EXPLORING MODEL PARALLELISM IN DISTRIBUTED SCHEDULING OF NEURAL NETWORK FRAMEWORKS

BY

PALLAVI SRIVASTAVA

THESIS

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

Urbana, Illinois

Adviser:

Professor Indranil Gupta

ABSTRACT

The growth in size and computational requirements in training Neural Networks (NN) over the past few years has led to an increase in their sizes. In many cases, the networks can grow so large that can no longer fit on a single machine. A model parallel approach, backed by partitioning of Neural Networks and placement of operators on devices in a distributed system, provides a better distributed solution to this problem. In this thesis, we motivate the case for device placement in Neural Networks. We propose, analyze and evaluate m-SCT, a polynomial time algorithmic solution to this end. Additionally, we formulate an exponential time optimal ILP solution that models the placement problem. We summarize our contributions as:

1. We propose a theoretical solution to the memory constrained placement problem with makespan and approximation ratio guarantees.

2. We compare and contrast m-SCT with other state of the art scheduling algorithms in a simulation environment and show that it consistently performs well on real world graphs across a variety of network bandwidths and memory constraints.

3. We lay the foundation for the experimental evaluation of the proposed solutions in existing Machine Learning frameworks.

To Cinda Heeren, my mentor and friend.

ACKNOWLEDGMENTS

I would like to thank my advisor, Professor Indranil Gupta, for his continual support and advice. I would also like to thank my colleagues and collaborators, Linda Cai and Jintao Jiang, without whom this thesis could not have been written.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

NN	Neural Network
TF	TensorFlow
LP	Linear Programming
ILP	Integer Linear Programming
CNN	Convolution Neural Network
SCT	Small Communication Time
UET-UCT	Unit Computation Time Unit Communication Time
ETF	Earliest Task First
RNN	Recurrent Neural Network
LSTM	Long Short-Term Memory
FFN	Feed Forward Neural Network
NMT	Neural Machine Translation
ML	Machine Learning

CHAPTER 1: INTRODUCTION

The past few years have seen a huge shift in interest towards Machine Learning. With the advent of deep learning, the complexity in training Neural Networks (NN) has increased manifold. The training process is expensive, resource intensive and time consuming. The time increases as the models become more complex and training data increases. For instance, in [1], training the 152-layer neural network for one iteration requires 11.3 billion floating point operations and the model is trained up to 6×10^5 iterations.

In order to keep up with the infrastructure requirements of such explosive growth, distributed computing in Machine Learning is the focus of a lot of current research [2] [3] [4] [5] and software development.

The software industry has universalized the usage of many specialized NN frameworks that are not only optimized to shorten the training time, but also standardize the development of Machine Learning algorithms. Several of these frameworks such as MXNet [4], Torch7 [6], Theano [7] and TensorFlow [8] are described in more detail in chapter 2.

Recent research has led to multiple developments in data parallelism [9] [10] [11]; a technique which focuses on distributing data across different processors, each possessing an independent copy of the model. This approach stems from SIMD (single instruction, multiple data) computer architecture [12, p. 182], one of the oldest ways of parallel processing on computers. Adequate user support has been provided for these techniques in leading Machine Learning (ML) frameworks [4] [6] [13] [7] [14] such as TensorFlow (TF) [8] [15]. While data parallelism is an effective way to handle burgeoning training data, it does little to alleviate the problems caused by the ever increasing size of the models themselves as they become more complex.

Model parallelism is an effective way to handle this issue as it distributes a single model across multiple processors which independently compute model parameters for the part of the model assigned to them. In parallel computing terms, it can be thought of as MPSD (multiple program, single data). Model parallelism can prove to be an indispensable tool for NN models which are too big to fit on a single machine. However, there is no explicit support for users to deploy this technique on most ML engines effectively and users are required to manually place parts of the model on different processors in an arbitrary fashion. Recent work on optimization of these placements [16] make use of reinforcement learning and are a step up from human expert placements and heuristics. However, using reinforcement learning for placement is both time and resource intensive and therefore has not been deployed widely.

In this thesis, we develop an algorithmic foundation for the device placement problem and

explore multiple models such as *k-min cut* and *scheduling with communication delay* to fit the problem. Our aim is to provide a relatively fast, easy to deploy and effective theoretical solution for this problem.

1.1 CONTRIBUTION OF THIS THESIS

1. We introduce the concept of *wait time* and use it to establish why scheduling with communication delay is preferred over k-min cut to model the placement problem.

2. In chapter 2 (section 4.4), we formulate an integer linear programming (ILP) problem using the placement problem constraints (including memory constraints). The running time for this optimal solution is $O(2^{n^2})$, where n is the total number of nodes.

3. In chapter 8, we further demonstrate the difficulty in applying LP relaxation or other convex optimization relaxation techniques to approximate the optimal solution in polynomial time.

4. Next, in chapter 4 (section 4.3.2), we propose m-SCT, a modified version of the SCT algorithm [17] that incorporates memory constraints in the system. We prove the modified approximation ratio to be within $(1+\frac{2+2\rho}{(2+\rho)m})\cdot\frac{1}{K-1}$ of the finite SCT algorithm approximation ratio, where the total memory available on all machines is K times the total memory required by the network, m is the number of available machines, and ρ is ratio between the maximum communication time between any two nodes and the minimum computation time for any node.

5. Both the optimal formulation and the m-SCT algorithm expect a directed acyclic graph (DAG). Since back-propagation in neural networks introduces some cycles, in chapter 5, we first establish the makespan for the forward computation case and put forth a mathematical proof (section 5.4.2, theorem 5.3) showing that the approximation ratio remains unchanged after back-propagation. Furthermore, we show that the makespan of the backward pass is within C_0 times the makespan for the forward pass (section 5.4.2, theorem 5.1), where C_0 is the maximum ratio between (a) corresponding backward pass edge weight and forward pass edge weight and (b) corresponding backward pass node weight and forward pass node weight.

6. In chapter 7, we experimentally compare the performance of m-SCT, m-ETF and m-TOPO using simulation. We vary the bandwidth and the total number of processors for

all experiments. For each case, we experiment under two memory constraints (a) fixed total memory in the system (section 7.1) and (b) fixed amount of memory available on each machine (section 7.2). We show that m-SCT consistently performs well for large bandwidth for both Inception-V3 and our small Convolution Neural Network (CNN) model.

7. Finally, we demonstrate the viability of using TensorFlow to implement our theoretical model. We implement the ability to inject device placement post hoc into user specified code and therefore show that it can successfully be used to develop a one-click device placement solution in the future.

1.2 OUTLINE OF THIS THESIS

1. In Chapter 2, we provide the system and algorithm preliminaries as well as provide background on Machine Learning systems and model parallelism.

2. In Chapter 3, we present the problem statement and related work. We discuss the *k-min cut* and *scheduling with communication delay* models as well as the relevant existing algorithms under the latter. We also touch upon the motivating factors for our thesis in device placement.

3. In Chapter 4, we discuss the exponential time optimal solution for placement problem. We also modify the polynomial time SCT algorithm to incorporate memory constraints and create m-SCT. We then discuss the new approximation ratio as well as an alternate way to calculate the greedy priority using weighted sums for m-SCT.

4. In Chapter 5, we discuss how to incorporate back-propagation in the algorithms from Chapter 4. We also prove some guarantees on the makespan of the backward pass.

5. In Chapter 6, we present our implementation. We discuss the details of our simulation including how to estimate communication and computation times for our models. We also provide a reference to the open source graph library and LP solver used in our experiments.

6. In Chapter 7, we present our experimental results and discussion. We simulate the m-SCT algorithm on Inception-v3 and a small CNN graph. We then compare its performance with that of memory constrained m-ETF algorithm as well as the topological sort algorithm m-TOPO, for varying network bandwidth and total number of processors.

7. In Chapter 8, we acknowledge the theoretical directions that did not yield successful results and the reasons for their failure.

8. In Chapter 9, we present our conclusions.

9. In Chapter 10, we set forth the future work for the second part of this project. Since the primary focus of this work is to lay a theoretical foundation, this chapter discusses the ensuing implementation and experimental work needed to create a holistic solution. We also briefly mention some limitations of our work that may be examined in greater detail at a future time.

CHAPTER 2: PRELIMINARIES

In this chapter we discuss some preliminary systems and algorithm terminologies.

2.1 SYSTEMS PRELIMINARIES

Model Parallelism: In the context of distributed ML, model paralleli refers to distributing a model across different devices in a way such that every part of the model is responsible for training the same data but is individually responsible for maintaining its assigned set of model parameters. In figure 2.1, the model is distributed horizontally between GPU 1 and GPU 2. TensorFlow provides implicit support for model paralleli. When the user specifies the device placement, the appropriate communication between the sub-graphs is inserted automatically.

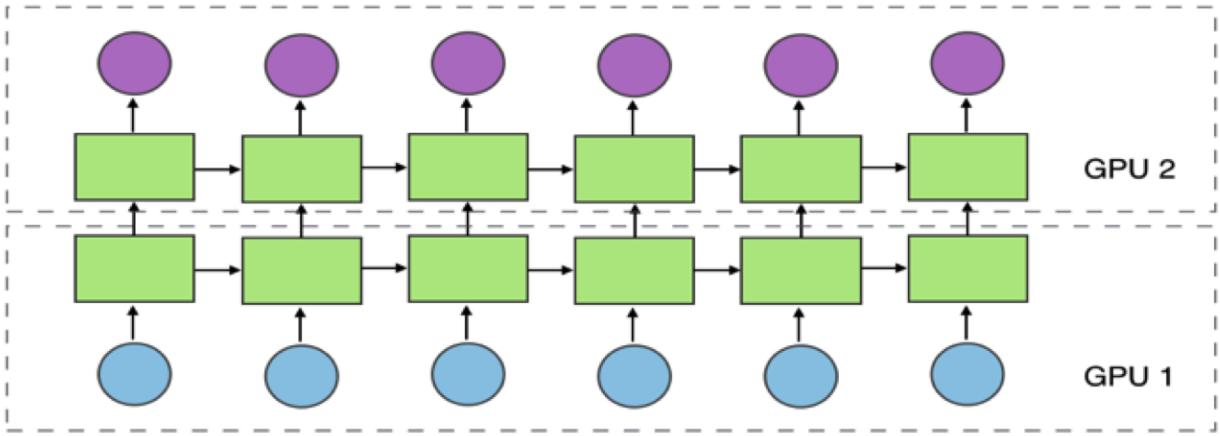

Figure 2.1: Model distributed across GPU 1 and GPU 2 to demonstrate model paralleli [4]

Machine Learning Frameworks: Recent years have witnessed the rise of multiple Machine Learning systems that are scalable and capable of supporting intensive computation. The increase in the availability of big data i.e., large and high quality datasets (such as [18] [19]), has spurred on the development of these frameworks even further. Apache's MXNet [4] is a popular framework that provides support for multiple languages. It also supports different programming paradigms including declarative and imperative programming. Torch7 [6] is an open source Machine Learning library that extends Lua [20], a lightweight scripting language, that focuses on efficient numerical computation. Microsoft's CNTK toolkit [13] specializes in training deep neural networks. It provides a number of optimized built-in com-

ponents to easily express many widely used Neural Networks (NN) models like Recurrent Neural Network (RNN) / Long Short-Term Memory (LSTM) [21] and Feed Forward Neural Network (FFN) [22] to name a few. Several other commonly used frameworks include Theano [7], Chainer [14] and Caffe [23]. For the purposes of this thesis and our experiments, we choose TensorFlow, Google's high performing and widely deployed Machine Learning (ML) engine.

TensorFlow TensorFlow is an open source software library for ML, developed by the Google Brain team. TensorFlow (TF)'s architecture is based on a *dataflow* graph, responsible for both computation and maintaining state. The *dataflow* graph also houses operations that are responsible for mutating the system's state. TF follows a very high level programming paradigm and doesn't restrict its users to any low level organization models (like the parameter server model). The nodes of the data flow graph can be arbitrarily mapped to physical devices or even particular CPUs and GPUs within a single machine. The flexibility of this paradigm is very conducive to exploring different device placements for model paralleli and testing their efficiencies.

The serialized version of the *dataflow* graph is known as *graphdef*. The serialization is especially useful as it allows for the consolidated user graph to be language independent and be used across a variety of TF instances and external applications.

TF Operations and Tensors Tensors [24] are generalizations of vectors and matrices to higher dimensions. In TF, the edges of the *dataflow* graph have associated tensors which represent the data produced and consumed by the *operation* nodes. TF tensors are generally immutable. A TF *operation* is a node in the *dataflow* graph that accepts tensors as input and produces tensors as output. It is the computation unit of the TensorFlow model.

