Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

# Graph Partitioning and Scheduling for Distributed Dataflow Computation

Larissa Laich

**Course of Study:** Softwaretechnik

**Examiner:** Prof. Dr. Kurt Rothermel

**Supervisor:** Dr. Muhammad Adnan Tariq, Dipl.-Inf. Christian Mayer

**Commenced:** September 14, 2016

**Completed:** March 16, 2017

**CR-Classification:** G.2.2, C.2.4

# Abstract

During the last years, the amount of data which can be represented and processed as graph structured data has massively increased. To process these large data sets, graph processing systems have been developed which distribute and partition a graph among multiple machines. Due to an increase in processing power and data collection, machine learning and especially neural networks have become very popular. Consequently, machine learning systems like TensorFlow have emerged. Machine learning models can be represented as dataflow graphs and often take days to train as the dataflow graph is executed thousands of times. Graph partitioning determines how the graph is divided and which node is placed on which device. Scheduling decides which node should be computed next during execution. Smart partitioning and scheduling can drastically reduce the total execution time. Most existing solutions do not consider serveral important constraints like memory limitations, device or colocation constraints which can be directly derived from the machine learning library TensorFlow. This thesis presents and evaluates different partitioning and scheduling strategies meeting the constraints required for a realistic environment. One of these developed partitioning algorithms is based on a heuristic function considering execution time, memory and traffic and tries to map time-critical nodes on fast devices. This partitioning algorithm performed very well in combination with a scheduling strategy which schedules the executable node first whose upwards path takes longest to compute. On the evaluated graphs extracted from TensorFlow, this strategy was up to 75 % and at least 45% better in terms of graph execution time than the slightly adapted popular HEFT algorithm which is a common benchmark. In combination with the aforementioned scheduling strategy a partitioning which aims to assign the critical path nodes to the fastest device showed equally promising results.

# Contents


1 Introduction 11
1.1 Contributions . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
1.2 Outline . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13

2 System Model and Problem formulation 15
2.1 TensorFlow™ . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
2.2 Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
2.3 Problem Formulation . . . . . . . . . . . . . . . . . . . . . . . . 17
2.4 Insertion of Send and Receive Nodes . . . . . . . . . . . . . . . 18
2.5 Execution Time Calculation . . . . . . . . . . . . . . . . . . . . 19
2.6 Simplifications . . . . . . . . . . . . . . . . . . . . . . . . . . . 21

3 Approach Overview 25
3.1 Evaluation Framework . . . . . . . . . . . . . . . . . . . . . . . 25
3.2 Timing Behaviour . . . . . . . . . . . . . . . . . . . . . . . . . . 29

4 Partitioning 31
4.1 Problem Formulation . . . . . . . . . . . . . . . . . . . . . . . . 31
4.2 Hashing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
4.3 HEFT . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
4.4 Batch Split . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
4.5 Iterated Critical Path . . . . . . . . . . . . . . . . . . . . . . . . 34
4.6 Critical Path . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
4.7 MITE . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
4.8 Depth First Search . . . . . . . . . . . . . . . . . . . . . . . . . 39

5 Scheduling 41
5.1 Problem Formulation . . . . . . . . . . . . . . . . . . . . . . . . 41
5.2 FIFO Scheduler . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
5.3 PCT Scheduler . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
5.4 MSR Scheduler . . . . . . . . . . . . . . . . . . . . . . . . . . . 42

6 Sample Data 45
6.1 Typical Examples in TensorFlow . . . . . . . . . . . . . . . . . . 45

6.2 Data Generation and Sample Data . . . . . . . . . . . . . . . . . . . . . . 46

7 Evaluation 49
7.1 Synthetic Graph Evaluations . . . . . . . . . . . . . . . . . . . . . . . . . 49
7.2 TensorFlow Graph Evaluations . . . . . . . . . . . . . . . . . . . . . . . 52
7.3 Evaluation on Few Devices . . . . . . . . . . . . . . . . . . . . . . . . . . 55

8 Related work 61

9 Discussion 63

10 Conclusion 65

List of Abbreviations 67

Bibliography 67

# List of Figures

1.1 Different scheduling algorithms influence the total execution time on the same partitioning. . . . 12

2.1 The dataflow graph is divided onto three machines . . . . 17
2.2 Post-processing to insert send and receive nodes after partitioning . . . . 19
2.3 The graph is divided onto three machines. Each node has a node id (upper value) and number of operations (lower value). The labels on the edges represent the tensor size. . . . . 20
2.4 Schedule of the sample graph shown in Figure 2.3 . . . . 21

3.1 Evaluation framework pipeline and its components . . . . 26
3.2 Sample graph labelled with node id, #ops and edge weigths . . . . 30

5.1 Scheduling strategies . . . . 43

6.1 Size and number of colocation groups . . . . 47

7.9 Device utilization for the scheduling and partitioning strategies . . . . 59
7.10 Number of executable nodes per device before making the decision which is node is executed next . . . . 60

# List of Tables

2.1 Scheduling order per device . . . . . . . . . . . . . . . . . . . . . . . . . 20
2.2 Notation overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22

7.1 Properties of six synthetic graphs . . . . . . . . . . . . . . . . . . . . . . 49
7.2 Sample graphs extracted from TensorFlow . . . . . . . . . . . . . . . . . . 52

# 1 Introduction

During the last decade the collection of graph structured data as well as the processing power massively increased. Due to these new possibilities distributed graph processing systems like Pregel [MAB+10], PowerGraph [GLG+12], GrapH [MTLR16] or GraphCEP [MMTR16] for large-scale graph computation emerged. A powerful application is the execution of machine learning algorithms which are for example used for natural language processing [CW08] or image classification [KSH12].

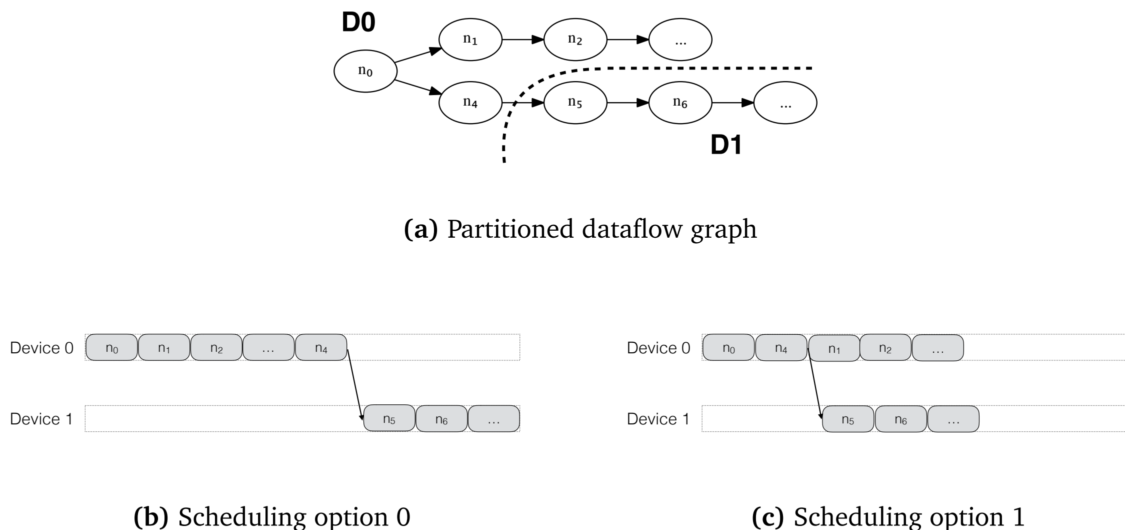
Machine learning models often consist of billions of parameters [DCM+12] and can be represented as dataflow graphs. In order to shorten the training and execution time the computation is distributed and therefore the graph is partitioned onto different machines. Moreover, the graph could be too large to fit on a single machine which is another reason for partitioning. Other possibilities to reduce the execution time are model replication [NSB+15], pipelining [AC86] or computing a batch of input data at once [LZCS14] and merge the results. This thesis focuses on the partitioning approach.

A graph can be partitioned by dividing its nodes into disjoint subsets, i.e. cut the edges [CSCC15]. The classic k-way partitioning problem is to divide a graph's node set in k equal sized subsets in a way that the number of cut edges is minimized. This problem is NP-complete [KK98]. When executing the graph, the node execution order is determined by the scheduling for each device.

Clearly, minimizing the execution time as well as the network traffic is an important challenge. The dataflow graph is executed multiple times and therefore runtime information can be collected and later used for an improved partitioning. Furthermore, it takes much time to execute the model in a similar manner over and over again [CZG+16]. Therefore, one might consider to spend more time on the partitioning and scheduling to reduce the overall execution time.

In order to meet all the from TensorFlow derived criteria for the partitioning of machine learning models the standard graph partitioning problem has to be extended. In particular, constraints like memory, device constraints or heterogenous devices have to be considered when searching for a realistic graph partitioning solution.

However, the partitioning problem has to be investigated in combination with the scheduling problem as a bad scheduling can increase the execution time drastically.




**(a)** Partitioned dataflow graph

**(b)** Scheduling option 0

**(c)** Scheduling option 1

**Figure 1.1:** Different scheduling algorithms influence the total execution time on the same partitioning.

Consider the example of Figure 1.1 (a) depicting a dataflow graph which is partitioned onto device 0 and device 1. In Figure 1.1 (b) the scheduling option 0 is visualized in which $n_4$ is executed only when $n_1$ and all its successors are executed. This becomes even worse the longer the execution of the direct and indirect successor nodes of $n_1$ takes. Scheduling option 1 which is shown in Figure 1.1 (c) would be a better choice as it enables parallel execution by executing $n_4$ before $n_1$ and thus shortens the total execution time.

In this thesis, we developed multiple partitioning and scheduling algorithms considering constraints which are derived from the machine learning library TensorFlow. We compare and evaluate these algorithms which aim to reduce the overall execution time and traffic.

## 1.1 Contributions

In particular, the thesis provides the following contributions:

1. We defined a system model which contains the constraints derived from the machine learning library TensorFlow. Moreover, the scheduling and partitioning problem was formulated considering the additional constraints.

2. To solve the modified partitioning problem, multiple partitioning strategies were developed. Moreover, three scheduling strategies were implemented.
3. In order to evaluate the aforementioned combinations of scheduling and partitioning algorithms an evaluation framework has been developed. The framework is easily extendable and allows a quick integration and implementation of new strategies.
4. We compared partitioning and scheduling algorithms with respect to their execution time and traffic. The results have been visualized.

## 1.2 Outline

Beyond the introduction chapter this thesis is structured as follows. In Chapter 2 TensorFlow is introduced, the system model is defined and the problem is formulated. Chapter 3 presents an approach to solve the problem and describes multiple components in the developed evaluation pipeline. The scheduling and partitioning problem is described more detailed in Chapter 4 and 5. In these chapters multiple strategies are presented and discussed. In Chapter 6 the generation of sample data is presented. These data sets are used to compare and evaluate the partitioning and scheduling strategies in Chapter 7. Related work is presented in Chapter 8 while future work and drawbacks of the existing solutions are discussed in Chapter 9. The whole thesis is concluded in Chapter 10.

# 2 System Model and Problem formulation

In the following sections the system model is defined and the problem is formulated. Therefore, the machine learning library TensorFlow and its properties are described in order to derive model properties. Beyond the model, the metrics execution time and traffic are presented which are used to evaluate partitioning and scheduling algorithms in chapter 7. An example illustrates how the execution time is calculated theoretically. For clarity, we reduced the model complexity of TensorFlow as shown in Section 2.6.

## 2.1 TensorFlow™

The system model and the required constraints are derived from TensorFlow [MAP+15] [ABC+16] [Sca16]. TensorFlow is a machine learning library which was developed by Google and published as open source in November 2015. It has become one of the most popular machine learning libraries since then. TensorFlow is widely used for deep learning models (multi-layer neural networks) and has a C++, python and Java API. In former times many researchers used a machine learning flexible library for research and an efficient one for production. TensorFlow wants to combine both in order to deploy models easily which have been developed during research.

To train a machine learning model in TensorFlow, a computational graph has to be defined and executed. In this case, a computational graph is a stateful dataflow graph. There are implicit data dependencies: A node can not be executed until all predecessor nodes are executed and their computed tensor (multidimensional array) is transferred.

Computations make side-effects to variables (model parameters) which are used in future runs and the results of the machine learning. During training, a forward pass [CZG+16] with the current values of the model parameters is calculated. In supervised learning, the result is compared with the true label and an error term is computed. Afterwards, the results are backpropagated and the weights of the model parameters are adjusted.

