algorithm due to its simplicity (in comparison to the $O(n^3)$ implementation) and the main computational bottleneck being the 0-graph bipartite-graph matching step being largely parallelizable when implementing the push-relabel algorithm for maximum flow [4]. However, we ended up implementing and parallelizing the $O(n^3)$ algorithm due to the promise of much larger gains in execution time reductions as n becomes very large.

We selectively-parallelized our serial implementation of the $O(n^3)$ Hungarian algorithm in an attempt to achieve speedup over the serial version. For our implementation of the $O(n^3)$ Hungarian algorithm method we chose to parallelize wherever we found

1. Code that performed redundant calculations up to n times
2. For loops that performed searches and variable updates in our bipartite graph B.

The serial implementation of the code is split up into 2 main functions. The first function initializes the labeling for U and V. This is a straightforward endeavor through a simple row-wise parallelization. The second function is the *augment*() function. This is an encapsulation of steps 2-4 and therefore can benefit the most from parallelization. *augment*() is an iterative function that is implemented with recursion. It performs a breadth-first search in order to find augmenting paths during the alternating-tree search. We did not implement a parallel-version of breadth-first search to solve this problem, but it has been shown to successfully achieve speedup among other optimizations [1]. We also avoid fully parallelizing the *augment*() function. This posed many difficulties as parallelizing a recursive function is a non-trivial task and may require dynamic parallel programs.

The parallel optimizations we applied were a search for and parallelization of repetitive and expensive operations with independent variables within *augment*(). We found that updating the labels U and V happen regularly, making it a good candidate. Augmenting the alternating tree is another expensive and recurring process. Parallelizing these respective functions has shown to successfully achieve speedup.

With this approach, *augment*() performs computation on both the Host and the Device. This required a large number of memory transfers from Host to Device and vice versa which became a major cause of slowdown as the input size grew. To combat this, we modify the code to minimize memory transfer. We found that the iterative augmentation of our tree did not require memory transfer on every iteration, but instead, a few memory transfers for each *augment*() call. Updating labels in contrasts requires more constant memory transfers; however, we found that distinct subsets of the input for this function are only modified in either the host or device respectively, requiring only one-way memory transfers. Memory transfer optimization is the biggest contributor of our overall speed-up.

In addition to improving memory transfers, we also flatten all 2-dimensional arrays in the serial implementation to a 1-dimensional arrays in the parallel implementation. Using simple index math, we greatly trivialize parts of our parallel code.

3 Methodology

Test cases were conducted for square $n \times n$ bipartite graphs of size $n = 10$ to $n = 20000$, with varying thread and memory allocation on the GPU.

For all test cases we generate adjacency matrices for all test case values of n, and known random seeds for cost score generation in order to facilitate reproducibility.

During each serial and parallel test case, the algorithm execution time was recorded using the C++ `clock` method. File IO time was ignored for test cases. Test cases had execution time and outputted assignment score evaluated.

The serial implementation of the Hungarian algorithm serves as the benchmark to compare our parallel optimization results. Benchmarks compared against the serial implementation are

1. Parallel execution time
2. Validation of output assignment scores.

Alongside our own serial implementation of the Hungarian algorithm, we utilize the existing R library `clue` to validate our algorithm output. This library contains the `solve_LSAP` function which computes the assignment problem using the Hungarian algorithm given an $n \times n$ bipartite adjacency cost matrix.

For all generated inputs, both our serial C++ and our parallel C++/CUDA implementation received the same outputted assignment scores as `clue` for similar inputs to the maximum and minimum task assignment problem.

All of our timing code is run and tested on the Mines Isengard system operating on a Linux OS. Isengard is the Mines supercomputer available to all campus users. The serial code is compiled using the *gcc* compiler and the -O3 compiler optimization flag. The parallel code is compiled using *nvcc* CUDA compiler under the `sm_35` architecture.

The CUDA architecture allowed us to test our algorithm with a range of CUDA GPU block grid sizes and active threads per block during execution. Active threads per block is defined by the user, and the necessary number of blocks is computed using n, the number of tasks, and threads per block. The equation shown below is how the number of blocks is determined.

$$\text{BLOCKS} = sizeof(\text{int}) * n/\text{THREADS\_PER\_BLOCK}$$

Code and test case scripts can be found at the git repository https://github.com/adzarrini/Hungarian.git.

4 Experimental Results

Our parallel implementation achieved speedup of up to 10× for large test cases.

Serial and parallel algorithm execution times can be seen for a subset of our threads per block test cases in figures 13 and 14. The serial implementation is shown to have much larger execution times than the parallel test cases.