TF Variables TF variables are mutable tensors. As a construct, they are best utilized when maintaining shared and persistent state that can be modified by operations according to the logical dictates of the user's model.

TF Timeline TF *timeline* is a TensorFlow object that can be used to record the computation time of tasks in the *dataflow* graph. The *timeline* object can be exported as a JSON file. The *timeline* tool's source code is available at tensorflow/tensorflow/python/client/timeline.py in the TensorFlow github repository.

2.2 ALGORITHM PRELIMINARIES

Directed Acyclic Graph (DAG) We define a DAG as a directed graph with no directed cycles.

Makespan The makespan of a schedule L for a graph G is the total execution time for graph G, given the device placement assigned by L.

Collocation We say two tasks are collocated if they are placed on the same machine.

Approximation Ratio We define the approximation ratio of an algorithm (for a minimization problem) as the maximum ratio between the algorithm's solution and the optimal solution.

CHAPTER 3: PROBLEM STATEMENT, MOTIVATION, AND PRIOR WORK

In this chapter, we present the problem statement, motivation and prior work for the placement problem.

3.1 PROBLEM STATEMENT

Given memory constraints, we want to generate a m-machine device placement for all the nodes in a dependency DAG G such that the makespan is minimized. Each node in G represents a computational task and each edge $u \rightarrow v \in E(G)$ represents that task v is dependent on task u. The node weights of G represent the computation time for each task, while the edge weights represent the communication size between 2 related nodes.
Each machine may not use more than M amount of memory, given the assumption that the total memory on any individual machine is M, and each node is associated with a certain amount of memory as well. Since Neural Networks (NN) require their dependency graphs to be executed thousands of times [25] [26], it is advantageous to pre-allocate memory for inputs and outputs associated with each task. Therefore, we assume each task is associated with a permanent memory usage that is persistent across training iterations.

3.2 MOTIVATION

Memory constraints With the advent of deep learning, the NN are increasingly growing in size and complexity. Most widely used models like RNNLM [27], Neural Machine Translation (NMT) [28] and Inception-v3 have very large memory footprints. According to [16] when the batch size for RNNLM and NMT is increased to 256 and their LSTM size is increased to 4096 and 2048 respectively, even a single layer of these models is unable to fit on a single machine. Model parallelism is the only viable option in such cases. This motivates the study and development of placement algorithms so that training times remain reasonably fast.

Limitations of Machine Learning solutions The placement problem is solved using reinforcement learning in [16]. This approach has several limitations as it essentially brute forces through a set of possible placements. Training is computationally intensive and depending on the setup, it can take up to several days to compute a single placement. The training process must be repeated if any changes are made either to the model or the devices

on which the model is being placed. Additionally, there is no way to determine the optimality of the solution obtained in this manner. It can also be argued that Machine Learning (ML) is an unreasonably exorbitant tool for this problem.

Placement as a part of a larger elastic solution If we consider the larger problem of developing elastic distributed systems [29], the constant allocation and de-allocation of resources for tasks would highly benefit from placement being computed on the fly. Therefore, it is worthwhile to invest research effort into algorithmic solutions that run faster than their ML counterparts.

3.3 PRIOR WORK

3.3.1 k-min cut

k-min cut Model *k-min cut* is an optimization problem which finds a minimum weighted edge cut that partitions a dependency graph G into k components. In the dependency graph for the placement problem, the edge weight is proportional to the communication time between the connected nodes. Therefore, a solution to *k-min cut* would provide a partition of G with minimal overall communication cost, resulting in a reduced makespan. *k-min cut* is an NP hard problem [30] and extensive research effort has been dedicated to providing good approximation algorithm for this problem [31] [32] [33]. A popular variation of the problem introduces balance constraints in order to obtain partition of uniform sizes [34] [35].

Deficiency of the Model The *k-min cut* model does not account for *wait time*. We define **wait time** as the machine idle time when no task can execute because their dependency have not been satisfied. In Figure 2.1, we show that two partitions with the same overall communication cost can have very different makespans. Figure 2.1 uses a dependency graph similar to that of Recurrent Neural Networks (RNN) and assumes unit computation and unit communication time. The number in each node denotes their starting time given the partition marked by the dotted line in the image. As we can see, the left partition results in a much smaller makespan than the right partition, even though they have the same total communication cost. Since *k-min cut* only minimizes the communication cost, we conclude that it is not a suitable model for our problem.

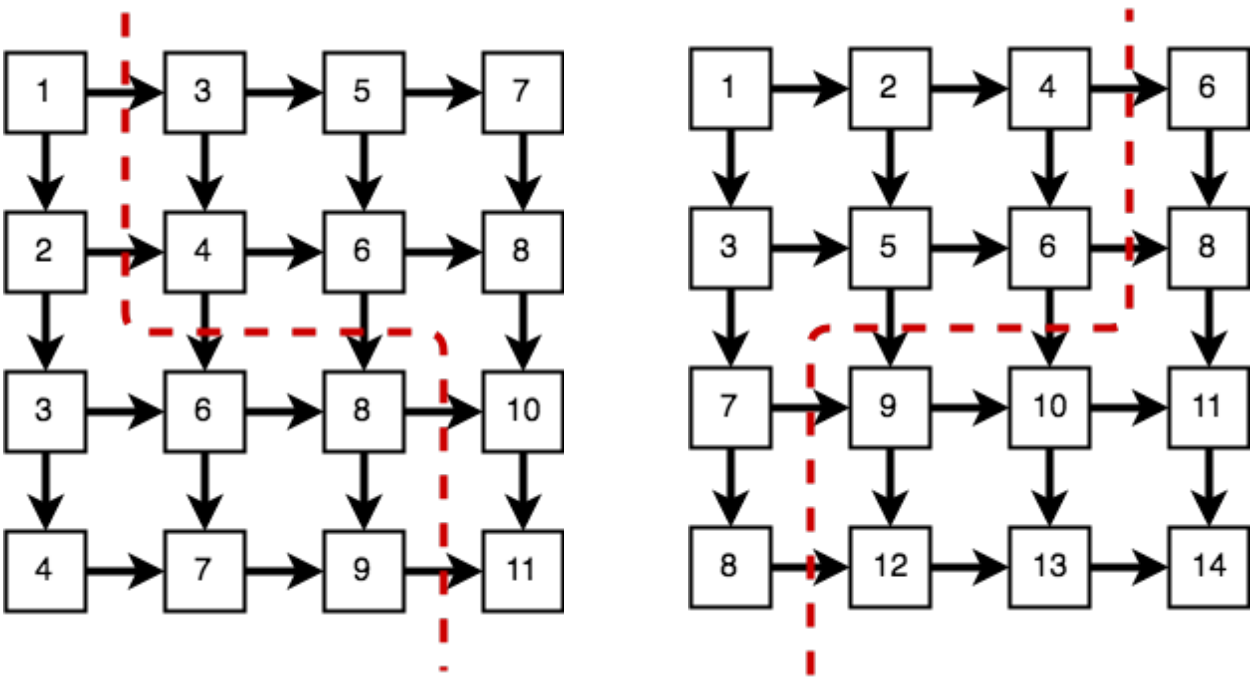

Figure 3.1: The left and right partitions have the same communication cost but different makespans

3.3.2 Scheduling with Communication Delay

Alternatively, we model the placement problem as a *scheduling with communication delay* problem.

Scheduling with Communication Delay Model Given a dependency DAG G, the *scheduling with communication delay* problem schedules tasks on machines while accounting for communication delays between related tasks placed on different machines. The problem's objective is to minimize the total execution time (makespan) of G. Since this model focuses on holistically minimizing the makespan instead of just concentrating on the communication time, it is better suited to the placement problem.

Variants of the problem The problem has three common variants. The unit computation time unit communication time (UET-UCT) version of the problem assumes unit computation time and unit communication time in the dependency graph G. The small communication time (SCT) version of the problem assumes that the ratio between the maximum communication time between any two nodes and the minimum computation time for any node is ≤ 1. The general version of the problem places no constraints on either communication or computation time.

NP-Hardness As proven in [36], the problem is NP hard even when reasonable accommodations are provided, such as infinite number of processors and UET-UCT conditions. Therefore, approximation algorithms are developed to provide a solution in polynomial time with good accuracy.

Existing Work Most approximation algorithms solve the *scheduling with communication delay* problem with special assumptions such as UET-UCT or SCT. To the best of the author's knowledge, no constant approximation ratio algorithm has been developed for the general *scheduling with communication delay* problem. For the general case, [37] describes an earliest task first (ETF) scheduling scheme which has the best known approximation ratio of $2 + \rho - \frac{1}{m}$, where ρ is ratio between the maximum communication time between any two nodes and the minimum computation time for any node in the graph and m is the total number of machines (for more details see section 4.2). Under the SCT constraint ($\rho \leq 1$), algorithms with better approximation ratios are known. By using linear programming (LP) relaxation and priority based greedy scheduling, [17] is able to achieve an approximation ratio of $\frac{4+3\rho}{2+\rho} - \frac{2+2\rho}{m(2+\rho)}$. Since $\rho \leq 1$ under the SCT assumption, the approximation ratio is constant (for more details see section 4.3).

Viability of the SCT Assumption Neural Networks are computationally intensive as mentioned in chapter 1. This characteristic, coupled with the fact that distributed NN frequently utilize high performance computers in data centers (resulting in smaller communication times), makes the SCT assumption viable. If for certain applications, the SCT assumption is significantly violated, a collocating scheme described in [38] can be applied at the pre-processing stage. Computationally small nodes are grouped together and treated as a single node to reduce ρ to near-SCT compliance.

CHAPTER 4: ALGORITHM DESIGN

In this chapter, we present the details of the three selected algorithms: topological sort, ETF [37] and SCT [17] algorithm to solve the scheduling with communication delay problem. We modify all three existing algorithms to incorporate the memory constraint per machine, resulting in modified algorithms m-TOPO, m-ETF and m-SCT. We assume the memory on all machines to be homogeneously distributed for the purposes of this thesis. We assume the total memory on any individual machine to be M. Furthermore we prove the approximation ratio of m-SCT.

We also model our memory constrained *scheduling with communication delay* problem with integer linear programming (ILP), which generates the optimal solution. For small to medium size graphs (up to 50 nodes), the results can be generated within a reasonable amount of time (at most a few hours). In the future, the optimal solution can be used to provide a baseline and measure the performance of our modified approximation algorithms.

4.1 TOPOLOGICAL SORT

4.1.1 Definition

Topological sort is a linear ordering of vertices in a DAG G, such that for each directed edge $u \rightarrow v$, u comes before v in the linear ordering.

4.1.2 Algorithm description

The existing topological sort algorithm [39] first topologically sorts the nodes in DAG G, then place nodes evenly on each machine.

4.1.3 Memory constrained version:

In our modified algorithm m-TOPO, we first number the total number of m machines from 1 to m. Then, we assign tasks to machines in increasing order. For any machine i, m-TOPO places nodes until a machine is full or the total number of nodes placed so far is $\geq \frac{in}{m}$, where n is the total number of nodes in the graph.

4.2 ETF

4.2.1 Algorithm description

The existing earliest task first (ETF) algorithm [37] schedules the earliest schedule-able task first, and the process is repeated until all tasks are scheduled. The algorithm maintains a task list I with unscheduled tasks and a machine list P with the earliest time a machine becomes available. The next available time of each machine is denoted by $free(p)$. The algorithm

1. computes the earliest schedule-able time for all tasks in I,
2. selects the task with the smallest earliest schedule-able time,
3. schedules the task from b on the machine where it can begin at the earliest.

For any task i, let s_i be the starting time and p_i be the computation time. Let $\Gamma^-(i)$ be the set of all immediate predecessors of task i. For any edge $i \to j \in E(G)$, let c_{ij} be the communication delay between tasks i and j. c_{ij} is only valid when tasks i and j are on different machines. Denote x_{ip} to be 0 when task i is on machine p and 1 otherwise. The earliest schedule-able time for any task j (a) is $\min_{p \in P} \left[\max \left(free(p), \max_{i \in \Gamma^-(i)} (s_i + p_i + c_{ij} x_{ip}) \right) \right]$. Under the small communication time (SCT) assumption, ETF has approximation ratio $2 + \rho - \frac{1}{m}$, which tends to 3 when ρ approaches 2 and m is large. A detailed description can be found in [37].

4.2.2 Memory constrained version

In our modified algorithm m-ETF, at each step we sort the task machine pair (t, p) according to the task t's earliest schedule-able time on machine p. Then we examine task machine pairs in the sorted order, until we find (t, p) where adding task t to machine p will not result in $p's$ memory being overloaded.

4.3 SCT

4.3.1 Algorithm description

The existing SCT algorithm [17] is similar to ETF [37], but prioritizes:

1. Scheduling tasks together that are also scheduled together in the infinite machine variation of the problem, as described in the following section.