### 2.1.1 Parallelism

Parallelism is an important concept in order to reduce the execution time. In TensorFlow, model as well as data parallelism is possible.

For data parallelism, the model is replicated [NSB+15]. The input data is split and different subsets of the data are trained on the replications of the model. The same global model is updated e.g by taking the parameter average of the model replications. These updates can be performed synchronously or asynchronously. In synchronous data parallelism, the data is divided into batches and computed on x model replicas. Afterwards, the gradient is computed and the model is updated synchronously. Alternatively, each replica updates asynchronously. Challenges for data parallelism are for example to tolerate different processing speeds of the model replicas.

For model parallelism, the model is split on different devices [NSB+15] and can be computed in parallel due to the placement on multiple devices. Two nodes can potentially be executed in parallel if they are placed on different devices and there is no data dependency between them.

## 2.2 Model

This section defines a system model in which the requirements derived from TensorFlow are considered. Figure 2.1 shows the acyclic dataflow graph which has to be divided on three machines. Each node represents a computation and has a unique id as well as computation or operation costs #ops which represent the necessary number of operations to compute the node's result. A node can not be executed until all direct predecessor vertices are computed. Furthermore, a vertex has a device constraint (CPU, GPU, TPU or ALL) indicating on which device type the vertex can be executed. TPU stands for Tensor Processing Unit [ABC+16] and is a custom chip developed by Google for machine learning applications. The device condition might reduce the number of feasible devices on which a vertex can be assigned. To store a vertex a defined amount of memory is necessary which is provided in advance.

Each edge represents a tensor being transferred from one vertex to another. An edge has a weight indicating the size of the transferred tensor. If a vertex has multiple outgoing edges all outgoing edges have the same weight and the result is available after the vertex is computed.

In addition to the set of vertices and edges a graph is defined by a set of colocations C. Each colocation constraint contains two nodes which have to be assigned to the same machine. The colocations are represented as red dots in Figure 2.1.

The graph G has to be divided into n subgraphs and mapped on a set of devices D = $\{d_1, d_2, ..., d_n\}$. A device has a device type (CPU, GPU or TPU), memory size and speed which is defined as number of operations per execution unit. Each device can communicate with every other device which is important to be able to transfer tensors. The communication speed between device x and y is defined by a transfer rate $t_{x-y}$. In our model only one node can be executed by a device at once. Once a node is executed it can not be interrupted as non-preemptive scheduling is assumed.

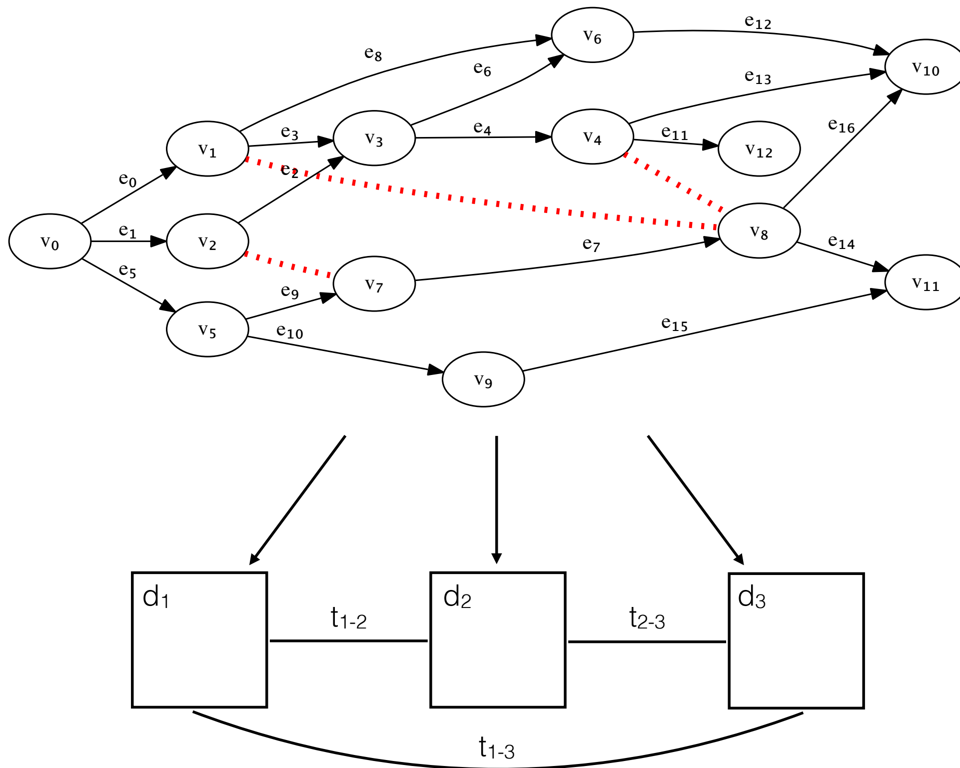



**Figure 2.1:** The dataflow graph is divided onto three machines

## 2.3 Problem Formulation

In order to fulfil the requirements of the TensorFlow library, the standard partitioning problem has to be adapted to consider multiple constraints.

A graph is defined by G = (V, E, C) in which V = $\{v_0, v_1, ..., v_r\}$ is a set of vertices, $E \subset (V \times V \times \mathbb{N})$ a set of weighted edges and $C \subset (V \times V)$ a set of colocations. Each

tuple $(v_x, v_y) \in$ C defines two vertices which have to be assigned to the same machine. Bigger colocation groups are formed transitively. In our model the graph is a directed acyclic graph.

The goal of graph partitioning is to split the set of vertices onto n machines. An edge cut divides the graph by assigning vertices to different machines and thereby cutting edges. Scheduling defines the vertex execution order. Both problems are further described in Chapter 4 and 5.

The overall goal is to minimize two factors:

The first and more important one is the execution time or also called makespan. It is defined as the first point of time on which every device has finished computation. This relation is expressed in Equation 2.1. All abbreviations and function names are listed in Table 2.2.

$$makespan = \max_{d \in D} exec(d) \tag{2.1}$$

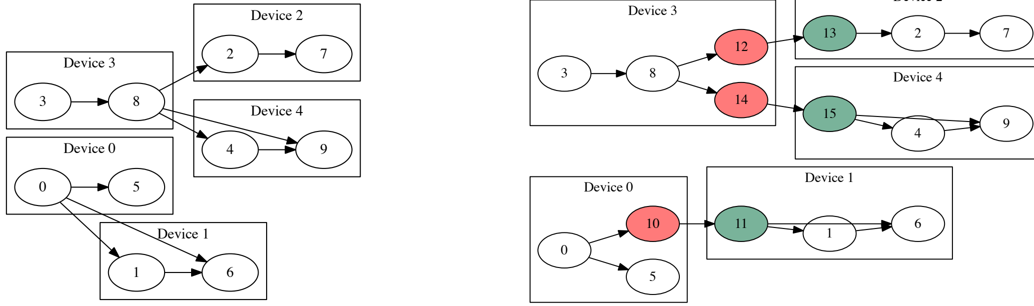
The other metric is the traffic which is defined as the sum of all cross-device edges' tensor sizes. This is formulated in Equation 2.2.

$$traffic = \sum_{e \in CE} size(e)\,, \quad CE = \{(v_x, v_y, x) \in E \wedge d(v_x) \neq d(v_y)\} \tag{2.2}$$

## 2.4 Insertion of Send and Receive Nodes

In this section the process of adding send and receive nodes is decribed. After partitioning, send and receive nodes are inserted for cross-device edges. If a vertex has multiple successor nodes which are assigned to the same machine only one send-receive-node-pair is inserted. This has the huge advantage of transmitting data only once. Furthermore, it also makes a better code abstraction level possible as send and receive nodes can be the only nodes responsible for inter-device data transfer.

Figure 2.2 (a) depicts a dataflow graph which is partitioned onto five devices. After partitioning, the send and receive nodes are inserted. The resulting graph is shown in Figure 2.2 (b). The new send nodes are coloured red and the new receive nodes are green. Send node 14 and receive node 15 transfer the output data from node 8 only once between device 3 and 4. If the output tensor size of node 8 is 50 bytes then this amount of data is also transferred on the edges (8,14), (14,15), (15,4) and (15,9).




**(a)** Partitioned graph before adding send and receive nodes

**(b)** Partitioned graph after inserting send and receive nodes

**Figure 2.2:** Post-processing to insert send and receive nodes after partitioning

## 2.5 Execution Time Calculation

Partitioning decides how the node set is split and which nodes are placed on which device. The scheduling determines the nodes' execution order. The overall goal is to minimize the total execution time and reduce the network traffic if possible. How the execution time is determined is illustrated in this section.

Figure 2.3 shows a partitioned graph. The nodes are labelled with their id and computation costs #ops. The tensor sizes are written on the corresponding edges. The abbreviation EU stands for execution unit. All abbreviations and function names are listed and explained in Table 2.2. The device speed of device 0 is 10 $\frac{Ops}{EU}$, the one of device 1 is 30 $\frac{Ops}{EU}$ and device 2 is 40 $\frac{Ops}{EU}$ fast. The transfer rate between device 0 and 1 is $\frac{25}{EU}$ while the transfer rate between device 0 and 2 is $\frac{50}{EU}$.

At first, the execution times for each node are calculated by dividing the node's number of operations with the device speed:

$$exec(n_0) = \frac{\#ops(n_0)}{speed(d_0)} = \frac{30\,Ops}{10\frac{Ops}{EU}} = 3\,EU \qquad exec(n_8) = \frac{\#ops(n_8)}{speed(d_0)} = \frac{10\,Ops}{10\frac{Ops}{EU}} = 1\,EU$$

$$exec(n_1) = \frac{\#ops(n_1)}{speed(d_0)} = \frac{50\,Ops}{10\frac{Ops}{EU}} = 5\,EU \qquad exec(n_2) = \frac{\#ops(n_2)}{speed(d_1)} = \frac{60\,Ops}{30\frac{Ops}{EU}} = 2\,EU$$

$$exec(n_6) = \frac{\#ops(n_6)}{speed(d_0)} = \frac{10\,Ops}{10\frac{Ops}{EU}} = 1\,EU \qquad exec(n_3) = \frac{\#ops(n_3)}{speed(d_1)} = \frac{30\,Ops}{30\frac{Ops}{EU}} = 1\,EU$$

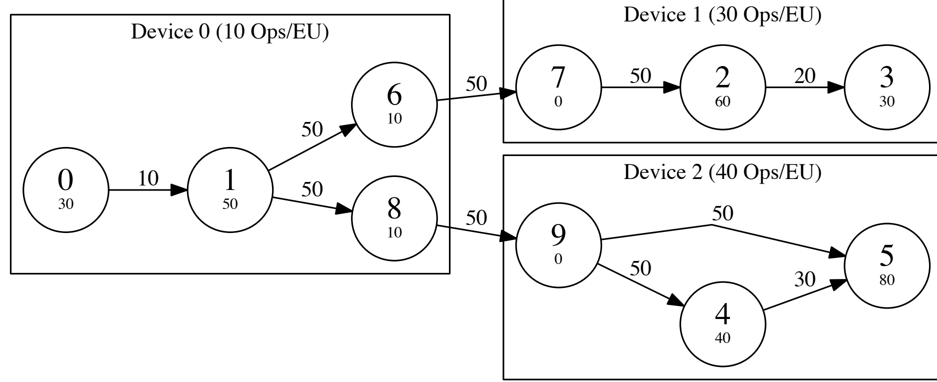



**Figure 2.3:** The graph is divided onto three machines. Each node has a node id (upper value) and number of operations (lower value). The labels on the edges represent the tensor size.

**Table 2.1:** Scheduling order per device

| Device | Order (in node ids) |
|---|---|
| 0 | 0, 1, 6, 8 |
| 1 | 7, 2, 3 |
| 2 | 9, 4, 5 |

$$exec(n_4) = \frac{\#ops(n_4)}{speed(d_2)} = \frac{40\,Ops}{40\frac{Ops}{EU}} = 1\,EU \qquad exec(n_5) = \frac{\#ops(n_5)}{speed(d_2)} = \frac{80\,Ops}{40\frac{Ops}{EU}} = 2\,EU$$

The receive nodes take zero execution units to receive the data. Moreover, the required time to transfer the computed results has to be taken into account:

$$\text{transfer time}(n_6, n_7) = \frac{\text{weight}(n_6, n_7)}{\text{transfer rate}(d_0, d_1)} = \frac{50}{\frac{25}{EU}} = 2\,EU$$

$$\text{transfer time}(n_8, n_9) = \frac{\text{weight}(n_8, n_9)}{\text{transfer rate}(d_0, d_2)} = \frac{50}{\frac{50}{EU}} = 1\,EU$$

The scheduling algorithm defines the execution order as shown in Table 2.1. The resulting execution sequence is visualized in Figure 2.4. The computed data from node

6 arrives at device 1 after 11 execution units. To finish processing device 1 needs additional 3 units. At device 2 the data arrives after 11 execution units and it needs 3 units to finish processing. So the overall execution time is 14 execution units.

