

© 2018 Shardul Natu

GPU-BASED LAGRANGIAN HEURISTIC FOR MULTIDIMENSIONAL
ASSIGNMENT PROBLEMS WITH DECOMPOSABLE COSTS

BY

SHARDUL NATU

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Industrial Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2018

Urbana, Illinois

Adviser:

Professor Rakesh Nagi

ABSTRACT

Multidimensional assignment problem (MAP) is one of the many formulations of data association problem which categorizes data based on various data sources. A higher number of data sources ensures an accurate categorization of data. But it also leads to a significant increase in the amount of data, consequently increasing the computation time where quick results are sought. In this work, we used Lagrangian relaxation technique to solve the MAPs with decomposable costs. But the major contribution was an efficient parallelization of this algorithm on a graphics processing unit (GPU) based programming architecture. Bigger problems with larger data sets were solved by using multiple processors with each having a GPU of its own. This not only handled the data by distributing it among the processors, but also increased the amount of parallelization to give us good iteration times. Problems with 796 million cost variables were solved on varying number of processors between 1 and 64, with significantly fast iteration times. Owing to the good scalability of the developed parallel solver, we successfully solved problems with 31 billion cost variables on processors ranging from 64 to 128 in good amount of time.

Keywords: Multidimensional assignment problem (MAP); Linear assignment problem (LAP); Graphics processing unit (GPU); Lagrangian relaxation.

To my friends, family and teachers, for their love, support and patience.

ACKNOWLEDGMENTS

I would like to start by thanking my thesis adviser, **Dr. Rakesh Nagi**, who has not only helped me with his advise, immense academic knowledge and timely guidance, but also with his support whenever needed. It goes without saying that Dr. Nagi has been a motivation for the quality of work he promotes.

I would also like to thank my mentor and friend, **Dr. Ketan Date**, for his help with numerous challenges I faced with the softwares and tools I used. He also helped me with the concepts of theoretical knowledge and their application that were central to this work.

I would also like to acknowledge **University of Illinois at Urbana-Champaign** and **National Center for Supercomputing Applications** for their super-computing facility, **Blue Waters**, on which the experiments were conducted.

Finally I would like to thank my brother, **Mehul**, and my friends **Nikita** and **Abhishek** for their continued support through times, good and bad.

TABLE OF CONTENTS

CHAPTER 1	INTRODUCTION	1
CHAPTER 2	SEQUENTIAL LAGRANGIAN HEURISTIC	6
2.1	Lagrangian Dual Problem	6
2.2	Primal-dual Algorithm	7
2.3	Repair Heuristic	9
CHAPTER 3	PARALLEL LAGRANGIAN HEURISTIC	11
3.1	Initialization	11
3.2	LAP Solution	12
3.3	Updating Lagrangian multipliers and LAP costs	15
3.4	Stopping Criteria	18
CHAPTER 4	RESULTS	20
4.1	Experiments with Single PE	21
4.2	Experiments with Multiple PEs	22
4.3	Discussion	25
CHAPTER 5	SUMMARY AND CONCLUSIONS	29
CHAPTER 6	REFERENCES	31

CHAPTER 1

INTRODUCTION

Multi-dimensional Assignment Problem (MAP) is a generalization of the Linear Assignment Problem (LAP) to $K \geq 3$ dimensions, and therefore, it is nothing but minimum-cost K -partite matching problem. To be more specific, if we have a K -partite graph with N vertices in each partition (or dimension), then the problem is to find N vertex disjoint subsets of size K (one vertex per dimension), such that the sum of the costs of the subsets is minimized. Different variants of MAP can be conceived, for different strategies of defining the subset costs in the objective function. Various mathematical formulations of MAP are thoroughly discussed by Poore and Rijavec (1993) [3], Poore (1994) [4], Poore and Robertson (1997) [5], Bandelt et al. (2004) [12], Walteros et al. (2014) [22]. The Multi-dimensional Assignment Problem with Decomposable Costs (MDADC) [2, 7] is one such variant in which each edge connecting a pair of nodes from two different partitions has a certain weight, and the goal is to minimize the sum of the weights of the pairwise edges included in the subsets (“clique” cost).

Let us consider the following Mixed Integer Linear Programming (MILP) formulation for the MDADC. Note that the indices $i, j, k \in \{1, \dots, N\}$, and indices $p, q, r \in \{1, \dots, K\}$.

$$\text{MDADC: } \min \sum_i \sum_j \sum_p \sum_{q>p} C_{ij}^{pq} x_{ij}^{pq}, \quad (1.1)$$

$$\text{s.t. } \sum_j x_{ij}^{pq} = 1, \quad \forall (i, p, q), : p < q; \quad (1.2)$$

$$\sum_i x_{ij}^{pq} = 1, \quad \forall (j, p, q), : p < q; \quad (1.3)$$

$$\left. \begin{aligned} x_{ij}^{pq} + x_{jk}^{qr} - x_{ik}^{pr} - 1 &\leq 0, \\ x_{ij}^{pq} + x_{ik}^{pr} - x_{jk}^{qr} - 1 &\leq 0, \\ x_{ik}^{pr} + x_{jk}^{qr} - x_{ij}^{pq} - 1 &\leq 0, \end{aligned} \right\} \quad \forall (i, j, k, p, q, r) : p < q < r; \quad (1.4)$$

$$x_{ij}^{pq} \in \{0, 1\}, \quad \forall (i, j, p, q) : p < q. \quad (1.5)$$

The decision variable $x_{ij}^{pq} = 1$, if vertex i from partition p is assigned to vertex j from partition q , and 0 otherwise. Constraints (1.2) and (1.3) enforce that a vertex from one of the partitions should be assigned to exactly one vertex from any other partition. Constraints (1.4) enforce the between-partition transitivity for the assigned vertices. It simply means that if vertex i in partition p is assigned to vertex j from partition q ($x_{ij}^{pq} = 1$), and vertex j is assigned to vertex k from partition r ($x_{jk}^{qr} = 1$), then either: (1) Vertex i should be assigned to vertex k by setting $x_{ik}^{pr} = 1$ (if it is beneficial to the objective function), or (2) One or both of the other two assignments should be made zero. C_{ij}^{pq} is the cost of assigning vertex i from partition p to vertex j from partition q .

MDADC is routinely used to model the Data Association problem, especially when the relationships of vertices within a particular graph are not important. Data association [13, 17, 18, 19] is a fundamental problem in data science applications involving multiple data sources. Data gathered by these sources may be in different formats. The goal of data association is to merge these data into a cumulative evidence that can be used for sense-making tasks or to obtain information about the current state of the real world. A generalization of this problem is called Graph Association problem in which the topological information is also utilized [13].

1. In **information fusion**, the role of data association is to identify and merge common references from the “soft data,” in the form of structured or unstructured natural language; and “hard data” gathered by cameras, LIDAR, acoustic sensors, etc., in the form of video and image files. These data are converted into graphs containing entity references and their relationships, using data processing algorithms. Pairwise sim-

ilarity scores are generated between vertices and edges, which are used in the objective function of the data association formulation. Based on these scores, a data association algorithm appropriately merges the entity references so as to maximize the total similarity score.

2. In **multi-target tracking** applications [14], multiple sensors scan the region of interest at different points of time and produce surveillance data. The sensors are typically physical sensors like cameras, RADAR, LIDAR, etc. These data are processed using detection and association algorithms to create a temporal track, which can be used for estimating the current state of the moving targets, within the region of interest.

MDADC is an NP-Hard problem, since it is a generalization of the NP-Hard 3-Dimensional Assignment Problem (3-DAP). Various approaches have been proposed in the literature, to solve MDADC optimally and sub-optimally. A large number of these approaches involve meta-heuristic local search procedures [6, 12]. These procedures are reported to quickly produce solutions of acceptable quality. However, a major drawback of these procedures is that they do not provide any guarantee on the optimality of the solution (in the form of an optimality gap), as a result of which, it is not possible to characterize the association solution.

Recently, Vogiatzis et al. (2014) [23] proposed algorithms based on decomposition techniques for solving the axial MDADC. Horizontal decomposition yields disjoint MDADC subproblems with K dimensions and fewer than N vertices per dimension. Solving these subproblems provides an upper bound on the original problem. Vertical decomposition yields disjoint subproblems with fewer than K dimensions and N vertices per dimension. Solving these subproblems provides a lower bound on the original problem. The authors tested both these decomposition approaches, in conjunction with a “repair heuristic,” to efficiently solve several small and medium-sized MDADC instances.