2. Scheduling an urgent task. An urgent task at time t is one that can be scheduled on any idle machine to begin at time t. Such a task has already been delayed scheduling and should not be further ignored.

When the above prioritizing scheme results in better approximation ratio with the SCT assumption. We describe below the algorithm in detail.

Infinite machine algorithm For the infinite number of machines case, the SCT assumption makes it possible to model the problem as an integer linear programming (ILP) with a meaningful linear programming (LP) relaxation. As discussed in the Reflection chapter (chapter 8), not all ILPs have meaningful LP relaxations.
The SCT assumption ensures that; for each task i, it is advantageous to schedule only one immediate successor j on the same machine as i. Scheduling two successors of i on the same machine as i is not optimal because the second task could have started earlier on a new machine. For any task i, a favorite child $f(i)$ denotes the preferred successor of i that is scheduled on the same machine as i.
The ILP is formulated as follows,

$$
\begin{cases}
\min w^{\infty} & \text{Minimize makespan } w^{\infty} \\
\forall i \to j \in E(G), x_{ij} \in \{0,1\} & x_{ij} = 0 \text{ when j is i's favorite child} \\
\forall i \in V(G), s_i \geq 0 & \text{All tasks start after time=0} \\
\forall i \in V(G), s_i + p_i \leq w^{\infty} & \text{all tasks should complete before makespan} \\
\forall i \to j \in E(G), s_i + p_i + c_{ij}x_{ij} \leq s_j & \text{Given edge } i \to j\text{, j must start after i completes. If on different machines, communication cost should be added} \\
\forall i \in V(G), \sum_{j \in \Gamma^+(i)} x_{ij} \geq |\Gamma^+(i)| - 1 & \text{Every node has at most 1 favorite child} \\
\forall i \in V(G), \sum_{j \in \Gamma^-(i)} x_{ij} \leq |\Gamma^-(i)| - 1 & \text{Every node is the favorite child of at most 1 predecessor}
\end{cases}
\tag{4.1}
$$

The ILP modeled above can be relaxed to LP by allowing x_{ij} to take any real value

between 0 and 1. This LP can be solved in polynomial time using the interior point method [40]. Then the SCT algorithm simply rounds the LP solution x_{ij} to be 1 if $x_{ij} \geq 0.5$ and 0 otherwise. It can be easily verified that the rounded solution complies with all constraints stated above. x_{ij} can be used to determine the favorite child of each task. j is i's favourite child if and only if $x_{ij} = 1$.

This infinite machine algorithm achieves an approximation ratio $\frac{2+2\rho}{2+\rho}$, as described in [17].

Finite machine algorithm The favourite child determined by the infinite machine case alongside the urgent task are used as priorities. Then priorities are incorporated into the base SCT algorithm to determine the final placement. In order to understand the algorithm, let's consider a partially completed schedule S at an arbitrary time t during the execution. Let i be the last task scheduled on machine p. For each remaining unscheduled task j, denote the machine on which j can start the earliest as $ep(j)$. Let's define a machine p to be *free* at time t if machine p is available at time t and it is possible to schedule i's favourite child $f(i)$ earlier on a different machine. In other words, $ep\big(f(i)\big) \neq p$. Let's define a machine p to be *awake* at time t if it is favourable to schedule i's favourite child on p. In other words, $ep\big(f(i)\big) = p$.

The algorithm maintains a list P of machines in the order of their earliest available time. Let's consider a machine p that is available at time t. If p is *free* at t, the algorithm schedules the earliest task available on p. If an urgent task is available, it will naturally also be the earliest task. If p is *awake*, the algorithm schedules an urgent task on p if there is an urgent task in I at time t, otherwise it schedules the favorite child $f(i)$ of i on p.

The algorithm repeatedly finds the next moment when a machine is available t, and schedules task on available machines as described above, until all tasks are scheduled.

If a task i and its favorite child $f(i)$ are scheduled together in the infinite machine algorithm, it results in a very good approximation ratio. This intuitively shows that $f(i)$ has a bigger influence on the start time of future tasks than i's other successors and thus it is advantageous to schedule i and $f(i)$ together even in the finite machine case. [17] proves that the SCT algorithm gives a better approximation ratio than ETF, as expected. The SCT approximation ratio is $\frac{4+3\rho}{2+\rho} - \frac{2+2\rho}{m(2+\rho)}$ approximation ratio, which tends to $\frac{7}{3}$ when ρ approaches 1 and m is large, while the ETF approximation ratio approaches 3 (as described in section 4.2). See [17] for detail of the proof of the approximation ratio.

4.3.2 Memory constrained version

For our memory constrained version of the SCT algorithm (m-SCT), we maintain the same priority scheme as the finite case SCT algorithm. To enforce the memory constraint, we assume each machine has the same amount of memory M. Let $K = \frac{mM}{\sum_{i=1}^{n} d_i}$, where m is the total number of machines, n is the number of nodes in graph G and for any node i in G, d_i is the size of memory required by i. Intuitively, K is the ratio of the total memory available from all machines to the total memory required by the model. When the memory M on a machine is exceeded, we drop it from the list of available machines for the duration of the algorithm. We prove below that m-SCT approximates the optimal solution within $(1 + \frac{2+2\rho}{(2+\rho)m}) \cdot \frac{1}{K-1}$ of the finite SCT's approximation ratio.

Theorem 4.1 Let the approximation ratio given by the finite SCT algorithm be α, then m-SCT has approximation ratio $\leq \alpha + (1 + \frac{2+2\rho}{(2+\rho)m}) \cdot \frac{1}{K-1}$.

Proof:

Since $M = \frac{K}{m} \sum_{i=1}^{n} d_i$, from a total of m machines, at most $\frac{m}{K}$ machines would be full (hence dropped) at any time. Therefore, there are at least $\frac{(K-1)m}{K}$ machines not dropped throughout the algorithm.

Let's denote w^{∞} as the makespan given by the infinite SCT algorithm and w^{∞}_{OPT} as the optimal solution to the infinite machine variation of *scheduling with communication delay* problem. Let w^{m}_{OPT} be the optimal solution to the m machine finite *scheduling with communication delay* problem and w^{m}_{OPTm} be the optimal solution to the memory constrained m machine finite *scheduling with communication delay* problem. Then, $w^{\infty}_{OPT} \leq w^{m}_{OPT} \leq w^{m}_{OPTm}$.

Since at least $\frac{(K-1)m}{K}$ machines are always available for scheduling in m machine m-SCT, it generates a makespan T' at least as good as the one generated by finite SCT with $\frac{(K-1)m}{K}$ machines, T.

From [17], the m machine finite SCT algorithm has makespan W such that $W \leq \frac{1}{m} \cdot \sum_{i=1}^{n} p_i + (1 - \frac{1}{m}) w^{\infty}$. Therefore, the $\frac{(K-1)m}{K}$ machine finite SCT algorithm has makespan

T such that,

$$
\begin{aligned}
T &\leq \frac{1}{\frac{(K-1)m}{K}} \cdot \sum_{i=1}^{n} p_i + (1 - \frac{1}{\frac{(K-1)m}{K}}) w^{\infty} \\
&\leq \frac{K}{(K-1)m} \sum_{i=1}^{n} p_i + (1 - \frac{K}{(K-1)m}) w^{\infty} \\
&= \frac{K}{K-1} \frac{1}{m} \sum_{i=1}^{n} p_i + (1 - \frac{K}{(K-1)m}) w^{\infty} \\
&\leq \frac{K}{K-1} w_{OPT}^{m} + (1 - \frac{K}{(K-1)m}) w^{\infty}
\end{aligned}
$$

As described in section 4.3.1, the approximation ratio between w^{∞} and w_{OPT}^{∞} is $\beta = \frac{2+2\rho}{2+\rho}$. For the makespan T' generated by m machine m-SCT,

$$
\begin{aligned}
T' \leq T &\leq \Big(\frac{K}{K-1} + (1 - \frac{K}{(K-1)m}) \beta \Big) w_{OPTm}^{m} && (4.2) \\
&\leq \Big((\frac{1}{K-1} + \frac{\beta}{(K-1)m}) + 1 + (1 - \frac{1}{m}) \beta \Big) w_{OPTm}^{m} && (4.3)
\end{aligned}
$$

The m machine finite SCT algorithm in [17] has the approximation ratio $\alpha = 1 + (1 - \frac{1}{m})\beta$. Using equation (4.2), m-SCT has an approximation ratio $\alpha + (1 + \frac{2+2\rho}{(2+\rho)m}) \cdot \frac{1}{K-1}$.

4.4 OPTIMAL SOLUTION

We modified the infinite machine ILP (described in section 4.3.1) to incorporate the finite machine and memory constraints. [41] [42] [43] [44] show other similar attempts in the area. Through this section, let V be the set of all tasks and E be the set of all edges in G, the dependency graph.

4.4.1 Memory constraints

In this section, we discuss how to incorporate memory constraints in the infinite machine SCT constrained ILP formulation. In this ILP, no variable records which machine a task is placed on. However, for enforcing the memory constraint, we must record the tasks associated with each machine. Let y_{ip} be 1 if task i is on machine p and 0 otherwise. Let m_i be the size of memory that needs to be reserved for task i. Denote P as the set of all machines and M as the upper memory limit for any machine. Then, the memory constraint for each

machine can be modeled as follows,

$$\forall p \in P, \sum_{i=1}^{n} y_{ip} m_i \leq M \tag{4.4}$$

We need a variable x_{ij}, which describes whether task i and task j are on the same machine for our ILP. Let x_{ij} be 1 if i, j are scheduled on the same machine and 0 otherwise. Then $x_{ij} = 1$, if and only if for some machine p, $y_{ip}y_{jp} = 1$ (both i and j are on machine p). Therefore, we model x_{ij} as,

$$x_{ij} = \sum_{p \in P} y_{ip} y_{jp} \tag{4.5}$$

However, (4.4) is not linear and cannot be added to an ILP. We now attempt to linearize (4.4). Let's define $y_{ijp} = y_{ip}y_{jp}$. Now, x_{ij} and y_{ijp} have a linear relationship. Fortunately, since y_{ip} and y_{jp} are both boolean variables, we can further express y_{ijp}'s relationship with y_{ip} and y_{jp} using linear equations. Namely,

$$\begin{aligned} y_{ijp} &\geq y_{ip} + y_{jp} - 1 \\ y_{ijp} &\leq y_{ip} \\ y_{ijp} &\leq y_{jp} \\ y_{ijp} &\in \{0, 1\} \end{aligned} \tag{4.6}$$

This linear model ensures that y_{ijp} is true if and only if both i and j are on machine p, i.e., $y_{ijp} = 1$ when $y_{ip} = 1$ and $y_{jp} = 1$ and 0 otherwise. Combining the equations above, the following equations model the memory constraints in totality,

$$
\begin{cases}
\forall p \in P, \sum_{i=1}^{n} y_{ip} m_i \leq M & \text{Total memory on each machine} \leq \text{M} \\
\forall i, j \in V, x_{ij} = \sum_{p \in P} y_{ijp} & \text{Tasks i and j are on the same machine when for some machine p, both i and j are on machine p} \\
\forall i, j \in V, p \in P, y_{ijp} \geq y_{ip} + y_{jp} - 1 & \text{If both } y_{ip} \text{ and } y_{jp} = 1, \text{ then both task i and j are on machine p and } y_{ijp} = 1 \\
\forall i, j \in V, p \in P, y_{ijp} \leq y_{ip} & y_{ijp} = 1 \text{ only when i is on machine p} \\
\forall i, j \in V, p \in P, y_{ijp} \leq y_{jp} & y_{ijp} = 1 \text{ only when j is on machine p} \\
\forall i, j \in V, p \in P, y_{ijp} \in \{0, 1\} & \text{Whether both i and j are on machine p} \\
\forall i \in V, p \in P, y_{ip} \in \{0, 1\} & \text{Whether i is on machine p} \\
\forall i, j \in V, x_{ij} \in \{0, 1\} & \text{Whether i and j are on the same machine.}
\end{cases} \tag{4.7}
$$

4.4.2 Finite machine constraints

In this section, we model the required finite machine constraints with linear equations. Let's define tasks i and j to be *unrelated* when task i is not an ancestor of task j and task j is not an ancestor of task i. An important difference introduced by the finite machine constraint is that we may have to put unrelated tasks on the same machine because of the limited number of total machines. In the original infinite machine SCT constrained ILP, there is no constraints specifying the relationship of *unrelated* tasks' starting time, since they would never be placed on the same machine. Here, if two *unrelated* tasks are scheduled on the same machine, we must additionally ensure that the execution time of the tasks do not overlap. This can be modeled as an additional constraint,

$$
s_i + p_i \leq s_j \text{ OR } s_j + p_j \leq s_i \tag{4.8}
$$

where for any task i, s_i is the starting time of task i and p_i is the computation time of task i. Equation (4.5) ensures that either task i is fully executed before task j's starting time or vice versa. This is important because it is not possible to parallelly execute tasks on a single machine. It is possible to convert the above nonlinear constraint into a linear form. Let's define $b_{ij} = 0$ when task i executes before task j and 1 otherwise. Then, equation (4.5) is

equivalent to the following constraints,

$$s_i - s_j \leq -p_i + (U + p_i)b_{ij} \tag{4.9}$$

$$s_i - s_j \geq L + (p_j - L)b_{ij} \tag{4.10}$$

Intuitively, when $b_{ij} = 0$, then i executes before j, and (4.6) enforces that i's execution must finish before j's starting time. In this case, (4.7) is always true and does not interfere. When $b_{ij} = 1$, (4.7) similarly ensures that j's execution must finish before i's starting time. In this case, (4.6) is always true and does not interfere. We choose constants L and U to appropriately activate which constraint dominates in each case. One way to choose L and U is to set $L = -(\sum_{i \in V} p_i + \sum_{i \to j \in E} c_{ij})$ and $U = \sum_{i \in V} p_i + \sum_{i \to j \in E} c_{ij}$. Therefore, when $b_{ij} = 0$, i is completed before j starts.