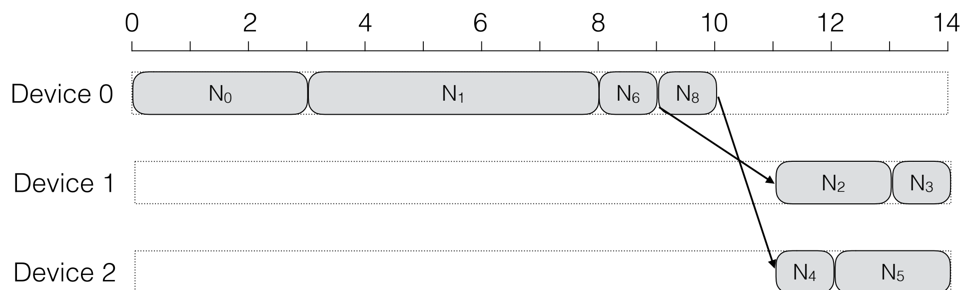



**Figure 2.4:** Schedule of the sample graph shown in Figure 2.3

## 2.6 Simplifications

In order to reduce the model and problem complexity some less important parts are not taken into account.

With respect to TensorFlow multiple properties could be additionally considered:

- Backpropagation: In order to adapt the model parameters, backpropagation is used and information flows in the reverse direction of the dataflow graph [CZG+16] [Sca16]. Backpropagation is very specific to machine learning and could be modeled by inserting additional nodes and edges to the graph [MAP+15].
- Control flow nodes [MAP+15]: TensorFlow allows the usage of control flow nodes. With control flow operators, it becomes for example possible to express iterations, if-conditions or while-loops on dataflow graphs.
- Common subexpression elimination [MAP+15]: Redundant parts of the model are eliminated and computed only once. This approach helps to avoid unnecessary computation. If this procedure eliminates nodes and edges before partitioning, our model can also take this into account.
- Model replication [MAP+15]: In order to allow data parallel training the model is replicated. We can also express model replication by replicating the whole graph. If only parts of the graphs are replicated some edges have to be inserted to connect the "source" nodes of the replicated parts with its predecessors.

**Table 2.2:** Notation overview

| | |
|---|---|
| G = (V, E, C) | Graph with vertex set V, edge set E and colocations C |
| $E \subset (V \times V \times \mathbb{N})$ | Edge set consisting of tuples (start node, end node, weight) |
| D | Device set |
| exec($n_x$) | Execution units (time) necessary to compute node $n_x$ |
| #ops($n_x$) | Number of operations necessary to compute node $n_x$ |
| speed($d_x$) | Speed of device $d_x$ (in operations per execution unit) |
| transfer time($n_x$, $n_y$) | Transfer time to transfer tensor from $n_x$ to $n_y$ |
| weight($n_x$, $n_y$) | Weight of edge ($n_x$, $n_y$) (= tensor size) |
| transfer rate($d_x$, $d_y$) | Transfer rate between device $d_x$ and $d_y$ |
| d($n_x$) | Device on which $n_x$ is placed by the partitioning algorithm. Is equal to -1 if the node is not placed yet. |
| type($d_x$) | Type of device $d_x$, e.g CPU or GPU |
| device constraint(n) | Device constraint of node n |
| memory size(d) | Memory size of device d |
| $V_{d_x}$ | Set of all nodes assigned to device $d_x$ |
| $V_{g_x}$ | Set of all nodes being member of the colocation group $g_x$ |
| exec(d) | Total execution time of device d |
| exec$_{min}$(d) | Minimal execution time possible if the device d would execute all nodes sequentially without any idle times |
| #ops($d_x$) | Sum of all nodes' operations assigned to $d_x$ |
| EU | Execution unit |
| Ops | Operations |
| size(e) | Size of the edge e = $(n_y, n_x)$. Equal to the output tensor size of node $n_y$ |
| size$_{est}$(n) | Estimated size of a node n's memory requirement. Sum of memory demand, output tensor size and input tensor sizes. |
| pred($n_x$) | Set of direct predecessor nodes of node $n_x$ |
| succ($n_x$) | Set of direct successor nodes of node $n_x$ |
| makespan | Time period from computation beginning until the last device has finished execution. |
| assignment unit | Colocation group or not colocated node (assigned as one unit during partitioning). |

- Computation pipelining [MAP+15]. Smart pipelining would increase the benefits of partitioning in most cases and has no impact in the worst case.
- Subgraph execution [ABC+16]: Only parts of the model are executed. In our model, we assume that all nodes are executed exactly once. However, subgraph execution can be handled by treating the subgraph as a graph. Nevertheless, this is only efficient if the executed subgraph does not change very often. If this is the case, more dynamic partitioning approaches would be required.

The tensors sizes also depend on the input data [Sca16]. We assume that the tensor sizes stay the same and to know them in advance. The TensorFlow developers also state that their placement strategy does estimate fixed tensor sizes based on a cost model or simulated execution [MAP+15].

Another simplification is that a node's execution duration is computed by its number of operations divided by the device speed. However, it could be different for each node device pair.

# 3 Approach Overview

This chapter provides an approach overview to solve the problem presented in Chapter 2. The first idea was to implement novel partitioning and scheduling algorithms within the TensorFlow library and compare them with already existing TensorFlow partitioning and scheduling algorithms. However, only a very simple placement algorithm is published by Google for now. Every node is mapped per default on the same device unless the user places the nodes manually. Additional factors like the cost model or memory constraints are not considered in the code published by Google, although described in the TensorFlow whitepaper [MAP+15]. The described scheduling strategy is also not available. This is the reason why we decided to develop an evaluation framework in Java.

## 3.1 Evaluation Framework

In order to be able to compare and implement partitioning and scheduling strategies faster and consider all the required constraints an evaluation framework has been developed. In the following sections the components of this evaluation pipeline are described. Figure 3.1 shows these components. At first, device and graph data is read and colocation groups are calculated. Some partitioning algorithms require certain preprocessing steps which are executed during the partitioning preprocessing. Afterwards, the graph is partitioned onto multiple devices and send and receive nodes are inserted during partitioning postprocessing. Depending on the scheduling strategy, some preprocessing steps are necessary. Subsequently, the scheduling is performed on each device. At the end of the computation, the results are collected and visualized.

Every part of the framework can be implemented independently. Furthermore, partitioning and scheduling algorithms can be integrated and exchanged easily.

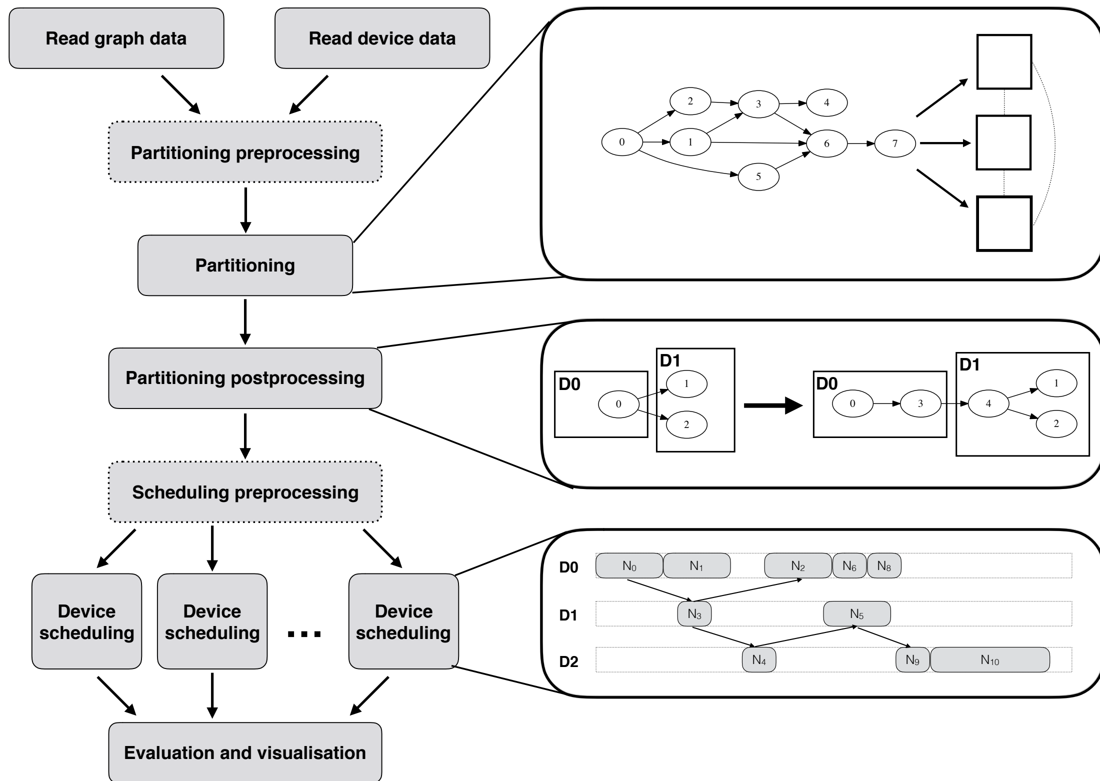



**Figure 3.1:** Evaluation framework pipeline and its components

### 3.1.1 Device and Graph Reader

In order to evaluate the strategies, dataflow graphs are extracted from TensorFlow. On top of that, sample synthetic graph and device data is created in order to test various graph and device combinations. The properties of these graphs and devices are further described in Chapter 6.

Within the evaluation framework a device reader is responsible for reading the device data from a csv-file. A custom graph reader reads the graph information from a specified file. It does not matter if the graph data is synthetic or extracted from TensorFlow: It is stored in an equal structure as csv-file.

**Algorithmus 3.1** Calculate colocation groups

```
static int componentId = 0;
procedure READ_NODE(n_x)
    ColocationComponent component;
    for all n ∈ col(n_x) do
        //colocationId -1 means that the node is not in any colocation group yet
        if n.colocationId == -1 then
            if component == null then
                //when creating a new component it is stored in a set of components
                component = new Component(componentId);
                componentId ++;
            end if
            //in addNode the colocationId of n is also set
            component.addNode(n);
        else
            if component == null then
                component = ColocationComponent.getComponentForId(n.colocationId);
            else
                //union
                component.merge(ColocationComponent.getComponentForId(n.colocationId));
            end if
        end if
    end for
end procedure
```

### 3.1.2 Partitioning Preprocessing

When reading the graph data, colocation groups are calculated. Algorithm 3.1 shows the algorithm which generates the colocation groups. Whenever a node $n_x$ is colocated all its colocated nodes $col(n_x)$ are checked. If the node $n_y \in col(n_x)$ is not in a colocation group yet it is added to the colocation group of $n_x$. If it is already in a colocation group the two colocations groups are merged. If two colocation groups are merged or a node is added to a colocation group the device constraint has to be checked. If member nodes have contradictory device constraints the input data is invalid and an exception is thrown. For a short and clear notation, we assume that $n_x$ is also a member of col($n_x$) in Algorithm 3.1.

Depending on the partitioning strategy other preprocessing steps are taken. If this is necessary for a partitioning strategy it is described more detailed in the partitioning chapter 4.

### 3.1.3 Partitioning

A partitioning algorithm splits the nodes on different devices and determines which node is placed on which device. Some devices might not be feasible for a specific node. To decide if a node can be placed onto a specific machine the following factors have to be considered:

- Colocation constraints
- Device constraints: Can the node be executed on a specific device type?
- How much memory is left on a device

### 3.1.4 Partitioning Postprocessing

The required send and receive nodes have to be inserted for cross-device edges. When a node is assigned during partitioning all direct predecessor and successor nodes are checked if they are placed on a different device. If so, the node is added in a set of nodes with cross-device edges. After partitioning new nodes are inserted by making use of these sets. If a node has outgoing cross-devices edges to the same device only one send-receive-node-pair is inserted as described in Figure 2.2 in Section 2.4.

### 3.1.5 Scheduling Preprocessing

Depending on the scheduling algorithm various preprocessing computations are necessary. If this is the case, they are described in the scheduling chapter 5.