Other ways of solving MDADC include the systematic tree search approaches such as the branch-and-bound (B&B), in which a lower bound can be obtained at each node by solving a Linear Programming (LP) relaxation of MDADC. The challenge with this approach lies the large number of decision variables and constraints in MDADC, which makes it difficult to solve the resulting LP relaxation using primal methods. However, the transitivity con-

straints can be relaxed and added to the objective function using Lagrange multipliers, and the resulting problem can be solved using one of the constructive dual techniques (see Chapter 2). There are two main advantages of using constructive dual techniques: (1) They provide a lower bound on the MDADC, which can be used in the B&B scheme; and (2) The partial (infeasible) solution obtained during each iteration may be repaired to obtain a feasible solution with a characterization in terms of the optimality gap. Such Lagrangian relaxation based approaches were studied by Poore and Rijavec (1993) [3] and Poore and Robertson (1997) [5], for the traditional formulation for MDADC, which uses multi-index variables.

Tauer and Nagi (2013) [9] studied the Lagrangian relaxation of the problem by relaxing the transitivity constraints (1.4) using Lagrange multipliers. The relaxed problem was decomposed into a large number of polynomially solvable LAPs, which were solved independently on multiple processors. The Lagrangian dual problem was solved using parallel sub-gradient optimization method in which the dual objective function may be improved in each iteration by adjusting the dual multipliers corresponding to the infeasible constraints. The authors used Hadoop Map-Reduce as the parallel programming architecture, and they showed that their parallel implementation exhibits decent scaling behavior on up to 96 processors. While this approach is certainly suitable for solving large problems, the Hadoop Map-Reduce is not efficient in frequent seeking of “small” data (such as node adjacencies and Lagrange multiplier values), leading to longer execution times. These execution times could be improved drastically by migrating to a different parallel programming architecture.

With the advent of Graphic Processing Units (GPUs), the amount of parallelism that can be achieved has increased drastically. Although GPUs have been used mainly to enhance graphical applications, they have been extremely useful in academic and industry-scale computations since they became programmable through parallel programming interfaces like Compute Unified Device Architecture (CUDA) introduced by NVIDIA Inc. With hundreds of Streaming Multiprocessors (SMs) and their effectiveness in vector and matrix operations, GPUs are very well suited for parallelizing the Lagrangian dual procedures. Additionally, GPU-based computing has been used successfully to speed up the Linear Assignment Solvers [8, 20, 10], which are important in solving MDADC. To this end, the main contribu-

tion of this work is a GPU-based algorithm for obtaining tight lower bounds on MDADC, which couples the GPU-accelerated Hungarian Algorithm of Date and Nagi (2016) [10] and a novel GPU-accelerated Lagrangian subgradient search scheme. We show that this algorithm achieves over 100x speedup when compared to the Hadoop Map-Reduce scheme of Tauer and Nagi (2013) [9]. These execution times could be improved drastically by migrating to a different parallel programming architecture.

This thesis is organized as follows. In Chapter 2, we describe the sequential Lagrangian heuristic and opportunities for parallelization on a GPU. In Chapter 3 we describe the various stages of our parallel algorithm, and their implementation on multi-GPU architecture. In Chapter 4, we present the experimental results on randomly generated problem instances. Finally in Chapter 5, we provide a summary and future research directions.

CHAPTER 2

SEQUENTIAL LAGRANGIAN HEURISTIC

2.1 Lagrangian Dual Problem

Let us consider the MDADC formulation (1.1)-(1.5). The transitivity constraints (1.4) can be relaxed and added to the objective function using Lagrange multipliers $\boldsymbol{\theta} = \langle \theta_{(ij)k}^{(pq)r} \rangle$, where, $\theta_{(ij)k}^{(pq)r}$ is the dual variable corresponding to constraint $x_{ij}^{pq} + x_{jk}^{qr} - x_{ik}^{pr} - 1 \leq 0$ (the indices in the parentheses denote the leading variable). This leads to the following Lagrangian relaxation LR($\boldsymbol{\theta}$):

$$\text{LR}(\boldsymbol{\theta}): \min \sum_i \sum_j \sum_p \sum_{q>p} ((C_{ij}^{pq} + \Theta_{ij}^{pq}) x_{ij}^{pq} - \Omega_{ij}^{pq}); \quad \text{s.t. (1.2), (1.3), (1.5).} \quad (2.1)$$

Here,

$$\begin{aligned} \Theta_{ij}^{pq} = \sum_k \left(\sum_r \left(\theta_{(ij)k}^{(pq)r} \right) + \sum_{r<p,q} \left(\theta_{(ki)j}^{(rp)q} - \theta_{(kj)i}^{(rq)p} \right) + \sum_{r>p,r<q} \left(-\theta_{(ik)j}^{(pr)q} + \theta_{(kj)i}^{(rq)p} \right) \right. \\ \left. + \sum_{r>p,q} \left(-\theta_{(jk)i}^{(qr)p} + \theta_{(ik)j}^{(pr)q} \right) \right). \end{aligned} \quad (2.2)$$

$$\begin{aligned} \Omega_{ij}^{pq} = \sum_k \left(\sum_r \left(\theta_{(ij)k}^{(pq)r} \right) + \sum_{r<p,q} \left(\theta_{(ki)j}^{(rp)q} + \theta_{(kj)i}^{(rq)p} \right) + \sum_{r>p,r<q} \left(\theta_{(ik)j}^{(pr)q} + \theta_{(kj)i}^{(rq)p} \right) \right. \\ \left. + \sum_{r>p,q} \left(\theta_{(jk)i}^{(qr)p} + \theta_{(ik)j}^{(pr)q} \right) \right). \end{aligned} \quad (2.3)$$

If $\nu(\cdot)$ represents the objective function of a problem, then for any $\boldsymbol{\theta} \geq \mathbf{0}$,

$\nu(\text{LR}(\boldsymbol{\theta}))$ provides a lower bound on $\nu(\text{MDADC})$ ($\nu(\text{LR}(\boldsymbol{\theta})) \leq \nu(\text{MDADC})$). To find the best possible lower bound, we need to solve the Lagrangian dual problem $\text{LD}(\boldsymbol{\theta})$: $\max_{\boldsymbol{\theta} \geq 0} \nu(\text{LR}(\boldsymbol{\theta}))$. Hence, the primary goal is to systematically search for the Lagrange multipliers which maximize the objective function value of the Lagrangian dual problem.

2.2 Primal-dual Algorithm

The Lagrangian dual problem $\text{LD}(\boldsymbol{\theta})$ of MDADC is solved using an iterative primal-dual strategy which has the following main stages:

2.2.1 LAP Solution

In the primal-dual algorithm, the first step is to choose initial values for the dual variables $\hat{\boldsymbol{\theta}}$ and solve $\text{LR}(\hat{\boldsymbol{\theta}})$. The salient feature of $\text{LR}(\hat{\boldsymbol{\theta}})$ is that it can be decomposed into polynomial-time solvable LAPs constructed for each pair of graphs (or partitions). To be specific, for each (p, q) , with $p < q$, we need to solve the problems:

$$\text{SP}_{pq}(\hat{\boldsymbol{\theta}}) : \min \sum_i \sum_j \left(C_{ij}^{pq} + \hat{\Theta}_{ij}^{pq} \right) x_{ij}^{pq}, \quad (2.4)$$

$$\text{s.t.} \quad \sum_j x_{ij}^{pq} = 1, \quad \forall i; \quad (2.5)$$

$$\sum_i x_{ij}^{pq} = 1, \quad \forall j; \quad (2.6)$$

$$x_{ij}^{pq} \in \{0, 1\}, \quad \forall (i, j). \quad (2.7)$$

The overall objective function value is simply the sum of the objective function values of the individual LAPs minus the sum of all the dual multipliers. This can be represented as follows:

$$\nu(\text{LR}(\hat{\boldsymbol{\theta}})) = \sum_p \sum_{q>p} \left(\nu(\text{SP}_{pq}(\hat{\boldsymbol{\theta}})) - \sum_i \sum_j \hat{\Omega}_{ij}^{pq} \right). \quad (2.8)$$

The values of Θ and Ω in the above equations are calculated according to Equations (2.2) and (2.3).