We must only enforce constraints (4.6) and (4.7) when tasks i and j are on the same machine. This leads to a slight modification to constraints (4.6) and (4.7), as we incorporate x_{ij} (as defined in section 4.4.1), which is a boolean variable describing whether tasks i and j are on the same machine, as follows,

$$\forall (i,j) \notin E, s_i - s_j \leq -p_i + (U + p_i)(b_{ij} + 1 - x_{ij}) \tag{4.11}$$

$$\forall (i,j) \notin E, s_i - s_j \geq L(2 - x_{ij}) + (p_j - L)b_{ij} \tag{4.12}$$

4.4.3 Final ILP formulation

Summarizing the above constraints for finite memory and number of machines, and combining them with the infinite machine ILP (described in section 4.3.1), we arrive at the final ILP.

$\min w$	Minimize makespan
$\forall i \in T, s_i \geq 0$	All tasks start sometime after 0
$\forall (i,j) \in E, x_{ij} \in \{0,1\}$	Whether i and j are on same machine
$\forall i \in T, s_i + p_i \leq w$	All tasks complete before makesapn
$\forall (i,j) \in E, s_i + p_i + (1 - x_{ij})c_{ij} \leq s_j$	j must start after i completes and if they are on different machines communication cost should be added

$$
\begin{cases}
\forall i \in V, p \in P, y_{ip} \in \{0,1\} & \text{Whether i is on machine p} \\
\forall i,j \in V, p \in P, y_{ijp} \in \{0,1\} & \text{Whether both i and j are on machine p} \\
\forall i \in V, \sum_{p \in P} y_{ip} = 1 & \text{Each task should be scheduled on exactly 1 machine} \\
\forall i,j \in V, p \in P, y_{ijp} \geq y_{ip} + y_{jp} - 1 & \text{If both } y_{ip} \text{ and } y_{jp} = 1, \text{ then both i and j are on machine p and } y_{ijp} = 1 \\
\forall i.j \in V, p \in P, y_{ijp} \leq y_{ip} & y_{ijp} = 1 \text{ only when i is on machine p} \\
\forall i,j \in V, p \in P, y_{ijp} \leq y_{jp} & y_{ijp} = 1 \text{ only when j is on machine p} \\
\forall i,j \in V, x_{ij} = \sum_{p \in P} y_{ijp} & \text{Tasks i and j are on the same machine when for some machine p, both i and j are on machine p} \\
\forall p \in P, \sum_{i=1}^{n} y_{ip} s_i \leq M & \text{Total memory used on each machine} \leq \text{M} \\
\forall (i,j) \notin E, s_i - s_j \leq -p_i + (U + p_i) b_{ij} + U(1 - x_{ij}) & \text{Enforces that when } b_{ij} = 0, \text{ i finishes before j starts, given that i, j are on the same machine } (x_{ij} = 0). \text{ If i and j are on different machines, constraint becomes meaningless and degrades to their starting time difference being less than an arbitrary large number } U \\
\forall (i,j) \notin E, s_i - s_j \geq L + (p_j - L) b_{ij} + L(1 - x_{ij}) & \text{Enforces that when } b_{ij} = 1, \text{ j finishes before i starts, given that i, j are on the same machine } (x_{ij} = 0). \text{ If i and j are on different machines, constraint becomes meaningless and degrades to their starting time difference being larger than an arbitrary small number } L
\end{cases}
\tag{4.13}
$$

CHAPTER 5: BACK PROPAGATION

In this chapter, we discuss how to account for back propagation in the algorithms proposed in chapter 4 and prove certain makespan and approximation ratio guarantees for it.

5.1 DEFINITION AND NEED FOR BACK PROPAGATION

Definition In neural network training, the back propagation algorithm (a) propagates the total error backwards through the Neural Network (NN) layers (b) computes the gradient of each weight and bias in the network, using the propagated error (c) applies the gradients calculated in (b) to their corresponding weights and biases (d) eventually minimizes the total error of the NN.

Need for BP The back propagation algorithm is needed to train multi-layer NNs. It corrects the weights and biases of every layer in the NN to reduce the error in each iteration of the training process.

5.2 ACCOUNTING FOR BACK PROPAGATION

The scheduling algorithms discussed in chapter 3 are only valid for directed acyclic graphs. The process of backpropagation in NN training introduces some cycles and is computationally intensive. Since BP is non trivial and accounts for a significant chunk of the makespan, we account for it separately.

5.3 SCHEDULING STRATEGY

Each *operation* node in TensorFlow (TF) has an associated *gradient* node that is responsible for the calculation of gradient for that operation. Similarly, each *variable* has an associated *GradientDescent* node whose primary purpose is the application of gradient and updating the variable. Together, these two nodes are primarily responsible for the BP process. Between these two functionalities, gradient computation accounts for the majority of the complexity.

If we collocate the *gradient* nodes with their corresponding *operation* nodes, we discover that the resulting backward dependency graph G' is the reverse of the forward propagation graph G, i.e., same vertices and reverse edges. TF provides a very straightforward interface to

specify these collocations. Unlike the *gradient* nodes, which must wait for their predecessors from G' before executing, the *GradientDescent* nodes have no such dependencies. The gradient application can be carried out at any time in the BP process after the gradient has been calculated for a node, including at the end. Given the lack of *GradientDescent* node inter-dependencies and their less intensive nature, their contribution to the makespan can be safely ignored for the purposes of our proofs. Their withdrawal from consideration serves another critical purpose. Without the *GradientDescent* nodes, the connections between the forward and backward computation graphs (G and G') are severed and the entire NN graph can be considered a DAG. Under this assumption, our previously discussed algorithms from chapter 4 can be successfully used.

5.4 GUARANTEES

Given the assumptions stated above, we prove certain guarantees for the BP phase in the remainder of this chapter. First, we show that the makespan of the backward pass is within C_0 (as defined in section 5.4.1) times the makespan of the forward pass. Next, we prove that if C (as defined in section 5.4.1) is constant across the NN, the initially established approximation ratio remains unchanged after back propagation. It is worth noting that while the makespan guarantee only holds if the *gradient* nodes are collocated with operations, in practice, it is possible to apply DAG algorithms simply after the removal of the *GradientDescent* nodes.

5.4.1 Preliminaries

In this subsection, we define a schedule and enumerate the necessary and sufficient conditions for a schedule to be legal.

Schedule We define a schedule S for graph G as a list that associates each node in G with a machine p on which it will be executed and a starting time s_i.

Backward Schedule Denote the makespan of schedule S as T. We define forward schedule S's corresponding backward schedule S' as a schedule where for any node i in G, (a) i is associated with the same machine as in S (b) i has starting time $s'_i = C_0(T - s_i - p_i)$, where s_i is i's starting time in S.

Legal Schedule Let's denote the computation time of node i in dependency graph G as p_i and the communication time between any two tasks i and j in G to be c_{ij}.
Schedule S is legal if and only if,

1. $s_i \geq 0$
2. $s_i + p_i \leq s_j$ or $s_j + p_j \leq s_i$ when task i and task j are on the same machine. This guarantees that tasks i and j do not overlap.
3. $s_i + p_i \leq s_j$ when task i and task j are on the same machine and there is an edge $i \to j$ in the dependency graph G. This guarantees that the precedence relationship between tasks i and j is honored, when they are on the same machine.
4. $s_i + p_i + c_{ij} \leq s_j$ when task i and task j are on different machine, and there is an edge $i \to j$ in the dependency graph G.cThis guarantees that the precedence relationship between tasks i and j is honored, when they are on different machines.

Backward dependency graph The backward dependency graph G' is the reverse of the forward propagation graph G, i.e., G' shares the same vertices with G but has reversed edges. We denote the computation time of a node j in G' as p'_j and the communication time between any nodes j and i in G' as c'_{ji}.

NN proportionality constant We define the NN proportionality constant C_0 for forward graph G and backward graph G' as the maximum ratio between (a) corresponding backward pass edge weight and forward pass edge weight and (b) corresponding backward pass node weight and forward pass node weight.

$$C_0 = max\Big(max_{j \in V(G)} \frac{p'_j}{p_j}, max_{i \to j \in E(G)} \frac{c'_{ji}}{c_{ij}}\Big)$$

NN proportionality function We define the NN proportionality function C as follows (a) For any node $i \in V(G)$, let $C(i) = \frac{p'_i}{p_i}$ (b) for any edge $i \to j \in E(G)$, let $C(i \to j) = \frac{c'_{ji}}{c_{ij}}$. We say function C is constant if for all $i \in V(G), C(i) = C_0$ and for all $i \to j \in E(G)$, $C(i \to j) = C_0$. Essentially, C_0 is the max over all values of C.

5.4.2 Theorems

Theorem 5.1 Given a schedule S on G, S's corresponding backward schedule S' is legal and $makespan(S') \leq C_0 \cdot makespan(S)$

Proof We know that $\forall i \to j \in E(G), c'_{ji} \leq C_0 c_{ij}$ and $\forall j \in V(G), p'_j \leq C_0 p_j$
Denote the makespan of schedule S as T.
We prove below that the schedule S' is legal:

1. We know $T \leq s_i + p_i \forall i$
   Therefore $s'_i = C_0(T - s_i - p_i) \geq 0 \ \forall i \in V(G)$

2. Let i, j be 2 arbitrary tasks assigned to the same machine.
   Since S is a legal schedule, we know that $s_i + p_i \leq s_j$ or $s_j + p_j \leq s_i$.
   $T - s_i - p_i \geq T - s_j - p_j + p_j$ or $T - s_j - p_j \geq T - s_i - p_i + p_i$.
   $C_0(T - s_i - p_i) \geq C_0(T - s_j - p_j) + C_0 p_j$ or $C_0(T - s_j - p_j) \geq C_0(T - s_i - p_i) + C_0 p_i$.
   $s'_i \geq s'_j + p'_j$ or $s'_j \geq s'_i + p'_i$.

3. Let $j \to i$ be an arbitrary edge in G' and i, j are assigned to the same machine in S.
   $i \to j \in E(G) <=> j \to i \in E(G')$.
   Since S is a legal schedule, we know that $i \to j \in E(G)$, $s_i + p_i \leq s_j$.
   $C_0(T - s_i - p_i) \geq C_0(T - s_j - p_j) + C_0 p_j$.
   $s'_i \geq s'_j + p'_j$ in G'.

4. Let $j \to i$ be an arbitrary edge in G' and i, j are assigned to different machines.
   $i \to j \in E(G) <=> j \to i \in E(G')$
   Since S is a legal schedule, we know that $i \to j \in E(G)$, $s_i + p_i + c_{ij} \leq s_j$.
   $C_0(T - s_i - p_i) - C_0 c_{i \to j} \geq C_0(T - s_j - p_j) + C_0 p_j$.
   $C_0(T - s_i - p_i) \geq C_0(T - s_j - p_j) + C_0 p_j + C_0 c_{ij}$.
   $s'_i \geq s'_j + p'_j + c'_{ji}$ in G'.

S' satisfies all condition for a legal schedule and therefore is legal.
$makespan(S') = max_i(s'_i + p'_i) = max_i(C_0(T - s_i)) \leq C_0 T = C_0 \cdot makespan(S)$

Theorem 5.2 When NN proportionality function C is constant, the makespan of optimal schedule for G' is C_0 times the makespan for optimal schedule for G.