### 3.1.6 Scheduling

The scheduling algorithm determines which node is executed next. In the system model only non-preemptive scheduling is supported which means that a node is executed completely once started. The scheduling algorithm has to consider the data dependencies in the dataflow graph. A node can only be executed when all its direct predecessor nodes have finished computation.

A node's calculated data is kept in RAM as long as its direct successor nodes have not started computation. Therefore, a mechanism similar to reference counting has been implemented to find out when the data is no longer needed and can be deleted. A counter is initialized on the number of direct successors. This counter is decremented

each time a direct successor node executes. If the counter reaches zero the result can be deleted.

### 3.1.7 Evaluation

Multiple metrics are possible to evaluate the scheduling and partitioning strategies. The first goal is to fulfil all the required constraints like device, colocation and memory constraints as a violation leads to an invalid partitioning. To measure the quality of the partitioning and scheduling algorithms, two metrics are taken into account: The network traffic and execution time which were defined in Section 2.3 The results of a computation are stored in csv-files and automatically visualized by matplotlib in python scripts.

## 3.2 Timing Behaviour

When simulating the scheduling, synchronization and communication between the devices' local time has to be considered to ensure the correct simulation of an execution pass. One approach might be to use the real world time as time synchronization mechanism. Consequently, the whole simulation would have to be implemented in a distributed system with all the programming overhead e.g. for message exchange. It would also be difficult to run the graphs exactly with the specified values like the transfer rate or to change certain values to evaluate different situations. Using real world time as synchronization mechanism leads to another disadvantage: One has to run the node computation in real time which could lead to a long evaluation duration.

For these reasons, a simulated timing behaviour is preferable. However, if each device has its own local timestamp it is necessary to avoid messages arriving in the future which would have had an influence on past decisions.

To illustrate this problem an example graph is provided in Figure 3.2. The node's upper value is the id and the lower value represents the node's #ops. The edges are labelled with their tensor sizes. In this example a parallel simulation is assumed without any synchronization mechanisms. The transfer rates and device speeds are $\frac{1}{EU}$. The simulation starts with device 2 and computes 50 EU as execution duration for node 7. The data arrive time of node 7's output tensor is computed to be at 80 EU. Device 1 becomes active in the simulation, executes node 5 and sets its local timestamp to 110 EU.

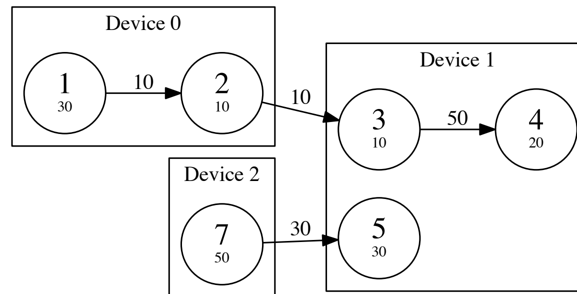



**Figure 3.2:** Sample graph labelled with node id, #ops and edge weigths

However, device 0 has simulated the execution of node 1 and 2 with an execution duration of 40 EU. The data is calculated to arrive at device 1 after 50 EU. The problem is that the local timestamp of device 1 is already at 110 EU in the simulation. Nevertheless, device 1 would have been able to execute node 3 after 50 EU if the simulation would have been implemented correctly. This is the reason why it is very important to have received all past data before making any scheduling decisions in the simulation.

To avoid making decisions without knowing all executable nodes at a specific timestamp the local timestamp of each device is collected. The device with the lowest local timestamp is selected to make the next scheduling decision. This algorithm has the disadvantage that the simulated scheduling can not be ran in parallel but also avoids the problem of making wrong scheduling choices or idle unnecessarily.

# 4 Partitioning

In this chapter various partitioning algorithms are described as well as their corresponding preprocessing steps. At first, the partitioning problem is defined formally. Afterwards, multiple partitioning strategies and their corresponding preprocessing steps are described.

## 4.1 Problem Formulation

A graph G = (V, E, C) consists of a set of vertices V = $\{v_0, v_1, ..., v_r\}$ and directed weighted edges $E \subset (V \times V \times \mathbb{N})$. As described in Section 2.2 a vertex is defined by an id, operation costs and a device constraint. The in C $\subset (V \times V)$ contained tuples $(v_x, v_y) \in$ C define vertices which have to be assigned to the same device.

The goal is to divide the graph's nodes onto n devices D = $\{d_1$, $d_2$, ..., $d_n\}$ into disjoint subsets. Devices are defined by an id, device type (CPU, GPU, TPU), memory size and speed. Abbreviations and functions are listed in Table 2.2. An assignment a: V $\rightarrow$ D has to be created by the partitioning algorithm meeting the following constraints:

1. $\forall (v_x, v_y) \in C : d(v_x) = d(v_y)$
2. $\forall v \in V : type(d(v)) = device\,constraint(v) \vee device\,constraint(v) = ALL$
3. $\forall d \in D : \sum\limits_{v \in V_{d_x}} size_{est}(v) < memory\,size(d)$

and minimizes together with the scheduling algorithm the following parameters:

1. execution time (previously defined in Equation 2.1)
2. network traffic (previously defined in Equation 2.2)

In the following, we call a colocation group or a not colocated node an assignment unit. Colocated nodes have to be mapped to the same device which is why we will assign them in one pass. Colocation groups and a not colocated nodes share the properties of a device constraint and memory demand.

A device is feasible for an assignment unit if:

1. The assignment unit's device constraint is equal to the device type or no device constraint exists.
2. The device has enough memory to be able to assign the whole assignment unit.

## 4.2 Hashing

Hashing is a very simple, well-known and fast placement strategy. Because of the colocation groups, memory and device constraints it is slightly modified to the following algorithm:

Iterate over colocation groups and try to assign them. For each colocation group a hash value is calculated based on the number of already assigned colocation groups modulo the number of devices. The device whose id is equal to the hash value is checked to be feasible or not. If a device is not feasible, search iteratively for the next feasible device and assign every node of the colocation group. After all colocation groups are assigned iterate over not colocated nodes and assign them to a feasible device in the same manner as the colocation groups.

## 4.3 HEFT

The Heterogeneous Earliest Finish Time (HEFT) [THW02] algorithm maps nodes to devices iteratively. It consists of two phases: The task prioritizing phase and the processor selection phase.

In the task prioritizing phase, the nodes are ranked due to their upward rank which describes the computation costs and data transfer costs from the given node to the "last" sink node. As soon as the upward ranks are computed the tasks are sorted in a descending order. In this order the tasks are assigned to processors within the processor selection phase.

In the processor selection phase the earliest finish time of a node for each device is computed in an insertion based manner. The maximum of the predecessor nodes' finish time and the required transfer time is taken as the earliest point of time in which the task can possibly start executing. Starting at this computed time each device is checked in order to find a time slot which is big enough to insert the task into. The node is assigned to the device with the lowest earliest finish time.

In order to meet the special constraints (e.g. colocation or device constraints) the HEFT algorithm was slightly modified. The earliest finish time is only computed on the

**feasible** devices, as a node can not be assigned to the other devices anyway. If the node is colocated, all colocated nodes have to be assigned to the same device. However, their predecessor nodes might not be assigned yet which is why the earliest finish time can not be computed for these nodes. So their computing time slot is only defined and added to the devices' time schedule when it is their turn due to the task prioritizing list.

### 4.3.1 Upward Rank

The upward ranks are an essential element in the task priorization phase. In the evaluation framework they are calculated in the partitioning preprocessing phase. The upward rank of a node $n_i$ is defined as the average computation costs $\bar{\omega_i}$ plus the maximum of the successors' upward ranks and average communication costs $\bar{c_{i,j}}$ of the edge from $n_i$ to $n_j$ (see [THW02]):

$$rank_u(n_i) = \bar{\omega_i} + \max_{n_j \in succ(n_i)} (\bar{c_{i,j}} + rank_u(n_j)) \tag{4.1}$$

The upward rank of a sink node $n_s$ is equal to $\bar{\omega_s}$. In the paper the average execution cost $\bar{\omega_i}$ for task i is calculated in a slightly different way, as they have estimated execution times for each task-processor pair. We define the average computation costs $\bar{\omega_i}$ as $\frac{\#ops(n_i)}{speed_{avr}}$, while $speed_{avr}$ is defined as the following:

$$speed_{avr} = \frac{\sum\limits_{d \in D} speed(d)}{|D|} \tag{4.2}$$

The calculation of the average communication cost is quite similar. The transferred data between two nodes is divided by the average transfer rate. The startup communication cost is considered in the computation costs of a send node. However, it is not processor specific at the moment. These average values are used as the placement decisions are not made yet.

## 4.4 Batch Split

In the Batch Split partitioning the graph's set of nodes is divided into different ranges. For each node the operations source rank as well as the operations sink rank is computed. The operations rank is used to sort the nodes in a descending order. The nodes of the critical path which should not be delayed are very likely to be all in the highest range. The nodes are assigned to the devices (also sorted based on their speed) with an assignment of the nodes of the highest range to the fastest device. Clearly, this is not

possible if a device is not feasible. Therefore, each node is assigned to the next feasible device which can be found in a similar way like in hashing. If a colocated node is in a range and is not already assigned yet, the whole colocation group is assigned as one unit.

The complexity of this algorithm is $O(n \times log(n))$ as the whole node set has to be sorted. The assignment of the nodes and calculation of the operations rank is possible in linear time.

### 4.4.1 Operations Rank

The operations rank is defined as sum of operations sink rank and operations source rank and is used to rate the “importance“ of a node.

Operations source rank counts the number of operations required to reach the given node from a source node and be able to execute it. This is expressed in Equation 4.3.

$$\text{operations source rank}(n_i) = \max_{n_j \in pred(n_i)} (\text{operations source rank}(n_j) + \#ops(n_j)) \quad (4.3)$$

Operations sink rank counts the number of operations required to execute the current node and the longest path to a sink node which is formulated in Equation 4.4.

$$\text{operations sink rank}(n_i) = (\max_{n_j \in succ(n_i)} \text{operations sink rank}(n_j)) + \#ops(n_i) \quad (4.4)$$

## 4.5 Iterated Critical Path

The basic idea for the Iterated Critical Path (ICP) strategy is to assign paths iteratively to devices as one unit. Thereby, we want to keep especially the first extracted critical path on the same device. This is only possible as long as there are no contradicting memory, device or colocation constraints. If so, the path has to be split.

This strategy is more computationally expensive than the other ones. The current critical path can be found in linear time but this procedure has to be repeated till no edge is left.

1. Get the critical path:
    a) Start at the source nodes and compute for each node the operations source rank. Extend the algorithm to set a path predecessor node for each node which is the one the maximum operations source node value is taken from.

b) Iterate over the sink nodes and take the one with the maximum operations source rank. This sink node is on the critical path.

c) Create a new path. Add the sink node to this path and follow the path predecessor nodes adding them to the path as well. Continue till there is no path predecessor node left which means that the whole path is traversed. Generate subpaths while adding nodes to the path based on already assigned nodes or contradicting device constraints. If a node is already assigned, it is not added to any subpath and finishes the current created subpath.

d) Remove the path's edges. Add the newly created sink or source nodes to the sink or source node sets.

2. For each subpath, check feasible devices and choose the device which has the minimum possible execution time exec$_{min}$ so far. As shown in Equation 4.5 exec$_{min}$(d) is equal to the number of operations assigned to device d divided by the device speed. Consequently, it stands for the minimal execution time possible if all nodes are executed sequentially without any idle times.

$$exec_{min}(d) = \frac{\#ops(d)}{speed(d)} \tag{4.5}$$

   If there is no feasible device further split the subpath in order to reduce the memory demand.

3. Continue until all nodes are assigned.

## 4.6 Critical Path

The Iterated Critical Path strategy is quite computationally intensive. It might not be necessary, to split the graph into paths until no path is left. For the above reason, the partitioning strategy Critical Path (CP) is introduced:

Compute the critical path based on upward ranks in the same manner as at the Iterated Critical Path partitioning. Assign the nodes of this path (as well as the nodes which are colocated with these nodes) on the fastest feasible device. Take the other nodes and assign them iteratively to the feasible device with the minimum possible execution time exec$_{min}$.

In comparison to the Iterated Critical Path, this strategy is much faster. Depending on the graph, many evaluations showed that there is a long critical path and all the other

extracted paths are pretty short. In these cases, it could be worth to save partitioning execution time, only assign the first extracted critical path and switch to a computationally less intensive method afterwards.