2.2.2 Lagrangian Multiplier Update

We now explain the multiplier update procedure, in which the dual objective function is improved by adjusting the dual multipliers corresponding to the infeasible constraints, which provide a valid subgradient to the objective function $\nu(\text{LR}(\hat{\theta}))$.

The multiplier update procedure is very similar to the standard gradient ascent procedure, where we advance along the (sub)gradients of the objective function (using a step size λ), until we reach some solution that is no longer improving. At that point, we calculate the new (sub)gradients and continue. The disadvantage of using subgradients instead of the gradient is that it is difficult to characterize an accurate step-size which is valid for all the active subgradients. Therefore, taking an arbitrary step along the subgradients might worsen the objective function from time to time. However, for specific step-size rules, it is proved that the procedure converges to the optimal solution asymptotically. One such rule is to decrease λ by some percentage after a certain number of iterations, so as to have $\lambda^k \rightarrow 0$ as $k \rightarrow \infty$, and $\sum_k \lambda^k \rightarrow \infty$.

2.2.3 Subgradient Optimization

The pseudocode for solving the Lagrangian dual problem $\text{LD}(\theta)$ is shown in Algorithm 1. Note that $\phi = \langle \phi_{(ij)k}^{(pq)r} \rangle$, where, $\phi_{(ij)k}^{(pq)r} = x_{ij}^{pq} + x_{ik}^{pr} - x_{jk}^{qr} - 1$, i.e., the constraint vector for constraint set (1.4). This algorithm iterates between the LAP solution stage and the Lagrangian subgradient search stage to obtain a tight lower bound on the MDADC. It is important to note that the constraint set of $\text{LR}(\theta)$ has integrality property. Therefore, the optimal objective function value $\nu^*(\text{LD}(\theta))$ will be no better than that obtained by solving the LP relaxation of MDADC. However, as mentioned earlier, it is not possible to obtain the LP relaxation bound for large instances of MDADC, using standard state-of-the-art solvers.

2.3 Repair Heuristic

Lagrangian subgradient optimization algorithm provides a valid lower bound on the MDADC. However, to compute the optimality gap, an upper bound (feasible solution) to MDADC is also required. The solution to $\text{LR}(\boldsymbol{\theta})$ satisfies the assignment constraints (1.2)-(1.3), but, it may not satisfy the transitivity constraints (1.4). However, we can devise a greedy heuristic that constructs a feasible solution to MDADC, by using hints from the solution of $\text{LR}(\boldsymbol{\theta})$. This heuristic was also used by Tauer and Nagi (2013) [9].

The construction heuristic aims at organizing $K \times N$ vertices into N cliques of K vertices each. We start with N cliques $\{X_1, \dots, X_N\}$ and populate them with vertices from any one of the partitions (one vertex per clique). Then, vertices from the remaining partitions are picked randomly and assigned to those cliques, using the following rule. A vertex i from partition p is assigned to a clique X_m such that: (1) X_m does not already contain another vertex from partition p ; and (2) X_m has the maximum number of vertices assigned to vertex i , according to the solution of $\text{LR}(\boldsymbol{\theta}^k)$. In case of a tie, the vertex i is placed in the clique with the smallest number of vertices.

Algorithm 1: Lagrangian subgradient optimization.

1. Initialize $k \leftarrow 0$, $\boldsymbol{\theta}^k \leftarrow \mathbf{0}$ and $\text{LB} \leftarrow -\infty$, and $\text{GAP} \leftarrow \infty$.
 2. Initialize step-size λ^k and multiplier $0 < \alpha \leq 1$.
 3. LAP solution:
 - (a) Solve $\binom{K}{2}$ LAPs of size $N \times N$ and obtain $\nu(\text{SP}_{pq}(\boldsymbol{\theta}^k))$
 $\forall p, q : p < q$.
 - (b) Set $\nu(\text{LR}(\boldsymbol{\theta}^k)) \leftarrow \left(\sum_p \sum_{q>p} \nu(\text{SP}_{pq}(\boldsymbol{\theta}^k)) - \Omega^k \right)$.
 - (c) If $\text{LB} < \nu(\text{LR}(\boldsymbol{\theta}^k))$, update $\text{LB} \leftarrow \nu(\text{LR}(\boldsymbol{\theta}^k))$.
 - (d) Update $\text{GAP} \leftarrow \frac{\text{UB} - \text{LB}}{\text{LB}}$. Stop if $\text{GAP} < \text{MIN_GAP}$.
 4. Feasibility check:
 - (a) Evaluate constraint vector $\boldsymbol{\phi}^k$ using solution \mathbf{x}^k .
 - (b) Stop if $\boldsymbol{\phi}^k \leq \mathbf{0}$.
 5. Lagrangian dual update:
 - (a) Update dual multipliers $\boldsymbol{\theta}^{k+1} \leftarrow \max\{0, \boldsymbol{\theta}^k + \lambda \cdot \boldsymbol{\phi}^k\}$.
 - (b) Update cost coefficients C_{ij}^{pq} using $\boldsymbol{\theta}^{k+1}$.
 - (c) Update step-size $\lambda^{k+1} \leftarrow \alpha \cdot \lambda^k$.
 6. Update $k \leftarrow k + 1$. Stop if $k > \text{ITN_LIM}$. Else, return to Step 3.
-

CHAPTER 3

PARALLEL LAGRANGIAN HEURISTIC

The sequential Lagrangian heuristic discussed in the previous chapter can be used for obtaining both upper and lower bounds on MDADC and possibly solving the problem to optimality. However, the main disadvantage of the sequential approach is that, for solving an MDADC of size (K, N) , we need to solve $O(K^2)$ LAPs of size $N \times N$ each, and update $O(K^3 N^3)$ Lagrange multipliers. An important observation about Lagrangian heuristic is that the $O(K^2)$ LAPs can be solved independently of each other and similarly, the $O(K^3 N^3)$ Lagrangian multipliers can be updated independently of each other. Therefore, with the help of an appropriate parallel programming architecture, it is possible to achieve significant speedup over the sequential algorithm. In this chapter, we provide the details of our parallel implementation.

3.1 Initialization

Our parallel programming architecture comprises of multiple CPU-GPU pairs in a computational grid. We will refer to each pair as a Processing Element (PE). Communication between the different CPUs is accomplished using the Message Passing Interface (MPI), which uses the Local Area Network to transfer data between the CPUs. The program is initialized with P number of MPI processes, equal to the number of PEs in the grid. It is assumed that one MPI process gets allocated to exactly one CPU. The cost matrices for the LAPs are split evenly across all the PEs in the grid, such that, each PE owns $\left\lceil \frac{K(K-1)}{2P} \right\rceil$ number of LAP matrices. The constraints and Lagrange multipliers are also split evenly across all the PEs in the grid, in that, each PE is responsible for evaluating the constraints formed by the variables in the associated subproblems, and for updating the corresponding Lagrange

multipliers. The main advantage of this scheme is that it increases the time spent by the PEs doing parallel work and reduces the communication time, which in turn improves the parallel speedup.

3.2 LAP Solution

One of the most important aspects of this work is to handle the huge amounts of data. Although the cost values are easier represented as a set of square matrices stacked together making them three dimensional, computationally it is beneficial to store them in a single array. This ensures that there is a contiguous memory allocation resulting in faster access times. Hence, the cost coefficients of MDADC are stored as a double floating point array, which is then split across the different PEs, as outlined in Section 3.1. The sub-array owned by a particular PE represents the cost coefficients of a collection of LAPs stored in row major order, as illustrated in Figure 3.1. A larger K may lead to a situation where a PE is given more problems than it can store in the GPU memory. To avoid this, we predetermine the number of subproblems a GPU can handle at a time and solve the subproblems in batches. After solving all the subproblems, the optimal row assignments of all the subproblems are communicated among the PEs using `MPI_Bcast` primitive. As transitivity constraints for any given subproblem depend on all the assignments of all the vertices present in the subproblem, it is important for each processor to have all the row assignments in order to compute the violations for the constraints allocated to it.