Proof We know that $\forall j \in V(G'), \frac{p'_j}{p_j} = C_0$, and $\forall j \to i \in E(G'), \frac{c'_{ji}}{c_{ij}} = C_0$.
Therefore, $max\Big(max_{j \in V(G')} \frac{p_j}{p'_j}, max_{j \to i \in E(G')} \frac{c_{ij}}{c'_{ji}}\Big) = \frac{1}{C_0}$.
Let OPT be the shortest makespan for any schedule on G with m machines. Let S^* be an optimal schedule for G.

Let OPT' be the shortest makespan for any schedule on G with m machines. Let S'^* be an optimal schedule for G'.

By Theorem 5.1 we know that there is a schedule S' for G' where $makespan(S') \leq C_0 makespan(S^*)$, therefore $OPT' \leq makespan(S') \leq C_0 makespan(S^*) = C_0 OPT$.
For the same reason $OPT \leq \frac{1}{C_0} OPT'$.
Therefore $OPT' = C_0 OPT$.

Theorem 5.3 When NN proportionality function C is constant, given a schedule S for G, the backward schedule S' has the same approximation ratio as S.

Proof By Theorem 5.2, the makespan of optimal schedule for G' OPT' is C_0 times the makespan for optimal schedule OPT for G.

By Theorem 5.1, $makespan(S') \leq C_0 makespan(S)$ and $makespan(S') \leq \frac{1}{C_0} makespan(S)$.
$makespan(S') = C_0 makespan(S)$.

Therefore, $\frac{makespan(S')}{OPT'} = \frac{makespan(S)}{OPT}$, which means the approximation ratio of S' equals the approximation ratio of S.

CHAPTER 6: IMPLEMENTATION

In this chapter, we provide implementation and simulation details of our selected algorithms. We resort to using the simulation technique as it allows for easy comparison of algorithms without committing to any framework and being influenced by the idiosyncrasies of any specific implementation.

6.1 SIMULATION

We implement the algorithms discussed in chapter 3 in NetworkX [45], a popular open source Python package for creation and manipulation of graphs. In order to compare all the algorithms and demonstrate the difference in their efficiencies effectively, we simulate distributed graph partitioning. We test our selected algorithms on Inception-V3 [46] and a small custom Convolution Neural Network (CNN) [47] graph model (described in greater detail in chapter 6). These graphs are good representations of commonly used models in Machine Learning (ML) frameworks, such as TensorFlow. For the simulation, first, we realistically estimate the computation time of each node and the inter node communication times. Then, we create a new graph in NetworkX where each node corresponds to a computation task from the selected model (with its weight being the computation time) and each edge corresponds to the dependency between these tasks (with the edge weight being the inter node communication time). Finally, we execute the selected algorithm on this graph, which yields the placement and the corresponding makespan.

6.2 ESTIMATION OF COMPUTATION TIME

To estimate the computation time we input a model into TensorFlow and generate the *graphdef*, an internal representation of the TensorFlow (TF) graph and part of its core framework (as described in section 2.1). The *graphdef* is then profiled using a TF python client, *timeline*, which outputs a JSON file. The JSON file details the computation time of each TF node as a *dur* argument as can be seen from figure 5.1. In order to avoid the initialization costs, we use the computation data from the 50th iteration and experimentally verify that the computation time for further iterations is approximately the same. The nodes which do not have any computation time associated with them and exist merely for the purposes of control flow, do not appear in the profiling. We manually add these nodes with zero cost in the simulation. The *timeline* visualization for our CNN model is presented

in figure 5.2.

It's worth noting that the TF graph is different from the logical graph as it includes nodes for variables, operations and several other ancillary nodes like read/write (as detailed in section 2.1). The use of the TF graph is preferable to the logical one as it gives substantial insight into how Machine Learning frameworks break down logical operations and establish the control flow. Simulation using this real world breakdown of nodes is more likely to yield realistic results that can be extrapolated to other existing Machine Learning systems as well.


```
{
    "name": "Const",
    "args": {
        "name": "adam_optimizer/gradients/fc1/add_grad/Shape_1",
        "op": "Const"
    },
    "pid": 1,
    "ts": 1524887406429175,
    "cat": "Op",
    "tid": 0,
    "ph": "X",
    "dur": 3
},
```


Figure 6.1: Timeline [8] output for a variable with computation time of 3ms

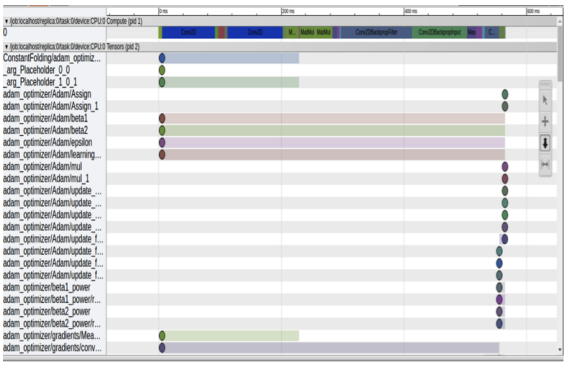

Figure 6.2: Timeline [8] visualization

6.3 ESTIMATION OF COMMUNICATION TIME

The inter node communication time factors significantly in all our scheduling algorithms. To estimate the communication time effectively, if two nodes are on the same machine, we assume communication time to be zero. This assumption is justified because communication over a network is many times more expensive than within the same machine. For all other cases, first, the total size of the data to be communicated is determined using equation (5.1). We obtain the tensor dimensions in a similar way as the computation times, using the *timeline* object. The JSON file obtained in this case provides the *tensor_description* argument which lists the dimensions of the tensor as well as the total bytes requested by the system for that tensor (based on its datatype) as shown in figure 5.3.

$$\text{Size of tensor} = \text{dim1} \cdot \text{dim2} \cdot \text{dim3} \tag{6.1}$$

```
        {
            "name": "_arg_Placeholder_0_0",
            "args": {
                "snapshot": {
                    "tensor_description": "dtype: DT_FLOAT\nshape {\n  dim {\n    size: 100\n  }\n  dim {\n    size: 784\n
}\n}\nallocation_description {\n  requested_bytes: 313600\n  allocator_name: cpu\n}\n"
                }
            },
            "pid": 2,
            "ts": 1524889506021756,
            "cat": "Tensor",
            "tid": 0,
            "ph": "O",
            "id": 1
        },
```


Figure 6.3: Timeline [8] output for tensor of dimensions 100,784 requesting 313600 bytes of memory

Once the size of tensor is determined and we assume a suitable network bandwidth, the communication time is calculated based on equation 5.2. We use a variety of bandwidths in our experiments in order to see how performance varies as the data centre bandwidth varies. Greater bandwidths imply smaller communication time, and help comply with the small communication time (SCT) assumption better (see section 4.3.2).

$$\text{Communication time} = \text{size/bandwidth} \tag{6.2}$$

Sometimes in TF, the size of the tensor is dynamically determined at run-time. Our models do not use any dynamic tensors as they are not conceptually different from the statically sized ones. Different techniques may be required to extract the size information for these cases in the future.

6.4 CONSTRAINTS

In the simulation, we introduce memory constraints on every processor and perform experiments using a fixed number of processors for each case. We vary the number of processors across experiments to note the makespan and partitioning trends as the number of processors change for different algorithms.

Unlike our simulation, TF has arbitrary rules about collocation of certain nodes on the same machine. These rules help assist the particular implementation architecture of TF and make it more productive. This is especially true in the case of backpropagation. Gradient nodes are commonly collocated with the original operation nodes. However, in our implementation, we do not explicitly collocate nodes and allow the algorithms to place them as

they deem fit. In the future, if TF experiments are carried out on similar models, some adjustments must be made to accommodate the collocation constraints.

6.5 LINEAR PROGRAMMING SOLVER

We use interior point method to solve the linear programming (LP) problems resulting from our algorithm. This method is preferred over other solvers such as simplex [48] because it guarantees polynomial execution time [49]. Specifically, we use the primal dual interior-point solver in Mosek optimization software [50], which has a run time complexity of $O(n^{3.5}L)$, where L is the maximum number of bits in the LP inputs (in our case 64, the number of bits in a python floating point number).

Experimentally, Mosek interior point solver is very fast, even for large graphs such as Inception-v3. For the Inception-v3 experiments in chapter 7, the Mosek interior point solver solves each LP in 3-10 seconds (see more details in chapter 7).

CHAPTER 7: EXPERIMENTAL RESULTS

In this chapter, we compare and contrast the performance of our selected algorithms on Inception-V3 and the small CNN model, and present our experimental results. For all our experiments, we assume a high speed data-center network (varying bandwidths) with no packet loss. We carry out the experiments on a Google VM machine (n1-standard-4 machine with 4 vCPUs and 15 GB memory).

The experiments are divided into two major categories; the fixed total memory experiments, where the total memory in the system remains constant regardless of the number of machines used, and the variable total memory experiments, where each machine has a fixed amount of memory and the overall memory of the system increases as the number of machines increase.

7.1 FIXED TOTAL MEMORY EXPERIMENTS

For the experiments in this section, we fix the total memory, which is distributed evenly across machines. As the number of machines increase, the memory per machine decreases.

7.1.1 Experiment on small CNN

Summary In this experiment, we compare m-TOPO, m-ETF and m-SCT's makespan for the small Convolutional Neural Network (CNN) graph. We present the plots for these cases in Figure 7.1 to Figure 7.4.

Specifications We fix the total memory across all machines to be 3 times the memory required by the CNN, distributed evenly across machines. We vary the bandwidths (1E5, 1E7, 1E9 and 1E11 bytes/second) and the number of machines (3, 6, 9, 12, 15) for all three alogorithms.

Trends and discussion In the first three cases (Figure 7.1 to 7.3), the makespans of m-SCT and m-ETF algorithms display a gradually increasing trend as the number of machines increase. This behaviour can be attributed to the fact that the small CNN model is not very parallizable and the memory constraint becomes tighter as the number of machines increase.

As the bandwidth increases from 1E5 to 1E9 bytes/second (Figure 7.1 to 7.3), makespan decreases for both m-ETF and m-SCT. These algorithms perform significantly better with

increasing bandwidths because it leads to smaller inter-node communication times for them and both of them have a constant approximation ratio under the small communication time (SCT) assumption.

Across all cases, m-ETF and m-SCT steadily outperform m-TOPO, except for a few configurations where m-TOPO chances upon an optimal placement.

In the first three plots, m-SCT has more consistent performance than m-ETF because it prioritizes the placement of favorite children and urgent tasks. Under tight memory constraints, like in this experiment, the effect of these optimizations is very pronounced. Very few tasks can fit on a single machine and m-SCT chooses these more carefully than m-ETF. In the last case (Figure 7.4), the 1E11 bytes/second bandwidth is so large for the small CNN graph that the communication time is negligible. Therefore, the differences between the performances of m-ETF and m-SCT are not very significant.