## 4.7 MITE

The idea of the partitioning strategy MITE is to consider multiple factors and use a heuristic function to assign the nodes. MITE stands for the different factors **M**emory, **I**mportance speed boost, **T**raffic and **E**xecution time. These factors aim to minimize the transferred data and the overall execution time. Moreover, devices which have proportionately much free memory in comparison to their total memory are preferred. Important nodes (based on their effect on the overall execution time) should be located on faster devices.

At first, the colocation groups are assigned and after that the not colocated nodes are placed. All feasible devices are checked and the assignment unit a (colocation group or not colocated node) is assigned to the device which minimizes the following function shown in Equation 4.6

$$\begin{aligned} heu(d_x, a) = & \text{traffic}(d_x, a) \times \\ & \text{execution time}(d_x, a) \times \\ & \text{memory}(d_x) \times \\ & \text{importance speed boost}(d_x, a) \end{aligned} \tag{4.6}$$

The traffic component is supposed to minimize the transferred data size, whereas the factor execution time should lead to a low overall execution time. The factor memory is supposed to favor devices with not much memory used, while importance speed boost influences important nodes to be mapped on faster devices. These factors are now explained more detailed.

As shown in Equation 4.7, $\text{traffic}_{temp}$ depends on the node to be assigned and the device $d_x$ to be checked as potential assignment device. The edge weight is divided by the transfer rate as a transfer of large data sizes is far worse for the execution time on a slow connection than on a high speed connection.

The transferred tensor size of all direct predecessor and successor nodes is considered, as long as they are already assigned and not located on the same device. One might also consider nodes which are not assigned yet. At the moment a not assigned node leads to zero increase in the traffic but is definitely worse than a node located on the

same device which also results in a zero increase. There are different possibilities how a not assigned node can be handled. At the moment the best case placement (the one on the same device) is assumed. Another option would be the worst case placement which would be on the device with the slowest connection on which no send-receive-node-pair for this data transfer exists yet. Moreover, one could take simply the average.

$$\begin{aligned} traffic_{temp}(d_x, n_i) = & \sum_{\substack{n_j \in pred(n_i) \\ d(n_j) \neq d_x \\ d(n_j) \neq -1}} \left(\frac{weight(n_j, n_i)}{transfer\,rate(d(n_j), d_x)} \times cond_{pred}(n_j, d_x)\right) \\ & + \sum_{\substack{n_j \in succ(n_i) \\ d(n_j) \neq d_x \\ d(n_j) \neq -1}} \left(\frac{weight(n_i, n_j)}{transfer\,rate(d_x, d(n_j))} \times cond_{succ}(n_i, d(n_j))\right) \end{aligned} \tag{4.7}$$

However, the aggregation of send and receive nodes has to be considered for the traffic which is the reason for $cond_{pred}(n_j, d_x)$ and $cond_{succ}(n_i, d_j)$.

The idea of $cond_{pred}(n_j, d_x)$ (as shown Equation 4.8) is that only the weights of edges with predecessor nodes matter which have not already another successor node on the device $d_x$. If there is another successor on $d_x$, there was already a send-receive-node-pair inserted and no additional data would have to be transferred.

$$cond_{pred}(n_j, d_x) = \begin{cases} 0 & \text{, if predecessor node } n_j \text{ has already another successor node on device } d_x \\ 1 & \text{, otherwise} \end{cases} \tag{4.8}$$

$cond_{succ}(n_i, d_j)$ (as shown Equation 4.9) checks if there is already another successor node of $n_i$ on device $d_j$. If so, no additional send-receive-node-pair is needed.

$$cond_{succ}(n_i, d_j) = \begin{cases} 0 & \text{, if another successor node of } n_i \text{ is on device } d_j \\ 1 & \text{, otherwise} \end{cases} \tag{4.9}$$

If a is a colocation group the sum of the member nodes' traffic has to be taken as expressed in Equation 4.10.

$$\text{traffic}_{temp}(d_x, a) = \begin{cases} \text{traffic}_{temp}(d_x, n_i) & \text{, if a is the not colocated node } n_i \\ \sum\limits_{n_i \in V_g} \text{traffic}_{temp}(d_x, n_i) & \text{, if a is the colocation group } g \end{cases} \quad (4.10)$$

However, $\text{traffic}_{temp}(d_x, a)$ can not be used directly as it has to be normalized. This is achieved by dividing by the maximum traffic$temp$ of all feasible devices. Obviously, this can only be done if the maximum traffic is not zero. If $\text{traffic}_{temp}$ is zero, the traffic (as shown in Equation 4.11) is set to a very small number in order to avoid that all other factors in Equation 4.6 are not considered anymore. We define $\hat{t}$ as the maximum $\text{traffic}_{temp}$ value of all feasible devices.

$$\text{traffic}(d_x, a) = \begin{cases} 0.000001 & \text{, if traffic}_{temp}(d_x, a) = 0 \wedge \hat{t} \neq 0 \\ 1.0 & \text{, if } \hat{t} = 0 \\ \frac{\text{traffic}_{temp}(d_x, a)}{\hat{t}} & \text{, otherwise} \end{cases} \quad (4.11)$$

The factor execution time shall minimize the overall execution time. Therefore, the number of already assigned operations is taken in addition to the number of operations of the checked assignment unit. The sum is divided by the device speed and the result is the minimal execution time of this device if the assignment unit would be on this device. This is expressed in execution $\text{time}_{temp}$ as shown in Equation 4.12. The idea is that the overall execution time is minimized if the maximum execution time of all devices is minimized.

$$\text{execution time}_{temp}(d_x, a) = \begin{cases} \frac{\#ops(d_x) + \sum_{n_i \in V_g} \#ops(n_i)}{speed(d_x)} & \text{, if a is the colocation group g} \\ \frac{\#ops(d_x) + \#ops(n_i)}{speed(d_x)} & \text{, if a is a not colocated node } n_i \end{cases} \quad (4.12)$$

For normalization, the execution $\text{time}_{temp}$ has to be divided by the maximum execution $\text{time}_{temp}$ $\hat{e}$ of all devices. If this $\hat{e}$ is zero, the execution time is set to 1.0 as shown in Equation 4.13.

$$\text{execution time}(d_x, a) = \begin{cases} 1.0 & \text{, if } \hat{e} = 0 \\ \frac{\text{execution time}_{temp}(d_x, a)}{\hat{e}} & \text{, otherwise} \end{cases} \quad (4.13)$$

As devices with not much memory used should be favored, the factor $\text{memory}(d_x)$ simply calculates the portion of used memory for device $d_x$. If there is no memory used on

device $d_x$ yet, memory$(d_x)$ is set to one tenth of the minimal non zero memory portion $\tilde{m}$ of all devices as expressed in Equation 4.14. $\tilde{m}$ is initialized to 1.

$$\text{memory}(d_x) = \begin{cases} \frac{\tilde{m}}{10.0} & \text{, memory portion of } d_x = 0 \\ \frac{\text{used memory of } d_x}{memory\,size(d_x)} & \text{, otherwise} \end{cases} \tag{4.14}$$

The factor importance speed boost shall lead to the assignment of important nodes on fast devices. The more important the node or the colocation group is the closer is importance(a) (shown in Equation 4.16) to 1. The faster a device is the closer is speed$(d_x)$ divided by the speed $\hat{f}$ of the fastest feasible device to the maximum of 1. As the device which minimizes the Function 4.6 is used for the assignment, the product has to be subtracted by 1 as shown in Equation 4.15. In general, faster devices should be favored by this factor which is why we did not simply take the absolute value of the difference between the importance and speed factor.

$$\text{importance speed boost}(d_x, c) = 1 - (\text{importance}(c) * \frac{\text{speed}(d_x)}{\hat{f}}) \tag{4.15}$$

The importance differs if a is a colocation group or a not colocated node and is defined in Equation 4.16. The importance of a node is given by its operations rank and divided by the maximum operations rank $\hat{o}$ which is given by nodes on the critical path.

$$\text{importance}(a) = \begin{cases} \frac{\sum\limits_{n_i \in V_g} operations\,rank(n_i)}{\hat{o} \times |V_g|} & \text{, if a is the colocation group } g \\ \\ \frac{operations\,rank(n_i)}{\hat{o}} & \text{, if a is a not colocated node } n_i \end{cases} \tag{4.16}$$

## 4.8 Depth First Search

Another partitioning strategy is Depth First Search (DFS) partitioning. The idea behind this strategy is to traverse the graph in a depth first search and assign the nodes based on a heuristic function.

The algorithm performs the following steps:

Sort the source nodes in a decreasing order based on their operations rank. Traverse the graph in depth first search starting at the beginning of the sorted list. Consequently, the algorithm starts with a source node which is on the critical path.

Perform the depth first search with the help of an already-visited-boolean and a recursive function. If a node is not assigned yet, assign the whole assignment unit. Place the assignment unit on the device minimizing the product of execution time and traffic which was defined in the MITE strategy.

# 5 Scheduling

As illustrated in Chapter 1, scheduling plays an important role in minimizing the overall execution time. In this chapter we define the scheduling problem and present three scheduling algorithms.

## 5.1 Problem Formulation

A scheduling algorithm defines the execution order for each device. The function $ex_order : V_d \rightarrow S$ maps all the device's vertices on natural numbers which define the order of execution. The set S does not contain duplicates and is defined as $S = \{n \in \mathbb{N} | \, n \in S \wedge y \in S \rightarrow n \neq y\}$.

For a correct scheduling the following criteria have to be fulfilled:

1. A node is executed only once.
2. Each node is executed.
3. Each device can execute at most one node simultaneously.
4. Once a device has started to execute a node, it can not be interrupted. ($\rightarrow$ non-preemptive scheduling)
5. A node n can be only executed as soon as all tensors of its direct predecessor nodes are computed and transferred to the device d(n). When all data is available, the node is called an executable node.
6. A device can only execute nodes which are placed on this device. A node can not be on multiple devices.
7. A device can only compute nodes if there is an executable node on this device. Idling is possible.

The overall goal is to minimize the makespan as defined in Equation 2.1. The graph is already partitioned when the scheduling starts. As the scheduling algorithm has no influence on the number of cross-device edges and every node is executed exactly once, the traffic parameter is independent from the chosen scheduling strategy.

## 5.2 FIFO Scheduler

For each device, the nodes which can be executed are stored in a collection. Whenever, a new node becomes executable it is added to the collection and is removed as soon as it was executed. At First In First Out (FIFO) scheduling the collection is a queue. Devices execute the nodes in the same order the nodes have been added to the queue.

## 5.3 PCT Scheduler

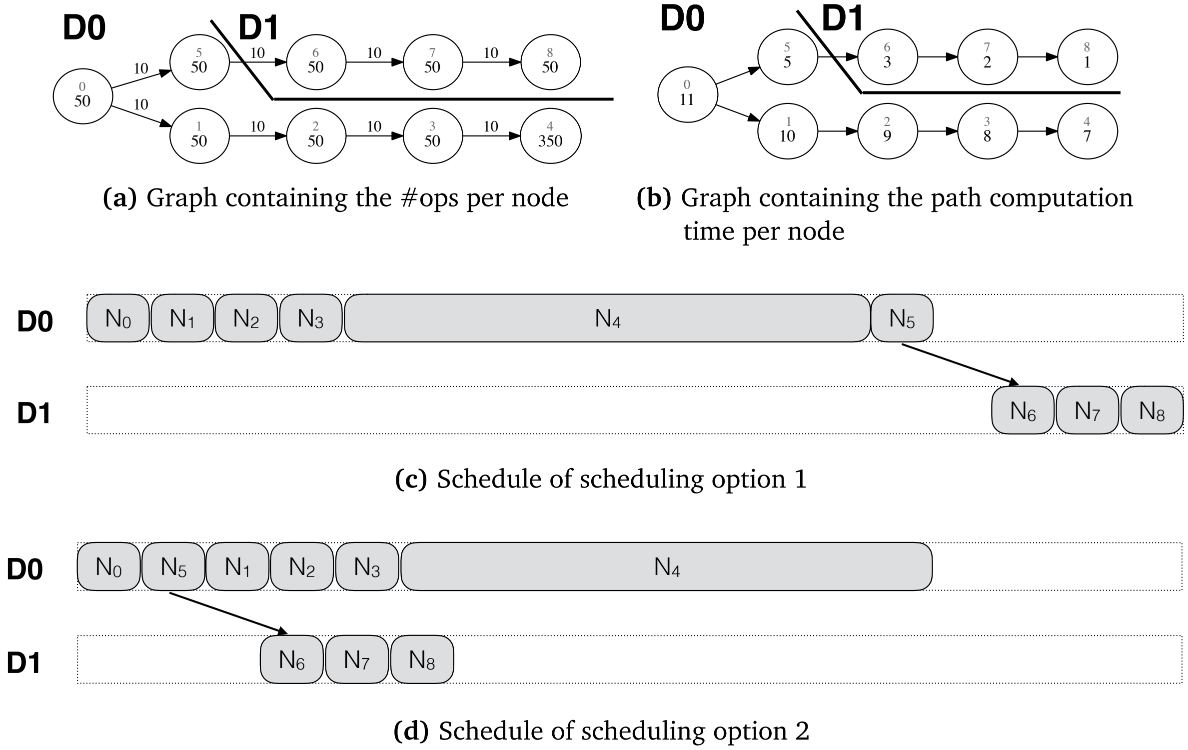
The upwards Path Computation Time first (PCT) scheduler executes the node first whose longest path of direct and indirect successors takes most time to compute. This can be achieved by selecting the executable node with the maximum upwards path computation time. The (upwards) path computation time (PCT) is calculated starting from the sink nodes as shown in Equation 5.1. The path computation time value of a sink node $n_i$ is the time it takes to execute $n_i$ on the device it was assigned to. If the node is not a sink node one has to consider the maximum of the time to transfer the data to one of its successor nodes as well as the path computation time of the same successor node additionally. The transfer time is 0, if both nodes are placed on the same device. Otherwise, it is the edge weight divided by the transfer rate between the two devices.

$$\text{PCT}(n_i) = \max_{n_j \in succ(n_i)} (\text{PCT}(n_j) + \text{transfer time}(n_j, n_i)) + \frac{\#\text{ops}(n_i)}{\text{speed}(d(n_i))} \tag{5.1}$$

The path computation time is calculated once in linear time during the scheduling preprocessing.