3.2.1 LAP Tiling

To solve LAP subproblems, we have leveraged the GPU-accelerated Hungarian algorithm of Date and Nagi (2016) [10]. The accelerated Hungarian algorithm is efficient in solving large LAPs, however, it is not as efficient in solving large number of small LAPs, due to CUDA kernel invocation overhead. For this reason, all the LAPs owned by a particular PE are combined or *tilled* into a single large LAP. There are a couple of ways the LAPs can be tiled. The simplest scheme is to place the cost matrices of individual LAPs in the diagonal of a larger matrix, with rest of the cost values set to

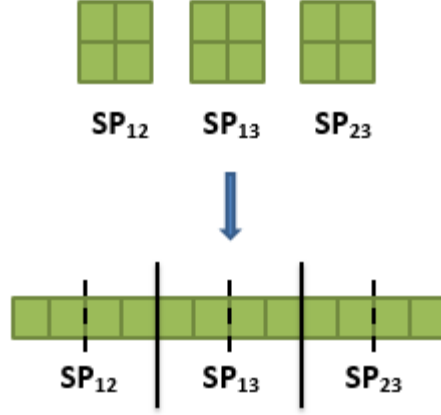


Figure 3.1: Schema for storing cost values for a problem with $K = 3$, $N = 2$

infinity (as shown in Figure 3.2); and then solve the resulting super-LAP. However, this scheme requires $O(K^4 N^4 / P^2)$ storage, since each processor handles $O(K^2 / P)$ LAPs. We can dramatically reduce the memory space requirement by (logically) stacking the LAP matrices as shown in Figure 3.3, and mapping the row and column IDs of each LAP to that of the single tiled LAP. This scheme requires $O(K^2 N^2 / P)$ of memory space. This single tiled LAP is then solved using the accelerated Hungarian solver. We believe that this solution strategy can be a significant contribution to many applications that require solving multiple small LAPs on a GPU.

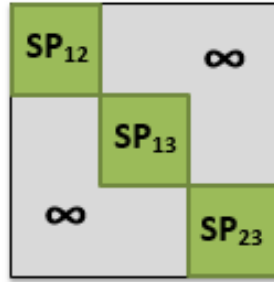


Figure 3.2: Diagonal tiling of multiple LAPs for $K = 3$

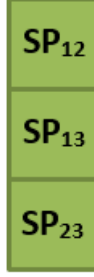


Figure 3.3: Stacked tiling of multiple LAPs for $K = 3$

3.2.2 Dynamic LAP Solver

In the Lagrangian subgradient optimization algorithm, violations of constraints lead to updates in the Lagrange multipliers and the cost coefficients of the LAP subproblems. For the problem instances studied in this paper, we found that the fraction of cost values that get modified from iteration to iteration remains below 20%. In such cases, the previous optimal assignments can be repaired to obtain new optimal assignments for the LAPs, rather than solving the LAPs from scratch. This dynamic Hungarian algorithm was introduced by Korsah et al. (2007) [21]. This algorithm works on the same principle as that of sensitivity analysis in the simplex algorithm. If the costs are modified, the dynamic algorithm first updates the dual multipliers and checks whether the primal assignments have zero dual reduced cost (a necessary condition for optimality). If this is true, then the algorithm terminates with the same primal solution and modified dual solution (both of which will have the same optimal objective value). If for any of the assignments, the dual reduced cost is non-zero, then that particular assignment is made zero and the Hungarian algorithm iterations are performed until optimality is achieved.

Within our GPU-accelerated framework, we implemented the dynamic Hungarian algorithm. To be able to use this dynamic LAP solver, we store the changes to the cost coefficients for each subproblem in a separate array (we will refer to these as “delta” costs). During the cost update step, only these delta cost values are updated by adding together the appropriate dual multipliers, while the original cost values are kept intact. At the beginning of each iteration, the delta costs are added to the original cost coefficients, and the dynamic Hungarian algorithm is invoked to obtain the new solution and the corresponding lower bound.

3.3 Updating Lagrangian multipliers and LAP costs

Once the optimal row assignments are found for $\text{LR}(\boldsymbol{\theta})$ and communicated to all the processors, next step is to update the Lagrange multipliers and the cost coefficients of the LAPs. This step is divided into the following three steps.

3.3.1 Constraint Enumeration

The goal of this step is to find the violations of transitivity constraints, which represent valid subgradients of the Lagrangian dual problem. The number of transitivity constraints increases dramatically with increasing K or N . For an LAP subproblem, the number of transitivity constraints equals $(K-2)N^3$, and since there are $\binom{K}{2}$ subproblems, the total of transitivity constraints equals $\frac{K(K-1)(K-2)N^3}{2}$, which is $O(K^3N^3)$, for $K \geq 3$. If we were to save the Lagrange multiplier values ($\boldsymbol{\theta}$) for all the transitivity constraint for a problem with $K = 100$ and $N = 100$, we would need to store 4.851×10^{11} values, which translates into over 3.5TB of memory. Fortunately, a large number of these constraints have non-zero slack, which means that the corresponding Lagrange multipliers must be zero. We can exploit this fact and save valuable memory space by storing only the non-zero Lagrange multipliers.

Table 3.1: Truth table for possible violation of

$$\phi_{(ij)k}^{(pq)r} \equiv x_{ij}^{pq} + x_{ik}^{pr} - x_{jk}^{qr} - 1 \leq 0$$

x_{ij}^{pq}	x_{ik}^{pr}	x_{jk}^{qr}	Violation
0	0	0	No
0	0	1	No
0	1	0	No
0	1	1	No
1	0	0	No
1	0	1	No
1	1	0	Yes
1	1	1	No

Since any given transitivity constraint depends on three assignments, we can determine the combination of assignments that would result in violation of this constraint. Table 3.1 illustrates a truth table for identifying whether a constraint is violated or not for different values of the variables. From this

information, we can narrow down our search for the violated constraints. The efficient constraint enumeration procedure is outlined in Algorithm (2). A unique identifier (ID) can be assigned to each constraint, calculated based on the variable IDs involved in that constraint (essentially a hash function). An inverse function can be applied to this unique ID, to recover the original variables (this is essential during the cost update step). If we number the vertices and dimensions starting from 0, the unique ID for constraint $\phi_{(ij)k}^{(pq)r}$ can be calculated using the expression: $N^3(K^2p + Kq + r) + N^2i + Nj + k$.

Each PE is responsible for enumerating constraints for its given set of subproblems. The constraint enumeration step is performed on the GPU using multiple threads, such that each thread is responsible for one row assignment of the LAP subproblems, which is treated as the leading variable in the constraint (please refer to Section 2.1). Starting from the leading variable, the thread constructs the complete constraint and asserts whether it is violated or not. For example, a thread that is assigned to the variable $x_{ij}^{pq} = 1$, deals with the constraint $\phi_{(ij)k}^{(pq)r} \equiv x_{ij}^{pq} + x_{jk}^{qr} - x_{ik}^{pr} - 1 \leq 0$, and checks whether the trailing variable values are: $x_{jk}^{qr} = 1$ and $x_{ik}^{pr} = 0$. If this is true, then the constraint $\phi_{(ij)k}^{(pq)r}$ is in fact violated, and so its ID is appended to the array of violated constraints. This array of violated constraint IDs is transferred to the CPU. This array is first sorted and then merged with the array of IDs from the previous iteration, with the following rule. If an ID appears in both old and new arrays, or only in the old array, its θ value from the previous iteration is stored; otherwise the new ID is stored with a zero θ value. These computations are performed on the CPU, and the step itself can be seen as a preprocessing step before we update the dual multipliers.

3.3.2 Lagrange Multiplier Update

After constructing the array of affected constraint IDs, the next step is to update the dual multipliers corresponding to these constraints. Since each PE enumerates the violations for its specific set of subproblems, it is responsible for updating the corresponding dual multipliers. The multiplier update step is performed on the GPUs, using multiple threads, such that each thread updates a single multiplier. The multiplier update procedure is outlined in Algorithm 3. If the number of violated constraints is fairly large, they cannot

Algorithm 2: Kernel for finding newly violated constraint IDs

```
1: Input: Row assignment array  $A$ 
2: Output: Array of IDs of violations  $I_{new}$ 
3:  $size = 0$ 
4: parallel foreach  $x \in \{1, \dots, |A_r|\} \forall r \in \{1, \dots, K\}$  do
5:   From  $A[x]$  get nodes assigned to each other,  $v_i^p, v_j^q$ 
6:   if  $r \neq p$  and  $r \neq q$  then
7:     for  $k \in \{1, \dots, N\}$  do
8:       if  $x_{ik}^{pr} = 1$  and  $x_{jk}^{qr} = 0$  then
9:         atomic:  $size \leftarrow size + 1$ 
10:         $I_{new}[size] \leftarrow N^3(K^2p + Kq + r) + N^2i + Nj + k$ 
11:       end
12:     end
13:   end
14: end
```

be processed on the GPU in a single iteration. In this case, the corresponding multipliers are updated in smaller batches.