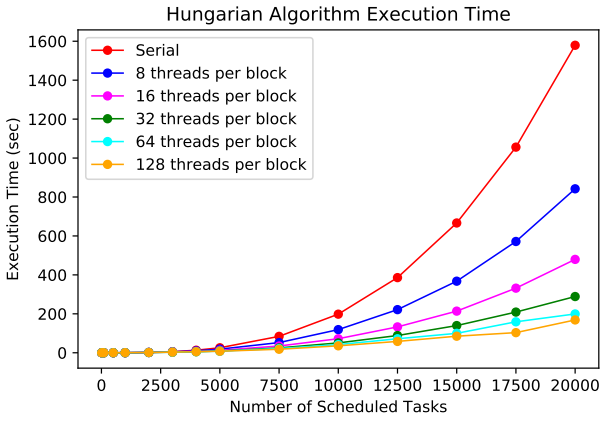

Figure 13. Execution time for the serial and multiple parallel Hungarian algorithm implementations.

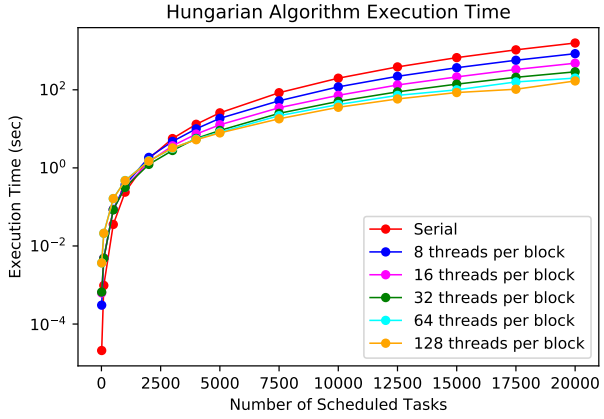

Figure 14. Execution time for the serial and multiple parallel Hungarian algorithm implementations.

Our parallel algorithm did not produce speedup for test cases where the number of tasks to schedule were small ($n < 2000$). Figure 15 shows how the serial implementation dominates as the fastest executing algorithm for smaller test cases. We see that for test cases larger than 2000, all cases saw execution times lower than that of the serial implementation. Larger values of test case inputs show more favorable response to parallelization.

We calculate parallel speedup, S_P, by computing

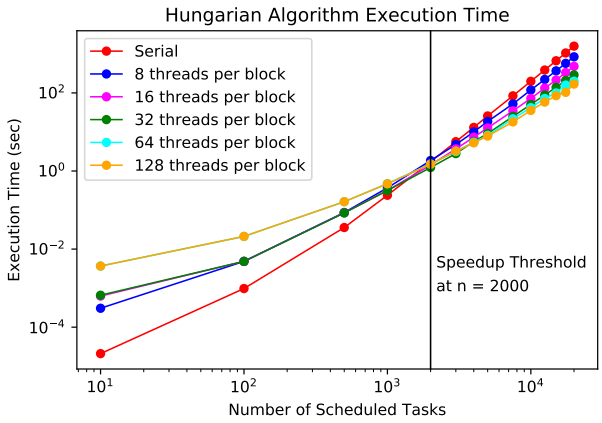

Figure 15. Parallel speedup was seen for task sizes of 2000 and greater.

$$S_P = \frac{\text{execution time old}}{\text{execution time new}} . \tag{6}$$

Parallel speedup for all test cases can be seen according to a changing thread count per block metric in figure 16. We observe a increasing parallel speedup for most test cases up until a thread count per block of 256. Thread counts per block after 256 tend to stall at an asymptotic speedup value. The largest test cases saw the largest speedup, which is over 9× in a comparison with the serial implementation.

We notice that increasing the number of threads per block does not increase performance at a rate similar to the increase in threads per block, as seen in figure 16. The decreasing marginal benefit of the threads per block size parameter is an example of Amdahl's law at work. Even though the larger graphs see a larger benefit from parallelization, only a portion of our program is parallelizable and we see the rest of the algorithm's execution time is left untouched by the parallel optimization.

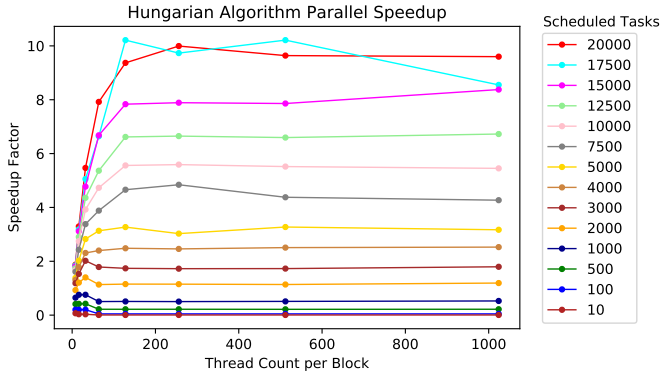

Figure 16. Speedup achieved on each test input according to number of threads per block used in implementation.