## 5.4 MSR Scheduler

The PCT scheduler only considers the path computation time for its scheduling decisions. However, if a lot of nodes depend on a specific node it could be better to schedule this




**(a)** Graph containing the #ops per node

**(b)** Graph containing the path computation time per node

**(c)** Schedule of scheduling option 1

**(d)** Schedule of scheduling option 2

**Figure 5.1:** Scheduling strategies

node first even if another one has a larger path computation time. This is especially the case if another idle device becomes active as a successor node becomes executable.

Schedules created by two different scheduling strategies on the same graph are illustrated in Figure 5.1. The assumed device speed is 50 $\frac{Ops}{EU}$, while the transfer rate is $\frac{10}{EU}$. Figure 5.1 (a) shows the dataflow graph annotated with node and edge weights. In Figure 5.1 (b) the nodes are labelled with their path computation time values. The schedule created by the PCT scheduler is visualized in 5.1 (c). Clearly, the schedule in Figure 5.1 (d) would be better as the overall execution time is much smaller.

Therefore, the new scheduling strategy Maximum Successor Rank first (MSR) is introduced in which nodes are selected based on the function successor rank in Equation 5.2. Each direct successor node of $n_i$ increases the successor rank by one. $n_i$ gets an additional point for each successor node which is not assigned to the same machine and another one for each successor becoming executable after computing $n_i$. This is formulated by $\text{pred}_{notex}(n)$ which contains the not executed predecessor nodes of n. After executing the last not executed predecessor node, the node becomes executable. The last term is an increase of 5 for each successor which becomes executable on an idle device. This is rewarded by so many points as a device becomes active again and

thereby probably reduces the overall execution time. The function idle(d) is 1 if device d is idle when making the scheduling decision. If not, it is 0.

$$\begin{aligned} \text{successor rank}(n_i) = \sum_{n_j \in succ(n_i)} ( & 1 \\ & + 1 * (d(n_j)\,!=\, d(n_i)) \\ & + 1 * (|pred_{notex}(n_j)| == 1) \\ & + (idle(d(n_j)) \wedge (|pred_{notex}(n_j)| == 1)) * 5\,) \end{aligned} \tag{5.2}$$

The idea is to favor the execution of nodes whose successor nodes can be executed afterwards and especially if it leads to a device becoming active. If all the rank values are equal, the path computation time is used as a tie breaker.

An obvious disadvantage of this strategy is that we have to check if another device is idle or not. This requires additional device communication while scheduling.

# 6 Sample Data

In order to evaluate partitioning and scheduling strategies sample data has to be collected. In this chapter typical model types of TensorFlow are described. Furthermore, extracted graph and cost models are analysed in order to generate realistic synthetic data.

Two kinds of graphs are used for the evaluation:

1. Real-world dataflow graphs extracted from TensorFlow and extended by required constraints.
2. Synthetic dataflow graphs

## 6.1 Typical Examples in TensorFlow

Typical machine learning models can be expressed in TensorFlow. In order to get a TensorFlow dataflow graph, a model has to be defined and exported [Sca16]. A typical model type is a linear model which is a weighted sum of variables assuming a linear relation between its input variables and output variable. It can be used to predict continuous values or to classify. In comparison to multi-layer networks, it can be quicker debugged and trained and works well with many features.

Another typical model type are convolutional neural networks. Convolutional neural networks suit well for data which can be structured as a grid in a way that related values are next to each other. Therefore, they are often used for image classification [CZG+16] [KSH12] and contain convolutions as the name suggests. Convolution is a mathematical operation that takes two functions as an input and provides another function as output. Practically, one can think of a convolution for example as sliding a kernel over the pixels of an image.

Recurrent neural networks are another typical model type and are for example used for language modeling [MAP+15] [CW08]. They often consist of multiple layers with recurrent connections.

## 6.2 Data Generation and Sample Data

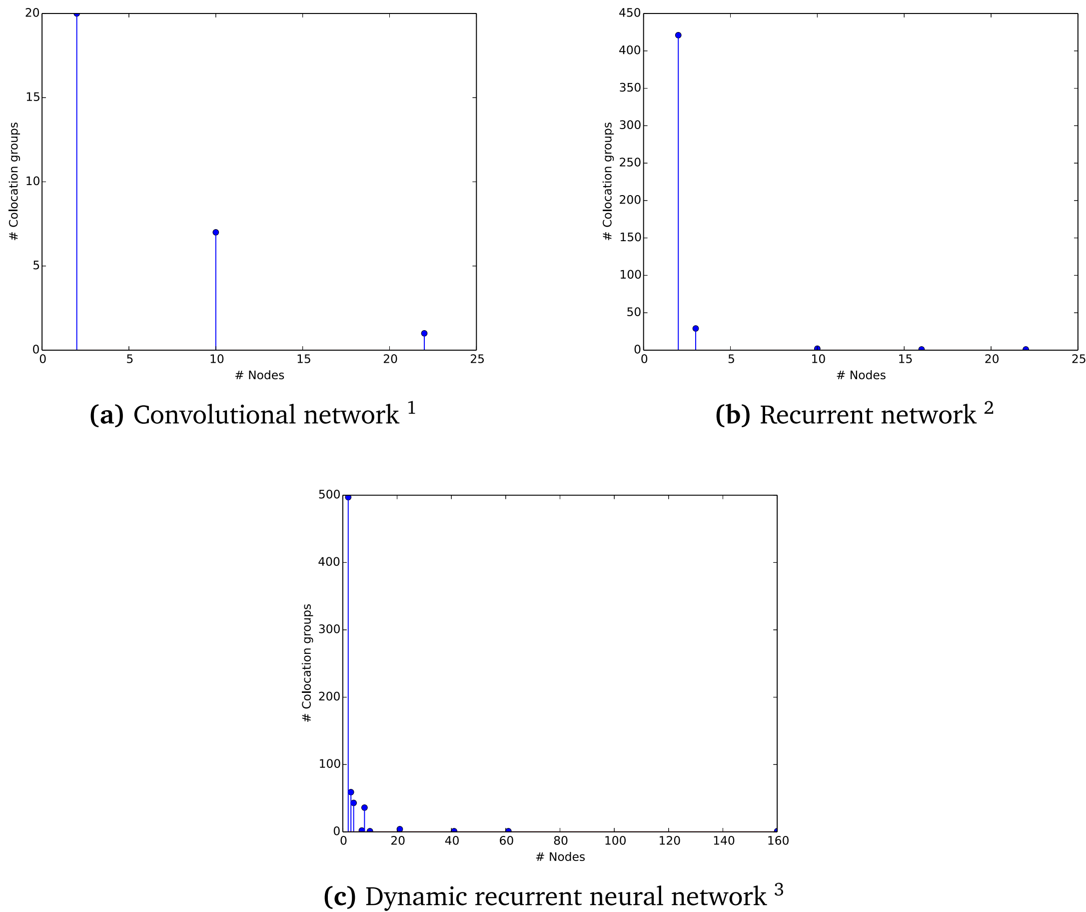
To evaluate partitioning and scheduling algorithms, sample data has to be collected. Therefore, different models in TensorFlow are generated and the graph as well as the cost model is extracted. Different configuration options have to be set that the cost model can be extracted. Information about the graph can be received by calling “tf.get_default_graph().as_graph_def()“ in python.

The extracted graph data contains the name and id of existing nodes, how these nodes are connected (“input: “Variable_1““) and which nodes are colocated (“loc:@Variable_1“). The cost model contains information about the compute costs. TensorFlow also collects information about the nodes’ memory sizes or edges’ tensor sizes but they are not easily extractable yet. For the evaluations the data is extended by the tensor sizes, #ops, device constraints and RAM demand.

As the TensorFlow sample models are not that large yet (∼5000 nodes) other data sources were required. A random graph generator was implemented in order to create directed acylic graphs provided with the required characteristics. These characteristics include the id, incoming and outgoing edges, colocated nodes, the tensor size of the output tensor, #ops, RAM demand and the device constraint for each node. A graph is generated in levels. As an edge is always from a node of a lower level to a node of a higher level cycles are avoided. Multiple ways to generate the edges were implemented:

When a new level was generated each node of the former levels is linked with a defined probability to each node in the new level. The drawback of this approach is the following: With each new level all existing nodes are linked to the latest inserted nodes with a defined probability. Hence, the lower the level the higher the probability for a node to have many edges. To get a more realistic distribution of edges, a limitation on the levels can be added e.g no edge can be between nodes with a distance higher than 5 levels. Alternatively, a function can be used to lower the probability of having an edge drastically the more distant two nodes are in terms of levels. In order to allow long range edges a method was implemented which randomly selects two nodes and adds an edge from the node with the lower level to the node with the bigger one. The edge is only added if the edge has not existed yet.

These typical graph characteristics depend on multiple input parameters: The number of levels as well as the minimum and maximum number of nodes per level. Furthermore, the minimum and maximum RAM demand to store a node can be defined. In order to set the size of a node’s output tensor the minimum and maximum tensor size can be provided. The number of operations for a node are set to be in a certain range as well. A node’s probability to have a GPU or CPU constraint is provided. Depending on how




**(a)** Convolutional network [1]

**(b)** Recurrent network [2]

**(c)** Dynamic recurrent neural network [3]

**Figure 6.1:** Size and number of colocation groups

the edges are generated the edge probability, an edge level limit or a function defining the edge probability can be set.

In order to add realistic colocation constraints we analysed the colocation groups in the graphs extracted from TensorFlow.

[1]https://github.com/aymericdamien/TensorFlow-Examples/blob/master/examples/3_NeuralNetworks/convolutional_network.py (last followed on 28.02.2017)
[2]https://github.com/aymericdamien/TensorFlow-Examples/blob/master/examples/3_NeuralNetworks/recurrent_network.py (last followed on 28.02.2017)
[3]https://github.com/aymericdamien/TensorFlow-Examples/blob/master/examples/3_NeuralNetworks/dynamic_rnn.py (last followed on 28.02.2017)

Figure 6.1 shows the size and number of colocation groups in three different graphs extracted from TensorFlow. These graphs consist of many small colocation groups with just a few member nodes, while there are only few groups with many colocated nodes.

For the evaluations, the edge weights are set randomly in a range of 1 and 100 as well as the number of operations per node. The memory demand of a node is also set randomly within this range.

The device data is also stored in a csv-file. Each device consists of a device id, available RAM, speed (number of operations per execution unit), the device type as wells as a transfer rate to all other devices. The transfer rate is encoded as an upper triangular matrix as the transfer rate of a device with an other device is bidirectional and a device has zero costs to communicate with itself.

When generating the device data, the number of devices can be set as wells as the device type probability. For the evaluations, the probability to be a CPU device is 60 % and being a GPU device is 40 %. The device speed is in a random range of 10 and 100 operations per execution unit. The faster a device is the less memory it has. The transfer rate between the devices is set to a random number in the range of 10 and 60.