Algorithm 3: Kernel for updating dual variables

```
1: Input: Assignment array  $A$ , array of IDs  $I$  and array of  $\theta$ s  $T$ 
2: Output: Updated array  $T$ 
3: parallel foreach  $i \in \{1, \dots, |I|\}$  do
4:   From  $I[i]$  and  $A$  get node assignments  $x_{ij}^{pq}, x_{ik}^{pr}$  and  $x_{jk}^{qr}$ 
5:    $T[i] \leftarrow \max\{0, T[i] - \lambda(1 + x_{jk}^{qr} - x_{ij}^{pq} - x_{ik}^{pr})\}$ 
6: end
```

3.3.3 LAP Cost Update

The final step of the Lagrangian subgradient optimization is updating the cost values of the LAPs, in which updated dual multipliers are added to the appropriate cost coefficients of the LAP subproblems. Recall that the dual multiplier values are not present on any single PE, but mutually dis-

joint subsets of multipliers are present on PEs on which they were calculated. Unfortunately, the dual multipliers present on a particular PE may be required for updating the cost coefficients on several other PEs, and it may be challenging to pre-compute these demand destinations. To achieve this, one approach is to gather the dual multiplier values on all the PEs, so that each PE has a copy of all the multipliers in its memory. This approach is not viable for large problems, since the PEs may not have enough memory space to store all the multipliers. To avoid this, we implemented an alternative approach in which each PE is selected as source, and it broadcasts its own subset of dual multipliers to the rest of the PEs. After receiving this batch of multipliers, all the PEs use the relevant multipliers from this batch to update the LAP cost coefficients in parallel. The process is repeated for all the PEs, one at a time. This approach saves significant amount of memory space, at the cost of increasing the communication overhead.

As stated earlier, we do not update the original cost value arrays, but update the “delta” cost array which stores the cumulative values of the multipliers associated with a particular LAP variable. These delta costs are initialized to zero before executing the cost update step, and the updated delta array is used in the dynamic LAP solver for solving the $LR(\theta)$ during each iteration. The cost update step is extremely straightforward. It involves just addition or subtraction of θ values to the appropriate cost coefficients, in accordance with the objective function (2.1). This step is performed on the GPU with multiple threads, where each thread takes one multiplier value and updates at most three cost coefficients (the ones which are native to the PE). This operation has to be carried out atomically, since multiple threads may try and update the same cost coefficient, potentially causing a race condition.

3.4 Stopping Criteria

During constraint evaluation step, if the number of violations are zero, it means that our solution has achieved primal feasibility. Since $LD(\theta)$ operates in the dual space, such solution would be optimal to the MDADC (Heldt et al. [11]). Otherwise, we can use the best known upper bound to compute the optimality gap, and the iterations can be stopped once we reach the desired gap, or if we reach the specified limit on the number of iterations.

Algorithm 4: Kernel for updating cost values

- 1: **Input:** Delta cost array D , Array of IDs I and array of θ s T
 - 2: **Output:** Updated array D
 - 3: **parallel foreach** $i \in \{1, \dots, |I|\}$ **do**
 - 4: From $I[i]$ and A get cost values D_{ij}^{pq} , D_{ik}^{pr} and D_{jk}^{qr}
 - 5: **atomic:** $D_{ij}^{pq} \leftarrow D_{ij}^{pq} + T[i]$
 - 6: **atomic:** $D_{ik}^{pr} \leftarrow D_{ik}^{pr} + T[i]$
 - 7: **atomic:** $D_{jk}^{qr} \leftarrow D_{jk}^{qr} - T[i]$
 - 8: **end**
-

CHAPTER 4

RESULTS

The accelerated Lagrangian heuristic was coded in C++ and CUDA C programming languages and deployed on the Blue Waters Supercomputing Facility at the University of Illinois at Urbana-Champaign. Each PE has an AMD Interlagos model 6276 CPU, with 8 cores, 2.3GHz clock speed, and 32GB memory; connected to an NVIDIA GK110 “Kepler” K20X GPU, with 2688 processor cores, and 6GB memory. The step size was initialized to $\lambda^0 = 0.001$, with the reduction factor $\alpha = 0.925$. Problem instances with different sizes were generated by varying the values of N and K . The number of vertices N was selected from $[50, 500]$ with increments of 50; and the number of partitions K was selected from $[50, 500]$ with increments of 50. Table 4.1 summarizes the number of variables and multiplier values for different problem sizes. For each problem category, we generated 15 problem instances using the following strategy:

1. For each partition, every vertex was given a label between 0 and $(N-1)$, without repetition. Therefore, exactly K vertices (one vertex from each partition) will have the same label. Let ℓ_i^p represent the label of vertex i from partition p . Then, the pairwise cost coefficients between the vertices from different partitions are calculated using the formula $C_{ij}^{pq} = |\ell_i^p - \ell_j^q|$. As a result, we get N vertex disjoint cliques, each containing K vertices, with a zero total cost. We will refer to this solution as the “ground truth,” which is stored separately.
2. Then, to generate a problem instance of MDADC, noise is introduced in the cost coefficients of the ground truth, with normal distribution $N(0, \sigma)$, $\sigma \in (0, 1)$. As a result, the pairwise cost coefficients get modified according to the expression: $C_{ij}^{pq} \leftarrow C_{ij}^{pq} + N(0, \sigma)$. For each of the five basic problem instances, we generated three variants with three $\sigma \in \{0.3, 0.4, 0.5\}$ (total of 15 instances for each problem category).

For $\sigma = 0$, the problem is trivial to solve, since assigning a value of 1 to all the zero-cost variables will recover the ground truth. As the value of σ increases, it becomes increasingly challenging to recover ground truth, as the problem becomes difficult to solve. A large σ value can also lead to the optimal solution being different than the ground truth, because the cost coefficients are disturbed beyond repair.

The reason behind using the above scheme is that we intend to use the algorithm for data association in information fusion applications; and in these applications, the pairwise edge weights in a particular clique are consistent in the following sense. Consider real-world entities A, B and C ; if A is similar to B and B is similar to C , then it is also likely that A is similar to C . As a result, the pairwise weights cannot be assumed to be completely random. We performed separate tests on instances with randomly generated edge weights, and for these instances, we found that there is a significant increase in the number of violated constraints, and also the Lagrangian scheme requires a large number of iterations. The same behavior is observed when the noise σ is increased (see the numerical results discussed next).

4.1 Experiments with Single PE

For these experiments, our accelerated Lagrangian heuristic was executed on the various problem instances using only a single PE. We recorded the average iteration time, its components, number of initial violations, and Lagrangian update iterations for different combinations of N , K , and σ , by fixing two of the three parameters and systematically increasing the remaining one. These results are shown in Figures 4.1 and 4.2, and Tables 4.2, 4.3, and 4.4. From Figures 4.1 and 4.2, We can see that the average iteration time increases if we increase any one of the three parameters. For larger K and N , the problem contains more variables and constraints, which naturally increases its complexity and execution time. The σ value represents the level of noise present in the pairwise cost coefficient. Larger σ corresponds to more noise, which in turn makes the LAP subproblems more difficult to solve. Larger σ also increases the number of violations in the transitivity constraints, resulting in increased execution time. From Tables 4.2, 4.3, and 4.4, we can see that, with increasing N the LAP solution step is the

Table 4.1: Table showing number of variables for various problem sizes

K	N	$ \mathbf{x} $	$ \phi $
100	100	4.9500×10^7	4.8510×10^{11}
	200	1.9800×10^8	3.8808×10^{12}
	300	4.4550×10^8	1.3098×10^{13}
	400	7.9200×10^8	3.1046×10^{13}
	500	1.2375×10^9	6.0638×10^{13}
200	100	1.9900×10^8	3.9402×10^{12}
	200	7.9600×10^8	3.1522×10^{13}
	300	1.7910×10^9	1.0639×10^{14}
	400	3.1840×10^9	2.5217×10^{14}
	500	4.9750×10^9	4.9253×10^{14}
300	100	4.4850×10^8	1.3365×10^{13}
	200	1.7940×10^9	1.0692×10^{14}
	300	4.0365×10^9	3.6086×10^{14}
	400	7.1760×10^9	8.5538×10^{14}
	500	1.1213×10^{10}	1.6707×10^{15}
500	500	3.1188×10^{10}	7.7657×10^{15}

major contributor in the execution time. However, with increasing K and σ , constraint evaluation and cost update steps become the major contributors, resulting in dramatic increase in both the average iteration time and the number of iterations, due to large number of initial violations.