5 Discussion and Conclusion

Our results show that the $O(n^3)$ version of the Hungarian algorithm can benefit from parallelization and see execution times increased by up to 10×.

Smaller cases applied to our parallel solution to the assignment problem do not perform as well as larger test cases. This is an expected phenomenon for our evaluation, as the input size needs to see a large enough benefit from parallelization to offset the added execution time from thread initialization and memory transfers between the host and device. The serial implementation is most useful for small scheduling tasks due to many less computations and higher relative cost for memory transfer penalties.

For large assignment inputs, our parallel implementation of the Hungarian algorithm has extremely favorable execution times compared to the serial CPU version of the algorithm. This is due to the fact that the computational complexity of the problem grows at an exponential rate with increasing n, as each 2× increase of n results in a total cost parameter increase of $4 \times n$. This increase in cost parameter complexity increases our time complexity from $O(n^3)$ to $O((2 * n)^3) = 8 * O(n^3)$ for an input that is just twice as large as the previous input size. This complexity increase results in very large increases of execution time for increasing sizes of task assignments. Due to this, the larger the task assignment for a problem, the greater the benefit from parallelization.

Even though the parallelization we applied to the algorithm was sparse in consideration to the entirety of the process, we were able to increase the efficiency of the execution of the algorithm by

1. Pinpointing execution bottle-necks in the augment().
2. Minimal memory transfers between the host code and the device were performed.

The iterative nature of the $O(n^3)$ algorithm made parallelization much more difficult than we initially had hoped. The main *augment*() function was serially-implemented as a recursive function that did not favor parallelization due to the requirement of shared data between recursive calls.

The main problem with our $O(n^3)$ approach was that we designed the serial method without the intent of parallelizing the function later on. The code is made to run efficiently on a CPU. While the serial implementation of the $O(n^3)$ Hungarian algorithm was implemented correctly, it was not optimized for transferring into a parallelized state. Further work could be done towards editing the approach we took towards solving the linear-assignment problem that is more conducive to solving the problem where a parallel breadth-first search could be implemented during the *augment*() function [1].

We initially wanted to evaluate parallelization of the $O(n^4)$ Hungarian algorithm, but difficulties in the serial implementation lead us to decide to focus our efforts in the $O(n^3)$ implementation due to the difference in time complexity and not a large difference in implementation difficulty. All of our research into implementing the $O(n^4)$ algorithm helped in our understanding of the more complex $O(n^3)$ algorithm, and we hope this is the case for the reader as well.

The $O(n^3)$ algorithm was our focus of optimization over the $O(n^4)$ implementation of the Hungarian algorithm due to much better time complexity in the $O(n^3)$ version. In the future, it would be interesting to compare the performance of a serial and parallel implementation of the $O(n^4)$ algorithm. We can say with certainty that with larger n, the serial implementation would perform much slower than the $O(n^3)$ implementation. However, because the $O(n^4)$ implementation is more readily parallelizable, we would expect to see cases in which it performs better than the parallel implementation of the $O(n^3)$ algorithm.

Our method is not optimized to the fullest potential of the algorithm, but it is proof of life of the possibility of achieving decreased execution times by parallelizing algorithms that are not obvious candidates for parallel speedup, such as an iterative, recursive function as the Hungarian algorithm.

References

[1] K. Date and R. Nagi, "GPU-accelerated Hungarian algorithms for the Linear Assignment Problem." Elsevier, Parallel Computing. 57 (9): 52-57. 2016 September.

[2] R. Jonker; A. Volgenant, "A shortest augmenting path algorithm for dense and sparse linear assignment problems." Computing. 38 (4): 325–340. 1987 December.

[3] J. Munkres, "Algorithms for the Assignment and Transportation Problems," Journal of the Society for Industrial and Applied Mathematics, 5 (1): 32–38. 1957 March.

[4] R. Anderson, J. Setubal, "A Parallel Implementation of the Push–Relabel Algorithm for the Maximum Flow Problem," Journal of Parallel and Distributed Computing (1995).

[5] TopCoder. 2019. *Assignment Problem and Hungarian Algorithm.* https://www.topcoder.com/community/competitive-programming/tutorials/assignment-problem-and-hungarian-algorithm/