# 7 Evaluation

In this chapter the partitioning strategies are evaluated in combination with the aforementioned scheduling algorithms. At first, the algorithms are executed on synthetic sample graphs and later on three graphs extracted from TensorFlow. Further evaluations are done on only seven devices. For these executions, we analysed the device utilization in order to investigate the reason for differences in the total execution time.

## 7.1 Synthetic Graph Evaluations

Various synthetic graphs were generated in order to evaluate combinations of scheduling and partitioning strategies. The properties of these graphs are listed in Table 7.1. The graph's edges were generated on two different modes: The level based approach and a random method to allow long range edges. These techniques were further described in Chapter 6. The graphs vary for example on their number of nodes, density or portion of colocated nodes. The reason for these variations is that a strategy might perform well on certain graphs but worse on others.

Six different partitioning strategies were executed in combination with three different scheduling strategies on 100 devices. Some of the strategies are non-deterministic as e.g. the order of nodes being assigned might differ. In order to avoid false conclusions the visualized execution time is the average of 10 executions and the standard mean deviation is shown as a grey line on each bar.

**Table 7.1:** Properties of six synthetic graphs

| Graph | #nodes | #edges (total) | #random edges | #level based edges | #levels | Minimum nodes per level | Maximum nodes per level | Edge level limit | Average node degree | #colocated nodes |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 36319 | 16076 | 8003 | 8073 | 300 | 50 | 200 | 20 | 0,44 | 5200 |
| 2 | 18168 | 25148 | 5015 | 20133 | 300 | 20 | 100 | 20 | 1,38 | 2762 |
| 3 | 7475 | 49510 | 0 | 49510 | 1000 | 5 | 10 | 3 | 6,62 | 263 |
| 4 | 9899 | 42486 | 21099 | 21387 | 200 | 20 | 80 | 10 | 4,28 | 1945 |
| 5 | 10440 | 47414 | 23535 | 23879 | 200 | 20 | 80 | 10 | 4,54 | 4768 |
| 6 | 26887 | 107144 | 53423 | 53721 | 500 | 10 | 100 | 20 | 3,98 | 4214 |

Figure 7.1 shows the simulated execution time of these scheduling and partitioning combinations. If the partitioning and graph combination stays the same, the FIFO scheduling strategy performs almost always the worst. In few cases the Maximum Successor Rank first (MSR) scheduling is slightly worse. The upwards Path Computation Time first (PCT) scheduling outperforms the other two scheduling strategies in almost all cases. Only in combination with MITE and Hashing, the MSR scheduler showed minimal better results in some runs on graph 1 and 2. We keep in mind that the path computation time value is used as a tiebreaker in the MSR strategy. However, this could be an explanation for equal results but not for the better ones. Probably, the partitioned graphs 1 and 2 contain sections like shown in Section 5.4 which is why MSR performs sometimes better. A reason why it only performs sometimes better can be a varying placement created by Hashing or MITE. The node placement of these strategies depends on the node assignment order which is not deterministic in the evaluation framework.

On closer examination of the partitioning strategies, Batch Split and Hashing were in general the worst strategies on all sample graphs. On graph 1 and 2 the execution time of MITE was about two times higher than HEFT, Critical Path and Iterated Critical Path. On graph 5 and 6 the execution time of MITE was almost the same and even smaller on graph 3.

Figure 7.2 shows the traffic generated when performing the strategies on the six synthetic sample graphs. In general, graph 1 has the lowest traffic while it is slightly higher on sample graph 2. On graph 3, 4 and 5 there is also more traffic, while the generated traffic of graph 6 is the highest by far. The reason for this observation is pretty obvious when looking at the number of edges in Table 7.1. Graph 2 (with ∼25 000 edges) has about twice the number of edges than graph 1, while graph 3, 4 and 5 have about twice the number of edges than graph 2. Graph 6 has the highest number with 107 144 edges. Clearly, the number of edges has an influence on the traffic as more edges increase the probability of having cross-device edges.

When analysing the traffic results one also notices that the traffic stays the same as long as the graph and the partitioning strategy stay the same. The traffic is independent from the scheduling strategy as it only depends on the used cross-device edges and all nodes are scheduled exactly once for sure.

For every graph Hashing, HEFT and Critical Path partitioning performed the worst in terms of traffic. This is no surprise as Hashing simply distributes the nodes on different machines and HEFT only aims to reduce the overall execution time. Thereby, it does not try to minimize the size of transferred data. After the critical path is assigned, Critical Path partitioning only takes care of the execution time minimization in its assignment decision. This is the reason why this strategy showed these traffic results. Batch Split partitioning leads to lower traffic as the mapping of nodes of the same ranges might result in neighbouring nodes to be mapped to the same device. Obviously, this approach

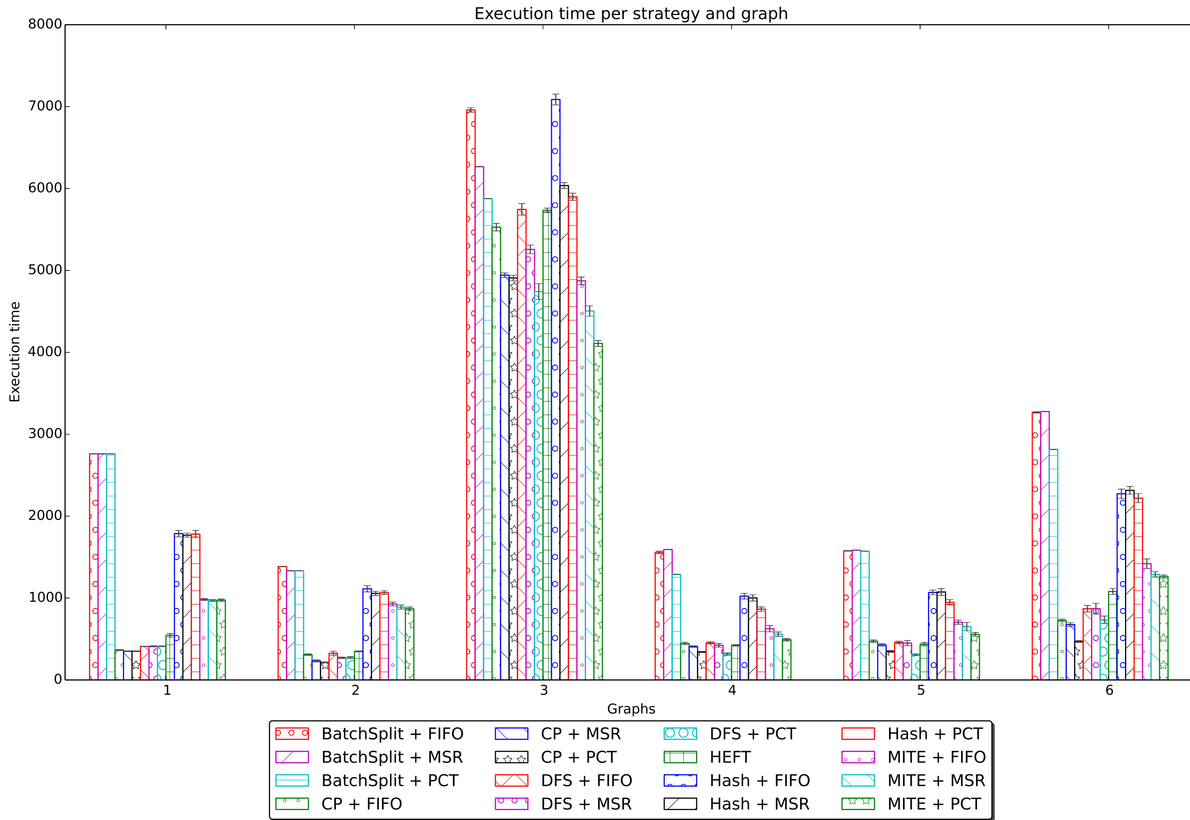



**Figure 7.1:** Execution time of scheduling and partitioning strategies on synthetic graphs

produces less cross-device edges resulting in lower traffic. When using MITE and DFS partitioning, a traffic factor is part of the assignment decision process. Therefore, it is logical that those two strategies performed well at the traffic evaluation.

The number of milliseconds it takes to partition the graph is shown in Figure 7.3. Executing HEFT took the longest by far. The differences in the execution duration of the HEFT algorithm correlates with the graphs' number of nodes as an insertion gap is searched for each node. Graph 1 has 36 319 nodes, graph 6 is the second largest with 26 887 nodes and graph 2 has 18 168 nodes. The next smaller ones are graph 4 and 5 and the least number of nodes has graph 3. The exact same ordering is given when looking at the partitioning execution times of the HEFT algorithm. MITE and DFS took comparatively long to partition graph 6. Normally, one would expect that the partitioning execution time of these strategies strongly depends on the number of nodes and number of devices, as the Function 4.6 is calculated for each assignment unit and device pair. However, Graph 6 is characterized by many edges. The traffic factor has to be computed in the Function 4.6. The time it takes to compute this factor highly depends on the number of edges.

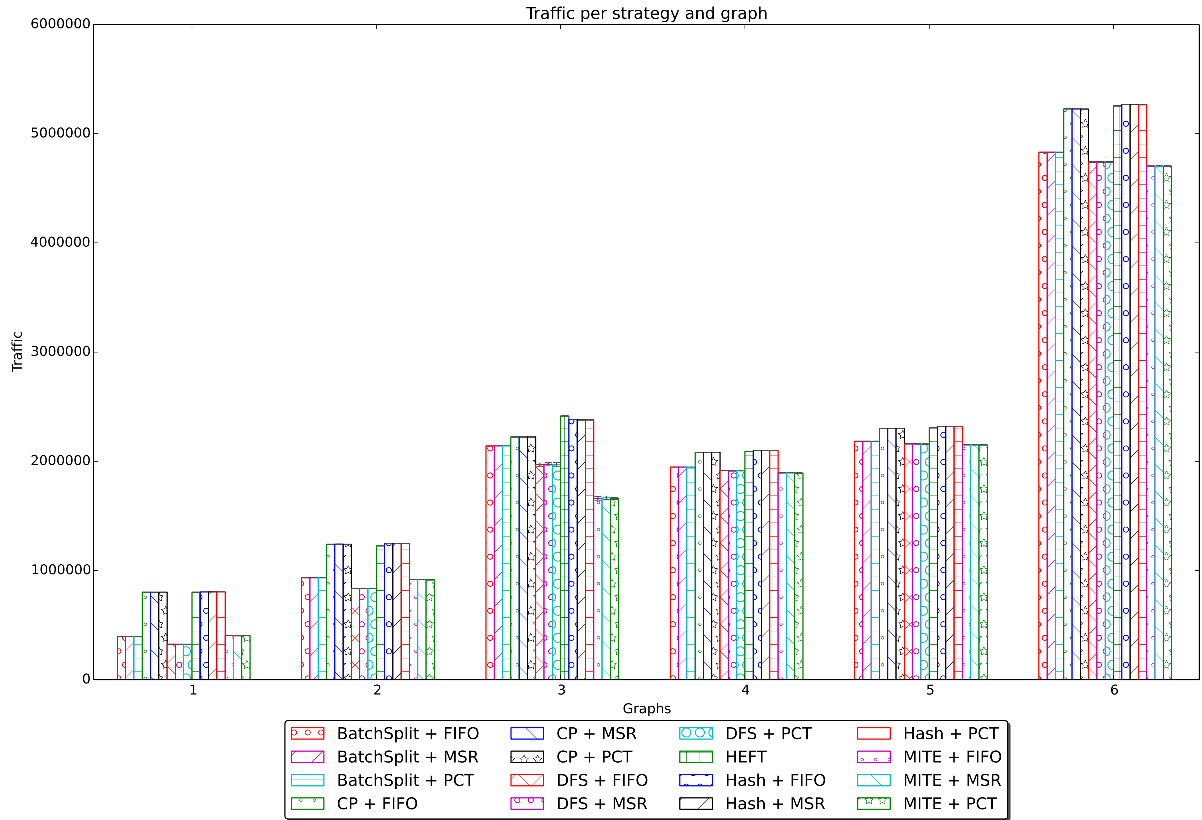



**Figure 7.2:** Traffic of scheduling and partitioning strategies on synthetic graph data

**Table 7.2:** Sample graphs extracted from TensorFlow

| Graph | #nodes | #edges | Average node degree | #colocated nodes | Additional remarks |
|---|---|---|---|---|---|
| convolutional _network | 347 | 531 | 1,53 | 104 | Convolutional network, extracted from TensorFlow |
| recurrent _network | 3069 | 5533 | 1,80 | 533 | Recurrent network, extracted from TensorFlow |
| dynamic_rnn | 5271 | 9214 | 1,75 | 1356 | Dynamic recurrent neural network, extracted from TensorFlow |

## 7.2 TensorFlow Graph Evaluations

The strategies were also compared on graphs directly extracted from TensorFlow. Table 7.2 lists the used graph data and shows graph characteristics like number of nodes, edges or colocations. These graphs are the same as used in Chapter 6 for the colocation visualization.