4.2 Experiments with Multiple PEs

For these experiments, our accelerated Lagrangian heuristic was executed on two of the large problem instances with $(K, N) \in \{(200, 200), (500, 500)\}$, using up to 128 PEs. Using multiple PEs, we are able solve these large instances of MDADC that cannot be otherwise solved on a single PE due to limitations on the memory and computational resources. To gauge the benefit of using multiple PEs, we performed strong scalability studies and reported the speedup obtained by systematically increasing the number of PEs. These results are shown in Tables 4.5 and 4.6, and Figures 4.3, 4.4, 4.5 and 4.6.

Table 4.2: Results for $K = 100$ and $\sigma = 0.4$ on single PE

N	Initial Violations	Avg. Itns.	Time per Itn. (s)	Iteration Time Breakdown (s)		
				SP Sol.	Con. Eval.	Cost Update
50	741,316	5.0	0.782	0.367	0.224	0.131
100	1,490,052	4.8	2.482	1.097	0.565	0.494
150	2,285,126	5.2	4.668	2.486	0.967	1.086
200	3,020,187	5.4	7.790	4.297	1.451	1.913
250	3,761,414	5.2	12.598	7.202	2.093	3.108
300	4,491,080	5.2	18.257	10.409	2.801	4.591

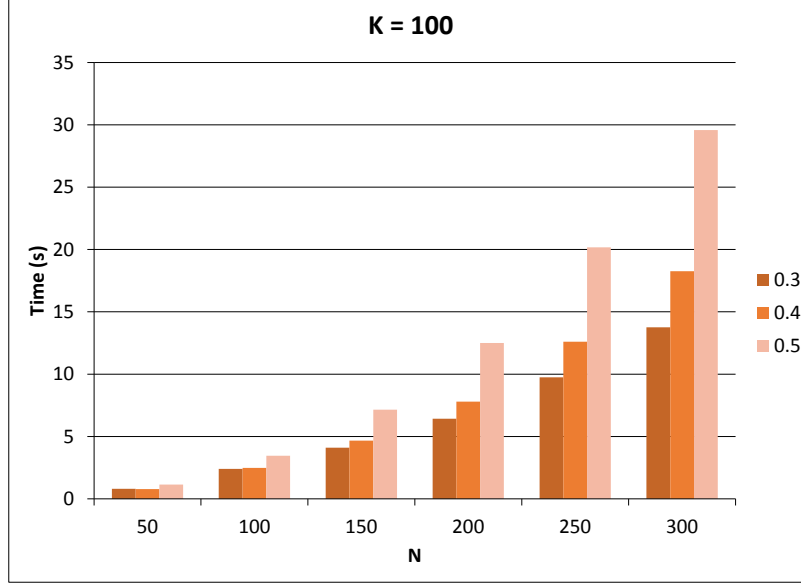
Table 4.3: Results for $N = 100$ and $\sigma = 0.4$ on single PE

K	Initial Violations	Avg. Itns.	Time per Itn. (s)	Iteration Time Breakdown (s)		
				SP Sol.	Con. Eval.	Cost Update
50	183,896	6.4	0.771	0.337	0.065	0.107
100	1,490,052	4.8	2.482	1.097	0.565	0.494
150	5,108,848	5.0	6.184	2.390	1.957	1.371
200	12,116,978	5.4	12.301	4.122	4.533	2.931
250	23,869,314	6.0	21.846	6.345	8.729	6.007
300	41,140,936	6.8	39.629	9.051	14.795	14.762

Table 4.4: Results for $K = 100$ and $N = 100$ on single PE

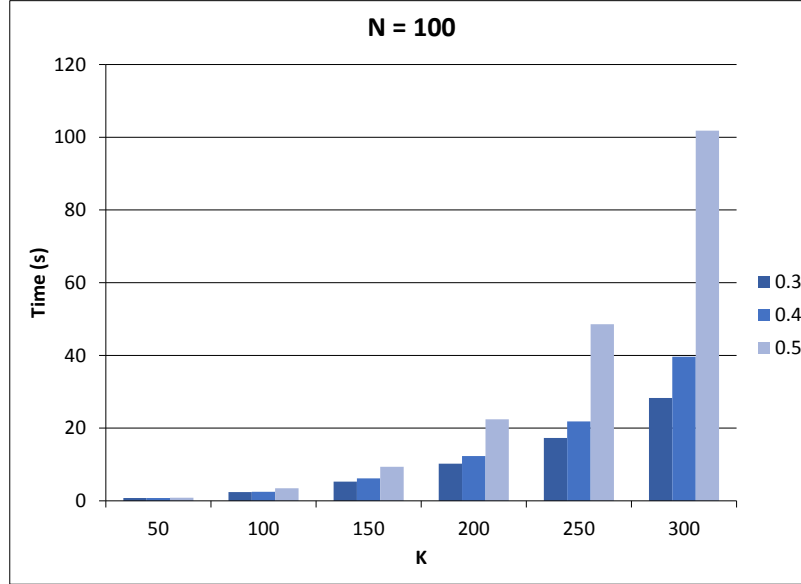
σ	Initial Violations	Avg. Itns.	Time per Itn. (s)	Iteration Time Breakdown (s)		
				SP Sol.	Con. Eval.	Cost Update
0.3	86,944	2.8	2.402	0.851	0.599	0.550
0.4	1,490,052	4.8	2.482	1.097	0.565	0.494
0.5	5,428,800	8.0	3.454	1.607	0.810	0.694

Figure 4.1: Average iteration time for varying N on single PE.



From the speedup profiles, we see that the constraint evaluation step benefits the most from parallelization, while the cost update step benefits the least. This is because the constraint evaluation step possesses the highest degree of granularity, which means that each constraint can be evaluated by a single thread. The cost update step is not granular because of three reasons: (1) The subsets of multipliers present on each PEs are broadcast and processed batches; (2) Each thread assigned to each dual multiplier only updates the relevant cost coefficients, which may result in thread divergence due to unequal allocation of work per thread; and (3) A thread uses `atomic` operations to update the cost coefficients, which may serialize its execution. We would like to point out that an efficient approach may be conceived for the cost update step, however, it is not explored in this work and left as a future research direction. The SP solution and multiplier update steps exhibit decent speedup, which suffers mostly because of the necessity of communicating intermediate results across all the PEs. We do see, however, that the speedup factors (both overall and component-wise) improve with increasing values of σ and for larger K and N . This is simply because, for these problems, the amount of parallel work exceeds the amount of sequential work,

Figure 4.2: Average iteration time for varying K on single PE.



including any inter-PE communication overhead. Therefore, the proposed parallel algorithm is very well-suited for solving large instances of MDADC, with noisy cost coefficients.

4.3 Discussion

From all the numerical results, it is clear that GPUs are extremely efficient in solving (or obtaining lower bounds on) large instances of MDADC. If we compare the iteration times of our GPU-based implementation, with those of the Hadoop Map-Reduce based implementation of Tauer and Nagi [9], we can see that our GPU-based approach is several orders of magnitude faster than Map-Reduce based approach. Additionally, the GPU-based implementation is able to handle very large instances of MDADC, without much deterioration in the parallel speedup. The Map-Reduce implementation will also be able to handle large problems, with adequate number of processors, however, we suspect that it will not be able to beat the average iteration time benchmark of our GPU-based implementation.