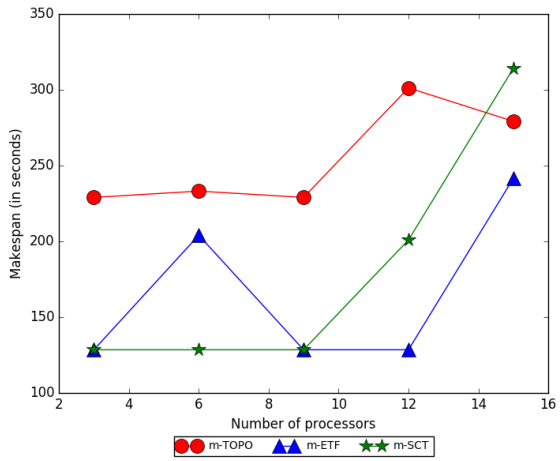

Figure 7.1: Makespan for small CNN with fixed total memory - 1E5 bytes/second

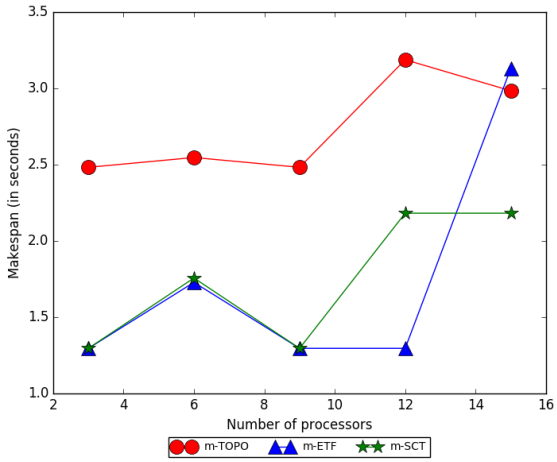

Figure 7.2: Makespan for small CNN with fixed total memory - 1E7 bytes/second

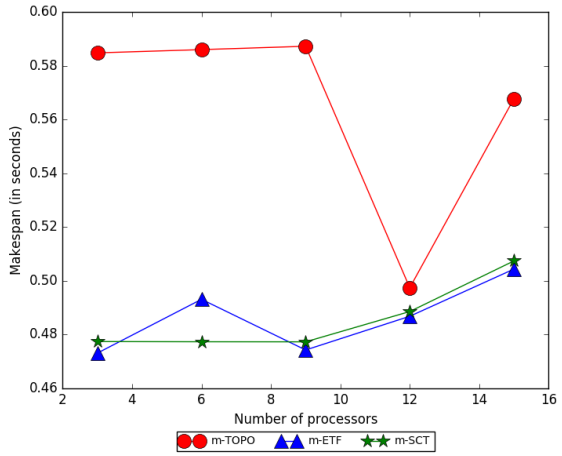

Figure 7.3: Makespan for small CNN with fixed total memory - 1E9 bytes/second

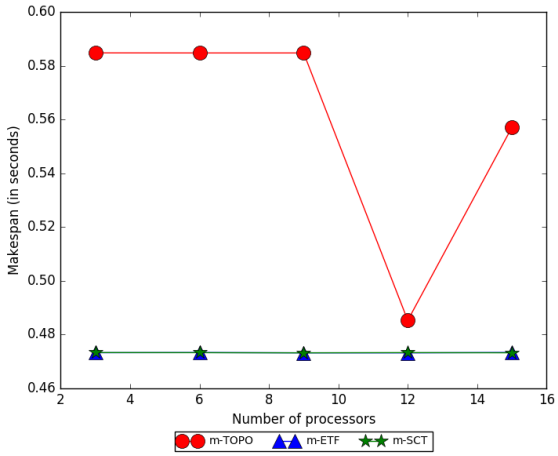

Figure 7.4: Makespan for small CNN with fixed total memory - 1E11 bytes/second

Fixed total memory on small CNN

7.1.2 Experiment on Inception-V3

Summary In this experiment, we compare m-TOPO, m-ETF and m-SCT's makespan for the Inception-V3 model. We present the plots for these cases in Figure 7.5 to Figure 7.8.

Specifications We fix the total memory across all machines to be 3 times the memory required by the Inception-V3 model. We vary the bandwidths (1E7, 1E9, 1E10 and 1E11 bytes/second) and the number of machines (3, 6, 9, 12, 15) for all three algorithms.

Trends and discussion For the E7 bytes/second bandwidth case (Figure 7.5), m-SCT performs poorly because the because the bandwidth is very low and the SCT assumption is significantly violated (as communication times become larger due to low bandwidth). As the bandwidth increases in the subsequent cases (Figure 7.6 to 7.8), m-SCT behaves better and outperforms both m-ETF and m-TOPO.

Across all cases (Figure 7.5 to 7.8), the makespan of m-SCT dips to a low point and rises afterwards. As the number of machines increase, the availability of free machines that can execute a ready task also increases. In this scenario, there are fewer than usual *urgent* tasks that are waiting to be executed and the m-SCT algorithm loses some of its advantage over m-ETF. For a large number of processors, the m-ETF and m-SCT graphs almost converge. For bandwidths other than 1E7 bytes/second (Figure 7.6 to 7.8) m-SCT and m-ETF out-

perform m-TOPO, similar to the CNN experiment.

We notice that the total memory constraint in this experiment (three times the required memory) is not very high. We expect that if this constraint is tightened further, the performance difference between m-ETF and m-SCT will become more pronounced.

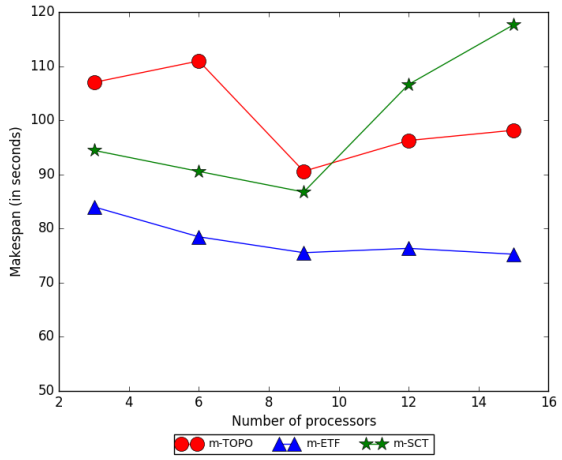

Figure 7.5: Makespan for Inception-V3 - fixed total memory - 1E7 bytes/second

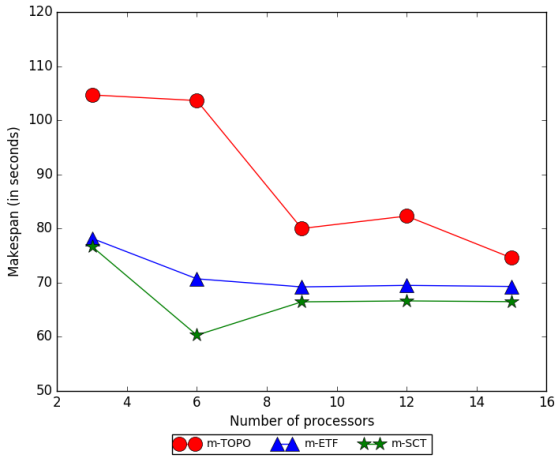

Figure 7.6: Makespan for Inception-V3 - fixed total memory - 1E9 bytes/second

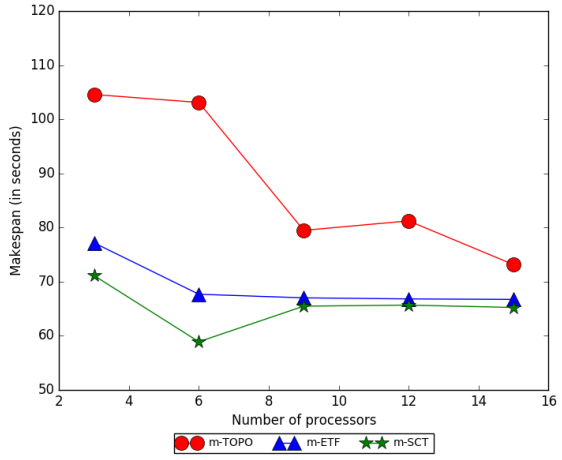

Figure 7.7: Makespan for Inception-V3 - fixed total memory - 1E10 bytes/second

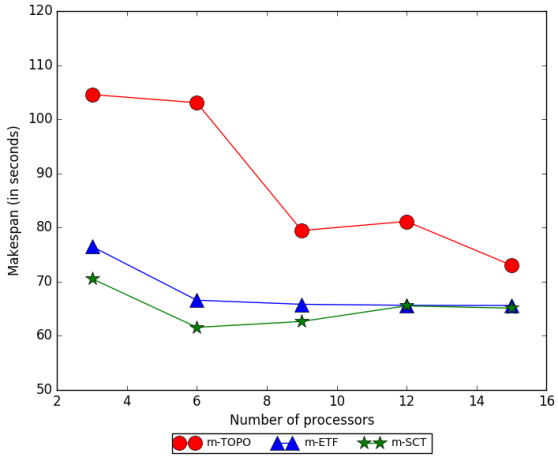

Figure 7.8: Makespan for Inception-V3 - fixed total memory - 1E11 bytes/second

Fixed total memory on Inception-V3

7.2 VARIABLE TOTAL MEMORY EXPERIMENTS

For the experiments in this section, we fix the total memory on each machine. The total memory in the system increases as the number of machines increase.

7.2.1 Experiment on small CNN

Summary In this experiment, we compare m-TOPO, m-ETF and m-SCT's makespan for the small CNN graph. We present the plots for these cases in Figure 7.9 to Figure 7.12.

Specifications We fix the memory of an individual machine as 75 MB. We vary the bandwidths (1E5, 1E7, 1E9 and 1E11 bytes/second) and the number of machines (3, 6, 9, 12, 15) for all three algorithms.

Trends and discussion Across all cases (Figure 7.9 to 7.12), m-ETF and m-SCT outperform m-TOPO, as expected.

Both m-SCT and m-ETF display a generally decreasing trend as the number of machines increase. The performance of m-ETF improves significantly with the increase in the number of machines as the tight memory constraint (from fewer machines) penalizes m-ETF heavily for not smartly prioritizing the placements of any tasks. For high number of machines, the effects of prioritizing become less prominent as machines become less crowded and many related tasks can easily fit on the same machine without making an extra effort to schedule them together. This behaviour is observed for all bandwidths.

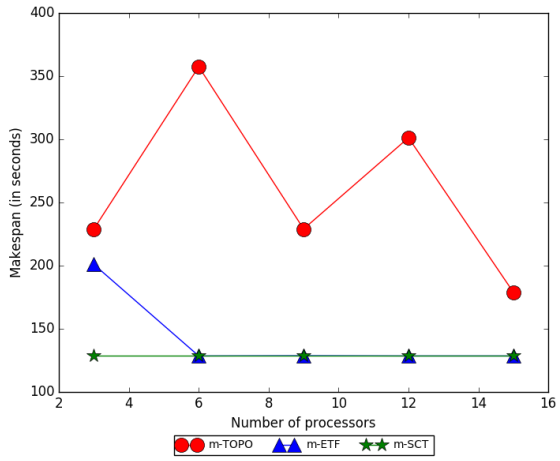

Figure 7.9: Makespan for small CNN 75MB RAM/machine - 1E5 bytes/second

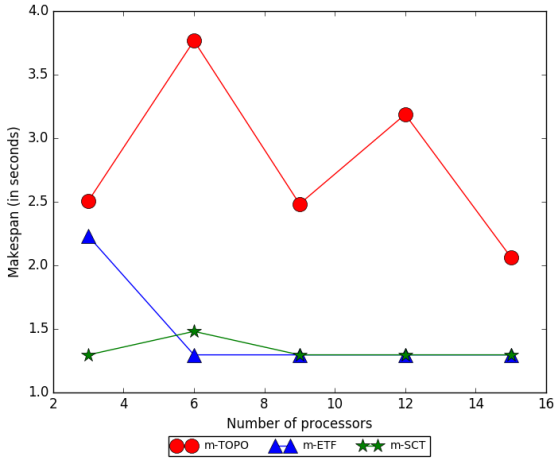

Figure 7.10: Makespan for small CNN 75MB RAM/machine - 1E7 bytes/second

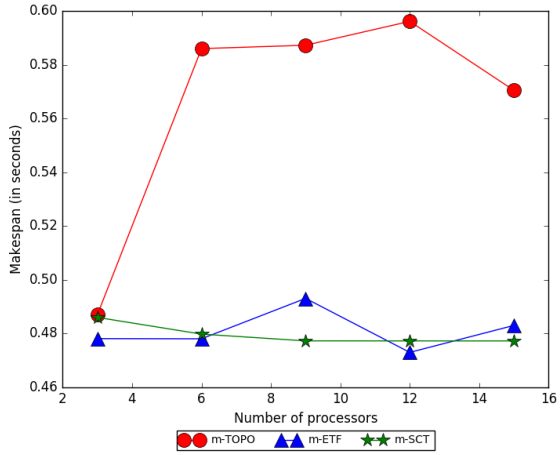

Figure 7.11: Makespan for small CNN 75MB RAM/machine - 1E9 bytes/second

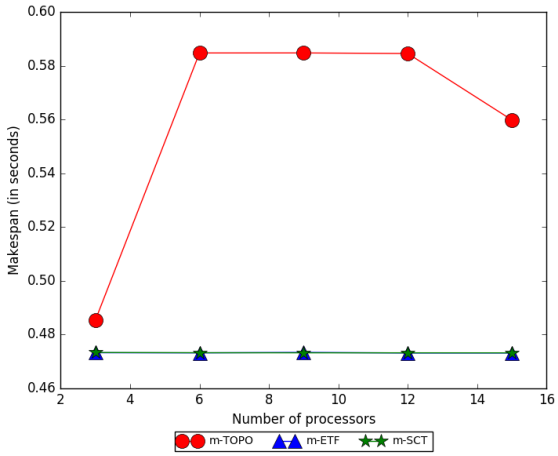

Figure 7.12: Makespan for small CNN 75MB RAM/machine - 1E11 bytes/second

Variable total memory on small CNN

7.2.2 Experiment on Inception-V3

Summary In this experiment, we compare m-TOPO, m-ETF and m-SCT's makespan for the Inception-V3 graph. We present the plots for these cases in Figure 7.13 to Figure 7.16.

Specification We fix the memory of an individual machine as 16 GB. We vary the bandwidths (1E7, 1E9, 1E10 and 1E11 bytes/second) and the number of machines (3, 6, 9, 12, 15) for all three algorithms. The Inception-V3 model is tested on a subset of imagenet (flower).

Trends and discussion Across all cases (Figure 7.9 to 7.12), m-ETF and m-SCT outperform m-TOPO, as expected.

Both m-SCT and m-ETF display a generally decreasing trend. This behaviour is very similar to the CNN experiment (section 7.2.1). The imagenet dataset can result in some very large tensor sizes for Inception-V3. When the bandwidth is as low as 1E7 bytes per second (Figure 7.13), the SCT assumption is severely violated and the priorities determined by m-SCT are not very accurate. In this case, m-ETF performs better than m-SCT because (a) the steadily increasing memory benefits m-ETF, as explained in section 7.2.1. (b) m-SCT uses inaccurate priority.

As the bandwidths increase (Figure 7.13 to 7.16), the SCT assumption becomes more

accurate and m-SCT starts outperforming m-ETF as expected.

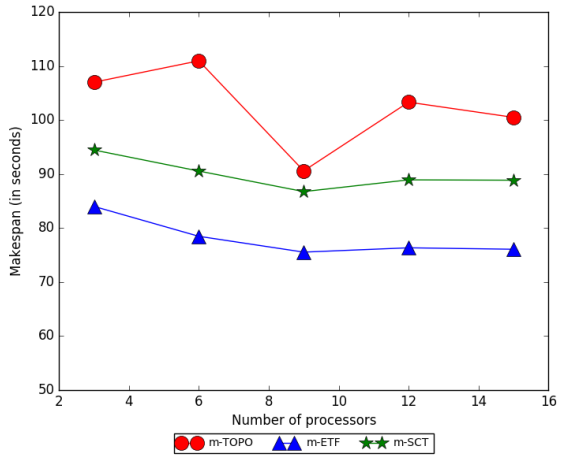

Figure 7.13: Makespan for Inception-V3 - 16GB RAM/machine - 1E7 bytes/second

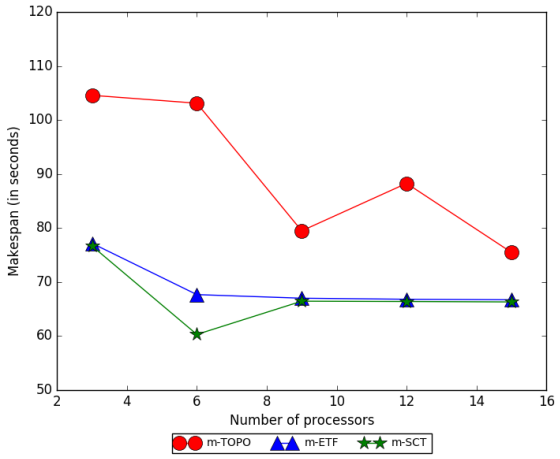

Figure 7.14: Makespan for Inception-V3 - 16GB RAM/machine - 1E9 bytes/second

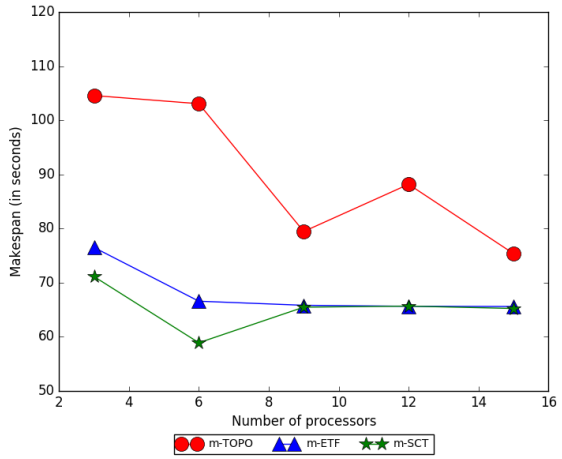

Figure 7.15: Makespan for Inception-V3 - 16GB RAM/machine - 1E10 bytes/second

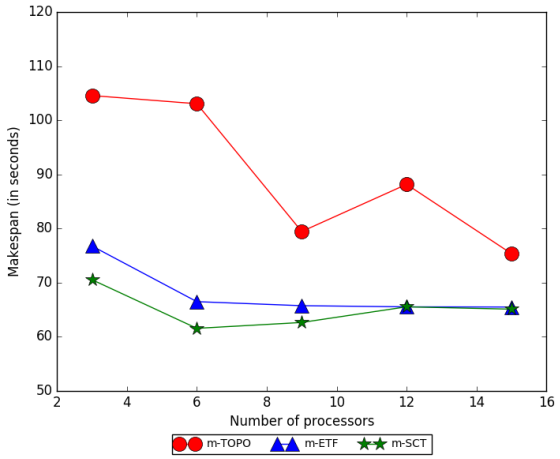

Figure 7.16: Makespan for Inception-V3 - 16GB RAM/machine - 1E11 bytes/second

Variable total memory on Inception-V3

7.3 OVERALL TREND

m-ETF takes slightly lesser time to execute as compared to m-SCT and can potentially be preferred when memory constraint is not very tight. m-SCT performs consistently well in all

cases (as long as the SCT assumption is not violated, which can happen at low bandwidths) and is significantly better performing than m-TOPO and m-ETF when memory is tightly constrained.

CHAPTER 8: REFLECTION

In this chapter we describe our unsuccessful attempts at approximating the solution to the optimal ILP formulation, modeling the memory constrained finite machine *scheduling with communication delay* problem, using a pure LP relaxation approach. We present the challenges and the reasons for failure of this method.

8.1 APPROXIMATION OF THE ILP SOLUTION

As illustrated in section 4.4, the optimal solution integer linear programming (ILP) optimally models the objective function and constraints for the memory constrained *scheduling with communication delay* problem. However, solving this optimal ILP is an NP hard problem and requires exponential time. We attempt to use the linear programming (LP) relaxation technique on the optimal ILP formulation (described in section 4.4) in order to obtain an approximate polynomial time solution to our problem. To formulate an optimal solution, we modify the infinite machine ILP [17] by incorporating finite machine and memory constraints. When we attempt to relax the additional constraints (to an LP) individually, we are faced with a unique set of challenges for each case. We describe them below in further detail.

8.1.1 Challenges in the finite machine constraints relaxation

In this section, we show that LP relaxation of ILP constraints (4.11) and (4.12), which model the finite machine assumptions, is not meaningful.

As discussed in section 4.4.2, under the finite machine assumption, it is possible for the ILP to schedule two *unrelated* tasks on the same machine. Let tasks i and j be two *unrelated* tasks. If i and j are scheduled on the same machine, then either i is executed before j, or j is executed before i. This results in an OR condition, as seen in equation (4.8). In the remainder of this section we reason that OR constraints are non convex and show that this non convexity renders the relaxation of the finite machine constraints meaningless.

When a solution set S is convex, if $a, b \in S$, then any of their convex combination $c = \lambda a + (1 - \lambda)b$ for some non-negative λ, must belong to the set S. For an OR condition in the form $x \leq L$ OR $x \geq H (L < H)$, $x = L$ and $x = H$ are feasible solutions. However, most convex combinations of L and H (e.g. $x = \frac{L+H}{2}$) are not solutions. This proves that the OR constraint is non convex.

One of the properties of LP is that its feasible region is always convex. Here, we are forced to relax the non convex solution space to a convex solution space, which is problematic. In the *OR* condition example above, the LP solution space would contain all x such that $L < x < H$, which the *OR* constraint should have excluded. In our case, equations (8.1) and (8.2) model the *OR* constraint as specified in section 4.4.2. To recap,

$$s_i - s_j \leq -p_i + (U + p_i)b_{ij} \tag{8.1}$$

$$s_i - s_j \geq L + (p_j - L)b_{ij} \tag{8.2}$$

In the ILP formulation, b_{ij} is a boolean variable that takes the value of either 0 or 1. After LP relaxation, b_{ij} is allowed to take any real value between 0 and 1. This implies that $s_i - s_j$ is able to take any real value between $-p_i$ and p_j, which further implies execution overlap between task i and j, exactly what we hope to avoid. For instance, when $b_{ij} = 0.5$, $s_i - s_j \geq \frac{L+p_j}{2}$ *OR* $s_i - s_j \leq \frac{U-p_i}{2}$. Since L is a very small negative number and U is a very large positive number, by equation (4.9) and (4.10), $s_i - s_j \geq$ a small negative number *OR* $s_i - s_j \leq$ a large positive number. In this case $s_i - s_j = 0$ is a feasible solution, indicating that tasks i and j will start at the same time on the same machine (an unacceptable result). To conclude, LP relaxation does not accurately approximate the ILP finite machine constraints.

8.1.2 Challenges in the memory constraints relaxation

In this section we discuss the challenges associated with meaningfully rounding the LP relaxation of the ILP memory constraints, due to the presence of many interdependent variables. From section 8.1.1, we know that for the finite machine case, a pure LP relaxation approach is infeasible. The LP must be supplemented by greedy scheduling in this case. We conclude that it is preferable to enforce the memory constraints in the greedy portion of the combined algorithm.

The linear equations based on the memory constraints introduce many highly correlated variables, which makes independent rounding hard. For example, in the final optimal ILP formulation (4.4.3), y_{ijp} is dependent on y_{ip} and y_{jp}. Furthermore x_{ij} is dependent on y_{ijp}. If we independently round y_{ip}, then we cannot also independently round y_{ijp} and x_{ij} without violating some of the LP constraints. Moreover, independent rounding must satisfy the memory constraint per machine (M) presented in equation (8.3) and recapped below,

$$\forall p \in P, \sum_{i=1}^{n} y_{ip} m_i \leq M \tag{8.3}$$

In the LP relaxation of (8.3), y_{ip} is allowed to take any value between 0 and 1. It is highly likely that y_{ip} will take some fractional value. After we obtain a solution to the LP relaxation problem, we must round the value of y_{ip} to 0 or 1 to satisfy the original ILP constraints. In the rounding process, if multiple fractional y_{ip}s are rounded up to 1, it is possible that equation (8.3) is no longer satisfied and the total memory required of machine p exceeds M. It is possible that with a complicated rounding scheme we could circumvent both of these issues, however, to the author's knowledge, no existing rounding scheme handles complicated correlation between variables and still results in a good approximation ratio. Alternatively, if we use LP relaxation + greedy scheduling approach, we can shift the burden of enforcing the finite machine as well as the memory constraints to the greedy portion of the algorithm and relax the infinite machine ILP to LP without any issues. This solution, combined with the small communication time (SCT) assumption, results in the m-SCT algorithm as described in section 4.3.

8.2 FUTURE DIRECTIONS

Our main roadblock is that no convex optimization approach can relax a non convex constraint. Advancements in the field of non convex optimization may help generate better algorithms for our placement problem. It may also be possible that the existing constraints be expressed or relaxed in novel ways to avoid the non convexity.

CHAPTER 9: CONCLUSION

In this thesis we have motivated the placement problem and proposed a theoretical optimal and an approximation solution to the placement problem. Our final contributions are :

- We modeled the optimal solution by modifying an existing infinite machine solution and introducing memory and finite machine constraints as detailed in section 4.4.1 and 4.4.2

- We proposed m-SCT, a memory constrained approximate solution to the placement problem, under the small communication time assumption. We proved that m-SCT approximates the optimal solution within $(1 + \frac{2+2\rho}{(2+\rho)m}) \cdot \frac{1}{K-1}$ of the finite SCT's approximation ratio, where K is the ratio of the total memory available from all machines to the total memory required by the model and ρ is ratio between the maximum communication time between any two nodes and the minimum computation time for any node in the graph.

- We compared and contrasted the performance of m-SCT against memory constrained versions of state of the art scheduling algorithms (m-ETF and m-TOPO, as detailed in chapter 7) using simulation.
  We showed that m-SCT consistently outperformed m-TOPO as long as the small communication time assumption was not violated.
  In experiments where the individual amount of memory on a machine was constrained but not the total memory, we showed that m-SCT performed well consistently without getting significantly affected by an increase in machines. m-ETF, however, performed poorly when the number of machines were fewer and significantly improved with the increase in the total number of machines. For both our models, Inception-V3 and the small CNN graph, the performance of m-SCT and m-ETF almost converged for the 15 machines case.

- We concluded that m-ETF can potentially be preferred to m-SCT when memory constraint is not very tight, since it takes slightly lesser time to execute as compared to m-SCT. m-SCT performs consistently well in all cases (as long as the SCT assumption is not violated, which can happen at low bandwidths) and is significantly better performing than m-TOPO and m-ETF when memory is tightly constrained (see chapter 7 for details).

We believe that our preliminary work will open up exploration in theoretical solutions for the placement problem rather than rely on brute force Machine Learning (ML) solutions,

as is the norm currently. We have argued that a theoretical approach is more conducive to the development of a one-click placement solution that is fast, efficient and implementable. Finally, we have detailed concrete future steps for the second part of this project.