Table 4.5: Strong Scalability Results for $K = 200$ and $N = 200$

σ	PEs	Time per Iteration (s) [Parallel Speedup]			
		SP Sol.	Con. Eval.	Cost Update	Overall
0.3	1	10.656 [1.00]	11.890 [1.00]	9.650 [1.00]	32.222 [1.00]
	2	6.012 [1.77]	6.095 [1.95]	6.435 [1.50]	18.569 [1.74]
	4	3.511 [3.04]	3.066 [3.88]	5.174 [1.87]	11.766 [2.74]
	8	3.454 [3.09]	1.513 [7.86]	4.522 [2.13]	9.502 [3.39]
	16	2.928 [3.64]	0.783 [15.19]	4.314 [2.24]	8.036 [4.01]
	32	2.463 [4.33]	0.390 [30.46]	4.300 [2.24]	7.169 [4.49]
	64	2.302 [4.63]	0.212 [56.12]	4.283 [2.25]	6.814 [4.73]
0.4	1	16.747 [1.00]	12.052 [1.00]	15.039 [1.00]	44.135 [1.00]
	2	8.919 [1.88]	5.998 [2.01]	9.672 [1.55]	24.747 [1.78]
	4	5.241 [3.20]	3.013 [4.00]	7.742 [1.94]	16.087 [2.74]
	8	3.524 [4.75]	1.524 [7.91]	5.602 [2.68]	10.696 [4.13]
	16	2.332 [7.18]	0.738 [16.32]	5.150 [2.92]	8.250 [5.35]
	32	1.760 [9.52]	0.368 [32.75]	4.953 [3.04]	7.113 [6.21]
	64	1.615 [10.37]	0.206 [58.42]	5.080 [2.96]	6.922 [6.38]
0.5	1	28.871 [1.00]	16.855 [1.00]	53.718 [1.00]	100.690 [1.00]
	2	14.746 [1.96]	8.161 [2.07]	30.321 [1.77]	53.850 [1.87]
	4	7.714 [3.74]	4.072 [4.14]	17.738 [3.03]	29.841 [3.37]
	8	4.546 [6.35]	2.049 [8.22]	15.134 [3.55]	21.900 [4.60]
	16	2.812 [10.27]	1.040 [16.20]	14.649 [3.67]	18.588 [5.42]
	32	1.778 [16.24]	0.482 [34.97]	8.635 [6.22]	10.949 [9.20]
	64	1.374 [21.02]	0.241 [70.05]	8.905 [6.03]	10.590 [9.51]

Table 4.6: Strong Scalability Results for $K = 500$, $N = 500$, and $\sigma = 0.4$

PEs	Time per Iteration (s) [Parallel Speedup]			
	SP Sol.	Con. Eval.	Cost Update	Overall
64	14.594 [1.00]	23.944 [1.00]	451.277 [1.00]	490.132 [1.00]
80	11.766 [1.24]	19.046 [1.26]	411.659 [1.10]	442.751 [1.11]
96	9.953 [1.47]	15.700 [1.53]	384.968 [1.17]	410.861 [1.19]
112	8.444 [1.73]	13.401 [1.79]	370.315 [1.22]	392.380 [1.25]
128	8.111 [1.80]	11.965 [2.00]	368.344 [1.23]	388.648 [1.26]

Figure 4.3: Speedup Factors for $K = 200$, $N = 200$ and $\sigma = 0.3$.

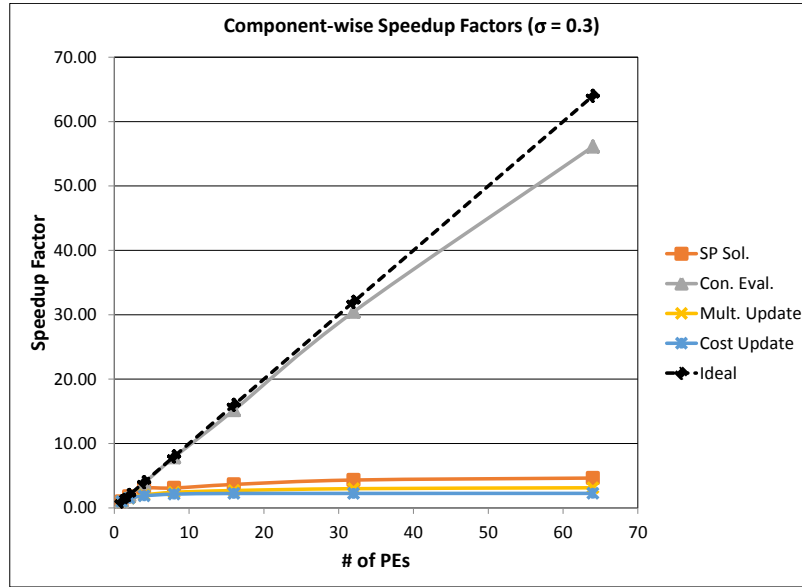


Figure 4.4: Speedup Factors for $K = 200$, $N = 200$ and $\sigma = 0.4$.

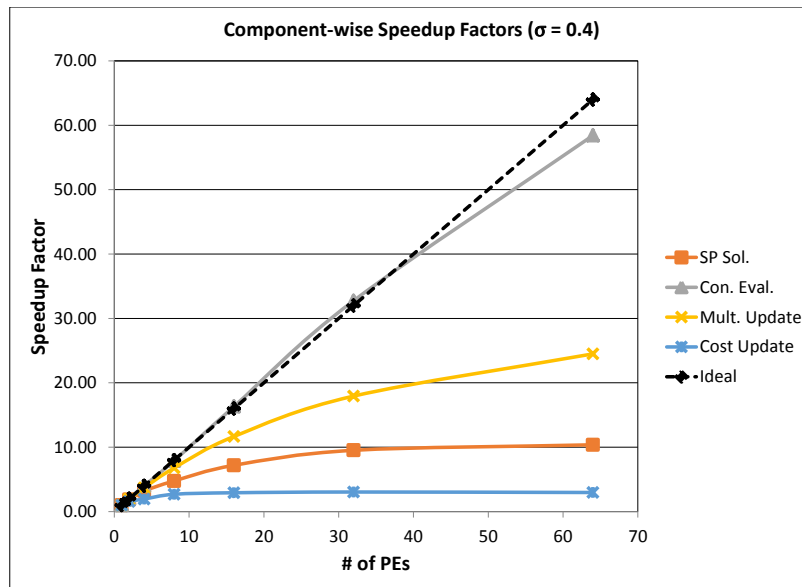


Figure 4.5: Speedup Factors for $K = 200$, $N = 200$ and $\sigma = 0.5$.

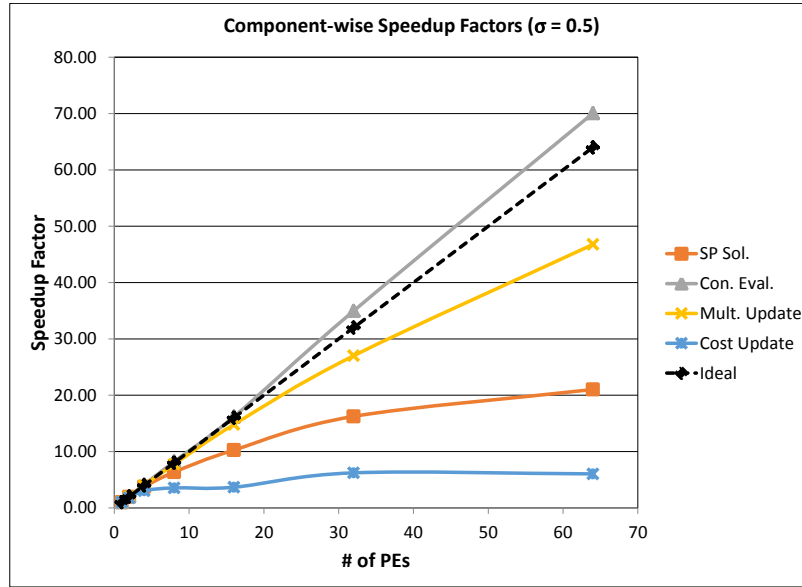
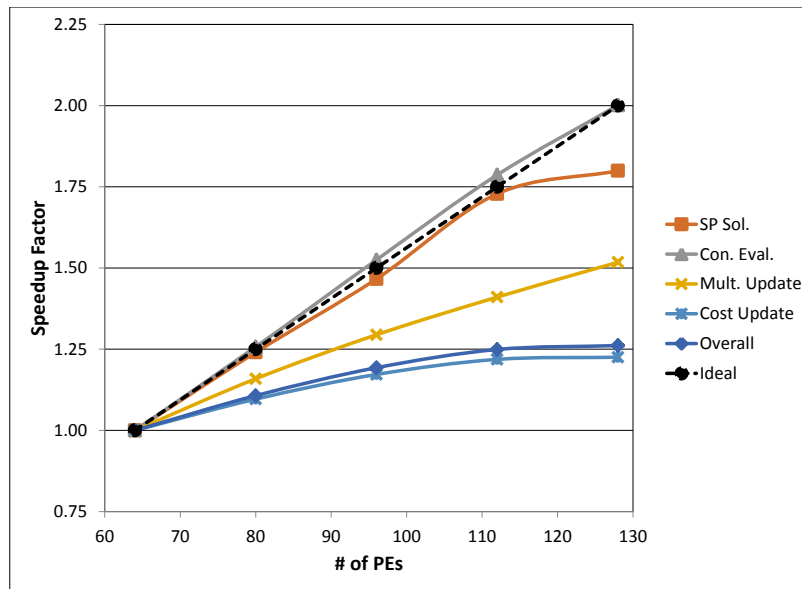


Figure 4.6: Speedup Factors for $K = 500$, $N = 500$ and $\sigma = 0.4$.