CHAPTER 10: FUTURE WORK

1. The immediate next step is to implement the theoretical algorithms presented in this thesis in TensorFlow. Experimental TF results would go a long way in qualifying our theoretical guarantees and simulation results. To this end, some implementation challenges that may arise are :

    - Arbitrary collocation constraints enforced by TensorFlow (TF) must either be followed or modified.
    - The TF scheduler behaves transparently and does not allow the users to explicitly decide the order of scheduled tasks. Changes to TF source code may be necessary for the required granularity of control.
    - The algorithms described in chapter 3 require that the communication cost between any two nodes in the *dataflow* graph be known. This cost will depend on the combination of TFs control flow and the properties of the underlying network. The control flow may exhibit some differences from the one used in our simulation as TF is capable of manipulating it at runtime. Overall, it may prove challenging to obtain the actual communication costs.
    - The presence of dropout layers and dynamic tensor sizes in the users graph may lead to another extraction challenge as TFs present implementation only makes this information available at runtime.

2. The final implementation goal is to develop a one-click solution that can transparently determine the appropriate partition and carry out model parallelism without explicit user involvement.

3. Throughout this thesis, we assume that the memory is distributed homogeneously among machines, in any distributed cluster. In the future, we may assume the presence of heterogeneous clusters and examine how they affect the performance of our selected algorithms.

4. m-SCT may be compared against additional baselines like vertical and random cut.

5. The m-SCT algorithm performs best when the ratio of communication time to computation time is low. In situations where the user graph consists of multiple nodes that are not computationally intensive and the underlying network is slow, the small communication time (SCT) assumption may be significantly violated. collocating small

nodes [38] and treating them as a single large node for the purposes of placement may prove to be an effective strategy that should be explored in these cases.

6. This thesis opens up a variety of new directions for modeling the placement problem. A well known scheduling model that can potentially lend itself well to this work is the classical resource-constrained project scheduling problem (RCPSP) as described in [44]. It is possible that exploring novel modelling directions and constraint formulations may help in alleviating some of the issues described in chapter 7.

REFERENCES

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *ArXiv E-prints*, Dec. 2015.

[2] T. Kraska, A. Talwalkar, and J. Duchi, "Mlbase: A distributed machine-learning system," in *Conference on Innovative Data Systems Research*, 2013.

[3] E. R. Sparks, A. Talwalkar, V. Smith, J. Kottalam, X. Pan, J. Gonzalez, M. J. Franklin, M. I. Jordan, and T. Kraska, "Mli: An api for distributed machine learning," in *2013 IEEE 13th International Conference on Data Mining*, Dec 2013, pp. 1187–1192.

[4] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *Computing Research Repository*, vol. abs/1512.01274, December 2015. [Online]. Available: http://arxiv.org/abs/1512.01274

[5] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu, "Petuum: A new platform for distributed machine learning on big data," *IEEE Transactions on Big Data*, vol. 1, no. 2, pp. 49–67, June 2015.

[6] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A matlab-like environment for machine learning," in *BigLearn, Conference on Neural Information Processing Systems Workshop*, no. EPFL-CONF-192376, 2011.

[7] F. Bastien, P. Lamblin, R. Pascanu, J. Bergstra, I. Goodfellow, A. Bergeron, N. Bouchard, D. Warde-Farley, and Y. Bengio, "Theano: new features and speed improvements," *ArXiv Preprint arXiv:1211.5590*, November 2012.

[8] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[9] H. Li, A. Kadav, E. Kruus, and C. Ungureanu, "Malt: Distributed data-parallelism for existing ml applications," in *The Tenth European Conference on Computer Systems*, ser. EuroSys '15. New York, NY, USA: ACM, 2015, ISBN: 978-1-4503-3238-5. [Online]. Available: http://doi.acm.org/10.1145/2741948.2741965 pp. 3:1–3:16.

[10] L. Bergstrom and J. Reppy, "Nested data-parallelism on the gpu," in *The 17th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '12. New York, NY, USA: ACM, 2012, ISBN: 978-1-4503-1054-3. [Online]. Available: http://doi.acm.org/10.1145/2364527.2364563 pp. 247–258.

[11] A. Sujeeth, H. Lee, K. Brown, T. Rompf, H. Chafi, M. Wu, A. Atreya, M. Odersky, and K. Olukotun, "Optiml: an implicitly parallel domain-specific language for machine learning," in *The 28th International Conference on Machine Learning (ICML-11)*, 2011, pp. 609–616.

[12] S. HANASSAB and H. A. FATMI, "Parallel processors for cybernetic systems," *Cybernetics and Systems*, vol. 11, no. 1-2, pp. 179–192, October 1980. [Online]. Available: https://doi.org/10.1080/01969728008960234

[13] F. Seide and A. Agarwal, "Cntk: Microsoft's open-source deep-learning toolkit," in *The 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2016, pp. 2135–2135.

[14] S. Tokui, K. Oono, S. Hido, and J. Clayton, "Chainer: a next-generation open source framework for deep learning," in *Workshop on Machine Learning Systems (LearningSys) in The Twenty-ninth Annual Conference on Neural Information Processing Systems (NIPS)*, vol. 5, 2015.

[15] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard et al., "Tensorflow: A system for large-scale machine learning." in *USENIX Symposium on Operating Systems Design and Implementation*, vol. 16, 2016, pp. 265–283.

[16] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean, "Device placement optimization with reinforcement learning," *ArXiv Preprint arXiv:1706.04972*, June 2017.

[17] C. Hanen and A. Munier, "An approximation algorithm for scheduling dependent tasks on m processors with small communication delays," in *Emerging Technologies and Factory Automation, 1995. ETFA '95, Proceedings., 1995 INRIA/IEEE Symposium on*, vol. 1, Oct 1995, pp. 167–189 vol.1.

[18] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *Computer Vision and Pattern Recognition, 2009. CVPR 2009.* IEEE, 2009, pp. 248–255.

[19] M. J. Menne, I. Durre, R. S. Vose, B. E. Gleason, and T. G. Houston, "An overview of the global historical climatology network-daily database," *Journal of Atmospheric and Oceanic Technology*, vol. 29, no. 7, pp. 897–910, July 2012.

[20] K. Jung and A. Brown, *Beginning Lua Programming*. Birmingham, UK, UK: Wrox Press Ltd., 2007, ISBN: 0470069171.

[21] F. A. Gers, J. A. Schmidhuber, and F. A. Cummins, "Learning to forget: Continual prediction with lstm," *Neural Computation*, vol. 12, no. 10, pp. 2451–2471, October 2000. [Online]. Available: http://dx.doi.org/10.1162/089976600300015015

[22] G. Bebis and M. Georgiopoulos, "Feed-forward neural networks," *IEEE Potentials*, vol. 13, no. 4, pp. 27–31, October 1994.

[23] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *The 22nd ACM international conference on Multimedia*. ACM, 2014, pp. 675–678.

[24] J. M. Landsberg, "Tensors: geometry and applications," *Representation theory*, vol. 381, p. 402, January 2012, ISBN: 978-0-8218-6907-9.

[25] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *ArXiv Preprint arXiv:1502.03167*, February 2015.

[26] D. C. Cireşan, U. Meier, J. Masci, L. M. Gambardella, and J. Schmidhuber, "Flexible, high performance convolutional neural networks for image classification," in *The Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume Two*, ser. IJCAI'11. AAAI Press, 2011, ISBN: 978-1-57735-514-4. [Online]. Available: http://dx.doi.org/10.5591/978-1-57735-516-8/IJCAI11-210 pp. 1237–1242.

[27] T. Mikolov, S. Kombrink, A. Deoras, L. Burget, and J. Cernocky, "Rnnlm-recurrent neural network language modeling toolkit," in *The 2011 Automatic Speech Recognition and Understanding (ASRU) Workshop*, 2011, pp. 196–201.

[28] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *ArXiv Preprint arXiv:1409.0473*, September 2014.

[29] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica et al., "Above the clouds: A berkeley view of cloud computing," Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Tech. Rep., 2009.

[30] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1990, ISBN: 0716710455.

[31] H. Saran and V. V. Vazirani, "Finding k cuts within twice the optimal," *SIAM Journal on Computing*, vol. 24, no. 1, pp. 101–108, February 1995. [Online]. Available: http://dx.doi.org/10.1137/S0097539792251730

[32] W. Fernandez de la Vega, M. Karpinski, and C. Kenyon, "Approximation schemes for metric bisection and partitioning," in *The Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '04. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2004, ISBN: 0-89871-558-X. [Online]. Available: http://dl.acm.org/citation.cfm?id=982792.982864 pp. 506–515.

[33] C. H. Ding, X. He, H. Zha, M. Gu, and H. D. Simon, "A min-max cut algorithm for graph partitioning and data clustering," in *Data Mining, 2001. ICDM 2001, IEEE International Conference on*. IEEE, 2001, pp. 107–114.

[34] S. Arora, S. Rao, and U. Vazirani, "Expander flows, geometric embeddings and graph partitioning," in *The Thirty-sixth Annual ACM Symposium on Theory of Computing*, ser. STOC '04. New York, NY, USA: ACM, 2004. [Online]. Available: http://doi.acm.org/10.1145/1007352.1007355 pp. 222–231.

[35] R. Krauthgamer, J. S. Naor, and R. Schwartz, "Partitioning graphs into balanced components," in *The Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '09. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1496770.1496872 pp. 942–949.

[36] J. Hoogeveen, J. Lenstra, and B. Veltman, "Three, four, five, six, or the complexity of scheduling with communication delays," *Operations Research Letters*, vol. 16, no. 3, pp. 129 – 137, October 1994. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0167637794900248

[37] J.-J. Hwang, Y.-C. Chow, F. D. Anger, and C.-Y. Lee, "Scheduling precedence graphs in systems with interprocessor communication times," *SIAM Jornal on Computing*, vol. 18, no. 2, pp. 244–257, April 1989. [Online]. Available: http://dx.doi.org/10.1137/0218016

[38] C. H. Papadimitriou and M. Yannakakis, "Towards an architecture-independent analysis of parallel algorithms," *SIAM Journal on Computing*, vol. 19, no. 2, pp. 322–328, May 1990.

[39] A. B. Kahn, "Topological sorting of large networks," *Communications of the ACM*, vol. 5, no. 11, pp. 558–562, November 1962. [Online]. Available: http://doi.acm.org/10.1145/368996.369025

[40] N. Karmarkar, "A new polynomial-time algorithm for linear programming," in *The Sixteenth Annual ACM Symposium on Theory of Computing*. ACM, 1984, pp. 302–311.

[41] R. Kolisch, "Serial and parallel resource-constrained project scheduling methods revisited: Theory and computation," *European Journal of Operational Research*, vol. 90, no. 2, pp. 320–333, April 1996.

[42] R. Niemann and P. Marwedel, "An algorithm for hardware/software partitioning using mixed integer linear programming," *Design Automation for Embedded Systems*, vol. 2, no. 2, pp. 165–193, March 1997.

[43] T. Davidovic, L. Liberti, N. Maculan, and N. Mladenovic, "Towards the optimal solution of the multiprocessor scheduling problem with communication delays," *Multidisciplinary International Scheduling Conference: Theory and Applications*, January 2007.

[44] F. B. Talbot, "Resource-constrained project scheduling with time-resource tradeoffs: The nonpreemptive case," *Management Science*, vol. 28, no. 10, pp. 1197–1210, October 1982.

[45] A. Hagberg, P. Swart, and D. S Chult, "Exploring network structure, dynamics, and function using networkx," Los Alamos National Lab.(LANL), Los Alamos, NM (United States), Tech. Rep., 2008.

[46] C. Szegedy, S. Ioffe, and V. Vanhoucke, "Inception-v4, inception-resnet and the impact of residual connections on learning," *Computing Research Repository*, vol. abs/1602.07261, February 2016. [Online]. Available: http://arxiv.org/abs/1602.07261

[47] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[48] R. H. Bartels and G. H. Golub, "The simplex method of linear programming using lu decomposition," *Communications of the ACM*, vol. 12, no. 5, pp. 266–268, May 1969.

[49] J. A. Tomlin, "Progress in mathematical programming interior-point and related methods," N. Megiddo, Ed. New York, NY, USA: Springer-Verlag New York, Inc., 1988, ch. A Note on Comparing Simplex and Interior Methods for Linear Programming, pp. 91–103, ISBN: 0-387-96847-4. [Online]. Available: http://dl.acm.org/citation.cfm?id=72638.72644

[50] E. D. Andersen and K. D. Andersen, "The mosek interior point optimizer for linear programming: an implementation of the homogeneous algorithm," in *High Performance Optimization*. Springer, 2000, pp. 197–232.