CHAPTER 5

SUMMARY AND CONCLUSIONS

For problems like Multi-dimensional Assignment Problems (MAPs) with decomposable costs, which include massive amount of computations and storing huge amounts of data, parallelization seems to be a good approach owing to the fact that the individual LAPs can be solved independently of each other. Apart from this major step, all the other steps, like finding violations, updating the Lagrangian multipliers and updating the cost values, can all be easily parallelized. With the advent of distributed systems and programmable GPUs, it became apparent that solving such MAPs using these technologies should make it computationally much more efficient.

In this work, we developed a GPU-accelerated Lagrangian heuristic for obtaining tight lower bounds and exact solutions for large instances of the MDADC. Our parallel architecture consists of a computational grid with multiple CPU-GPU pairs. Main contributions of this thesis are: (1) Efficient parallelization of the various steps of the Lagrangian heuristic on single and multi-GPU systems; (2) An efficient strategy for solving multiple (small) LAPs using “tiled” matrices; (3) Leveraging dynamic LAP solver when cost updates are minor; and (4) An efficient scheme for enumerating violated constraints, and storing/retrieving the non-zero dual variables. Due to these improvements, our parallel solver algorithm is significantly faster than the sequential approach, and it is able to handle large problem instances with $K = 500$ partitions, each with $N = 500$ vertices, which contains over 31 billion variables and 7 quadrillion constraints.

One thing that negatively affects scalability in this work is the fact that all the row assignments have to be stored on each of the GPUs while finding violations. This, sometimes, leads to out-of-memory errors when the problem size goes beyond the aforementioned limits for both N and K together. This can be reduced using better distribution of row assignments for each device finding violations for a particular set of LAPs. Problems with huge N values

can also lead to scarcity of memory on GPUs because of LAPs getting too large to fit on a single GPU. This can be countered using the multi-GPU version of the parallel Hungarian algorithm developed by Date and Nagi [10]. In this work we distribute LAPs among multiple processors assuming that each LAP is small enough to fit on a single GPU. Extending this to large LAPs that are distributed among multiple processors might not be as straight-forward. But once implemented, this can make the algorithm completely scalable.

Another important extension of this work would be to explore the possibility of applying the Lagrangian dual ascent algorithm for updating the dual multiplier values. The dual ascent algorithm always operates on the provable directions of ascent, rather than suspected directions employed by the naïve subgradient optimization. As a result, the dual ascent algorithm is capable of generating a non-decreasing sequence of strong lower bounds, which will be extremely valuable in the B&B scheme. This task however is non-trivial, since it may require exploring alternate formulations for MDADC which will be amenable for the application of the Lagrangian dual ascent.

GPU programming is fairly new, which makes this algorithm look promising. Once we have more powerful GPUs with more memory and faster communication to the processors, most of the memory restrictions mentioned above might become easy to overcome. Future endeavors can be directed at developing similar algorithms which are faster and more efficient on new hardware architectures.

CHAPTER 6

REFERENCES

- [1] J. Munkres. Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial and Applied Mathematics*, Vol. 5, No. 1: 32-38, 1957.
- [2] H.-J. Bandelt, Y. Crama and F. C. R. Spieksma. Approximation algorithms for multi-dimensional assignment problems with decomposable costs. *Discrete Applied Mathematics*, Vol. 49: 25-50, 1994.
- [3] A. B. Poore and N. Rijavec. A Lagrangian relaxation algorithm for multidimensional assignment problems arising from multitarget tracking. *SIAM Journal of Optimization*, Vol. 3, No. 3: 544-563, 1993.
- [4] A. B. Poore. Multidimensional assignment formulation of data association problems arising from multitarget and multisensor tracking *Computational Optimization and Applications*, Vol. 3, No. 1: 27-57, 1994.
- [5] A. B. Poore and A. J. Robertson. A new Lagrangian relaxation based algorithm for a class of multidimensional assignment problems. *Computational Optimization and Applications*, Vol. 8: 129-150, 1997.
- [6] A. J. Robertson. A set of greedy randomized adaptive local search procedure (GRASP) implementations for the multidimensional assignment problem. *Computational Optimization and Applications*, Vol. 19: 145-164, 2001.
- [7] Y. Kuroki and T Matsui. An approximation algorithm for multidimensional assignment problems minimizing the sum of squared errors. *Discrete Applied Mathematics*, Vol. 157: 2124-2135, 2009.
- [8] R. Roverso, A. Naiem, M. El-Beltagy, S. El-Ansary and S. Haridi. A GPU-enabled solver for time-constrained linear sum assignment problems. *Informatics and Systems (INFOS), The 7th International Conference on:* 1-6, 2010.
- [9] G. Tauer and R. Nagi. A map-reduce Lagrangian heuristic for multidimensional assignment problems with decomposable costs. *Parallel Computing*, Vol. 39, No. 11: 653-668, 2013.

- [10] K. Date and R. Nagi. GPU-accelerated Hungarian algorithms for the linear assignment problem. *Parallel Computing*, Vol. 57: 52-72, 2016.
- [11] M. Held, P. Wolfe and H. P. Crowder, Validation of subgradient optimization. *Mathematical Programming*, Vol. 6, No. 1: 62-88, 1974.
- [12] H.-J. Bandelt, A. Maas and F. C. R. Spieksma. Local Search Heuristics for Multi-Index Assignment Problems with Decomposable Costs. *The Journal of the Operational Research Society*, Vol. 55, No. 7: 694-704, 2004.
- [13] G. Tauer, R. Nagi and M. Sudit. The graph association problem: mathematical models and a lagrangian heuristic. *Naval Research Logistics (NRL)*, Vol. 60, No. 3: 251-268, 2013.
- [14] A. B. Poore. Multidimensional assignment problems arising in multi-target and multisensor tracking. *Nonlinear Assignment Problems*: 13-38, 2000.
- [15] T. Milenković, W. L. Ng, W. Hayes and N. Pržulj. Optimal Network Alignment with Graphlet Degree Vectors. *Cancer Informatics*, Vol. 9: 121-137, 2010.
- [16] R. Singh, J. Xu and B. Berger. Global alignment of multiple protein interaction networks with application to functional orthology detection. *Proceedings of the National Academy of Sciences*, Vol. 105, No. 35: 12763-12768, 2008.
- [17] K. Date, G. A. Gross, S. Khopkar, R. Nagi and K. Sambhoos. Data association and graph analytical processing of hard and soft intelligence data. *Information Fusion (FUSION), 2013 16th International Conference on*: 404-411, 2013.
- [18] K. Date, G. A. Gross and R. Nagi. Test and evaluation of data association algorithms in hard+soft data fusion. *Information Fusion (FUSION), 2014 17th International Conference on*: 1-8, 2014.
- [19] G. A. Gross, K. Date, D. R. Schlegel, J. J. Corso, J. Llinas, R. Nagi, Rakesh and S. C. Shapiro. Systemic test and evaluation of a hard+soft information fusion framework: Challenges and current approaches. *Information Fusion (FUSION), 2014 17th International Conference on*: 1-8, 2014
- [20] C. N. Vasconcelos and B. Rosenhahn. Bipartite graph matching computation on GPU. *Energy Minimization Methods in Computer Vision and Pattern Recognition*: 42-55, 2009

- [21] G. A. Korsah, A. Stentz and M. B. Dias. The Dynamic Hungarian Algorithm for the Assignment Problem with Changing Costs. *Robotics Institute, Carnegie Mellon University*, 2007
- [22] J. L. Walteros, C. Vogiatzis, E. L. Pasiliao and P. M. Pardalos. Integer programming models for the multidimensional assignment problem with star costs. *European Journal of Operational Research*, Vol. 235, No. 3: 553-568, 2014
- [23] C. Vogiatzis, E. L. Pasiliao and P. M. Pardalos. Graph partitions for the multidimensional assignment problem. *Computational Optimization and Applications*, Vol. 58, No. 1: 205–224, 2014