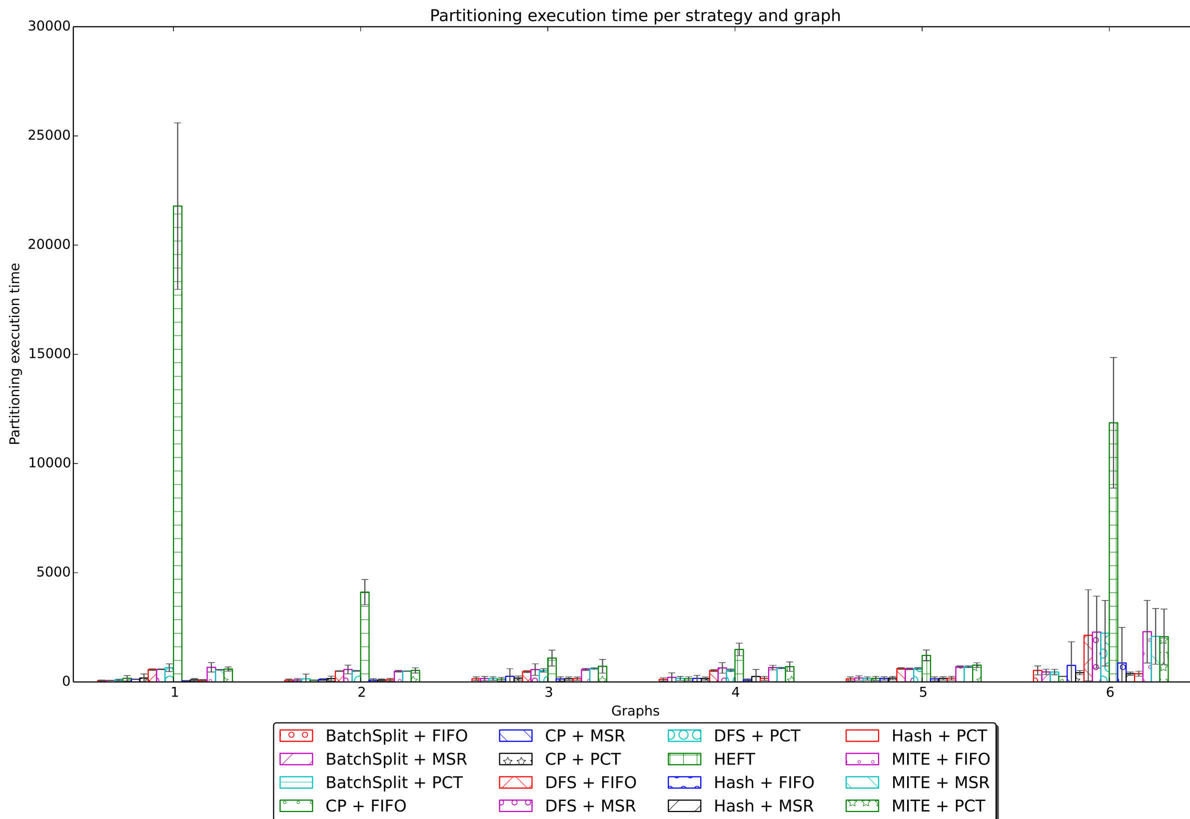



**Figure 7.3:** Partitioning execution time of scheduling and partitioning strategies on synthetic graph data

As the PCT scheduling leaded to the lowest execution time for all the partitioning strategies on every graph only this strategy is visualized in Figure 7.4. The computation was simulated for 50 devices. To avoid false conclusions based on randomness the average of 100 runs was taken as a result.

In Figure 7.4 the graph execution time is visualized. Clearly, Hashing performed the worst on all graphs. MITE and Critical Path partitioning showed the best results. DFS partitioning leaded to slightly worse results. It is very surprising that Iterated Critical Path is worse than Critical Path partitioning on each of the graphs. Critical Path only considers the critical path, while Iterated Critical Path splits the whole graph until each node was assigned to a path. The reason for the difference in the execution time performance might be the different path assignment strategies. Critical Path partitioning maps the critical path on the fastest device possible, while Iterated Critical Path takes the device with the minimal number of operations divided by the device speed. At the beginning, this value is zero for all devices. As the first extracted path is the critical one, this procedure might lead to the mapping of the critical path on a slow device. HEFT

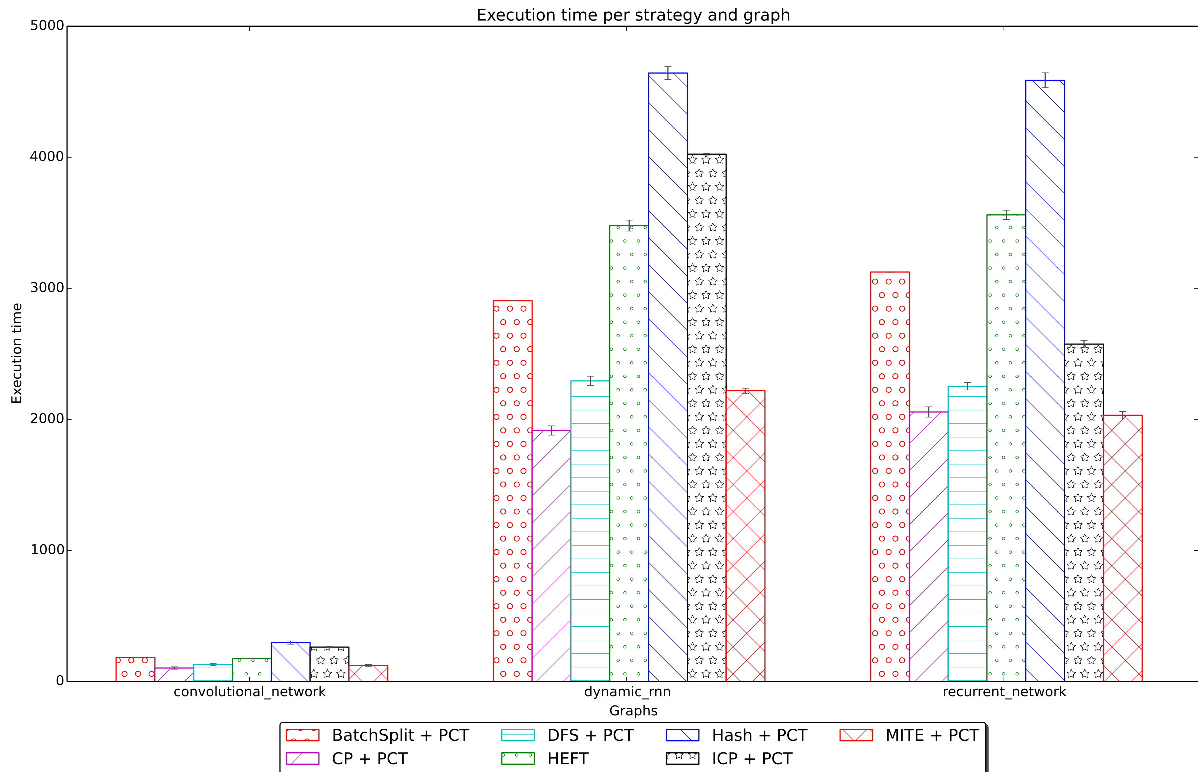



**Figure 7.4:** Execution time evaluation of partitioning and scheduling strategies for three graphs extracted from TensorFlow

was up to 75% and at least 45% worse than the results of the combination of MITE and PCT scheduling.

Figure 7.5 shows the traffic depending on the different strategies for each of the three graphs. Similar to the synthetic graphs, Hashing and HEFT performed the worst. Critical Path partitioning leaded to lower traffic. Iterated Critical Path showed better traffic results than Critical Path partitioning as all paths are computed and tried to be mapped to the same devices. As a path contains neighbouring nodes the repetition of this path assignments resulted in lower traffic. MITE, Batch Split and DFS partitioning performed best in terms of traffic. Batch Split showed better traffic results on the extracted TensorFlow graphs than on the synthetic ones. A reason for this could be a difference in the graph structure leading to more cross-device edges when running Batch Split on the synthetic graphs. Due to their assignment decision process, MITE and DFS partitioning performed very well as already mentioned before.

Another interesting evaluation parameter is the time required to partition the graph. This parameter is visualized in Figure 7.6. Clearly, Iterated Critical Path partitioning is the most computationally intensive strategy as all paths have to be computed and

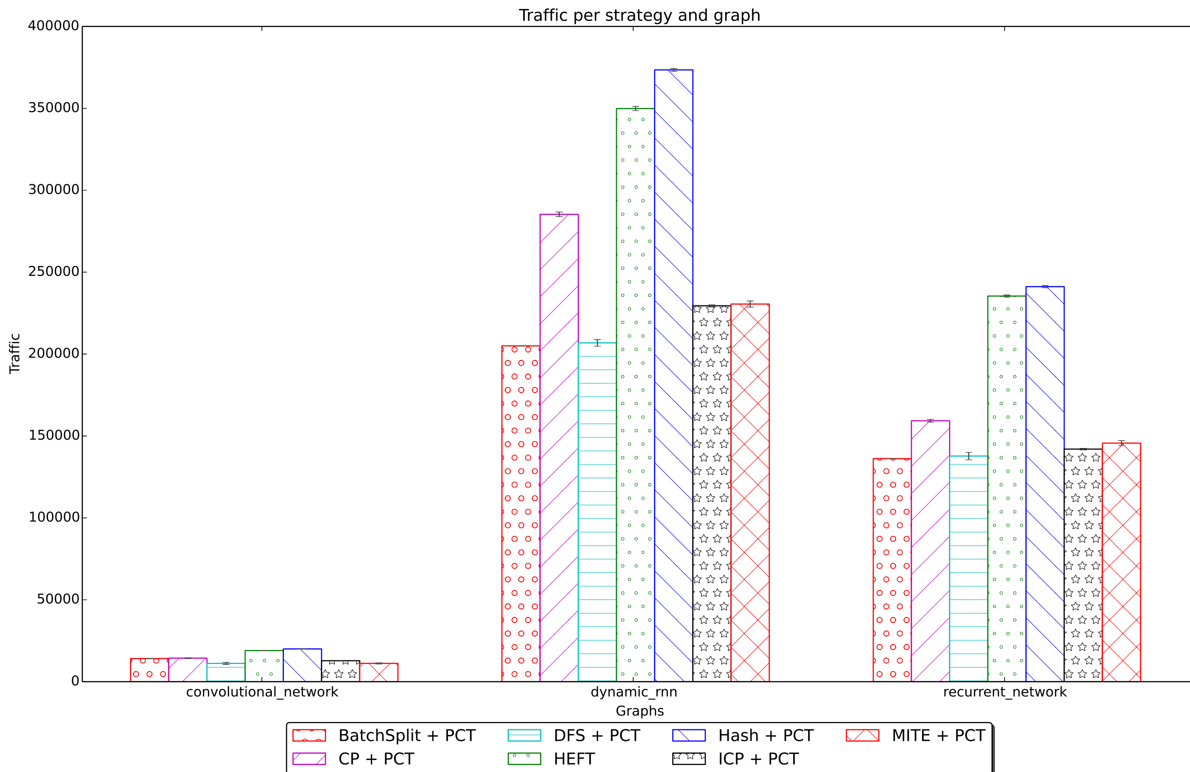



**Figure 7.5:** Traffic of partitioning and scheduling strategies for three graphs extracted from TensorFlow

assigned. Searching for a gap to insert a node into is also computing intensive. This is the reason why the HEFT algorithm has the second largest partitioning time. In comparison to the evaluations on the synthetic graphs the partitioning execution time of HEFT was comparatively low. This observation can be explained by the number of nodes which is far less in the TensorFlow sample graphs. Clearly, Hashing is quite fast as it partitions the node set based on a simple hash function. One might expect the Batch Split to be slower than MITE and DFS partitioning as all nodes are sorted depending on their operations rank. However, the size of the three sample graphs is not really large and the assignment decision is way faster in Batch Split partitioning. At MITE and DFS partitioning each device is checked for each assignment unit and various factors have to be computed for the feasible devices.

## 7.3 Evaluation on Few Devices

During the development of the partitioning and scheduling strategies the visualization of the device utilization provided much insight in how to improve the strategies. For a

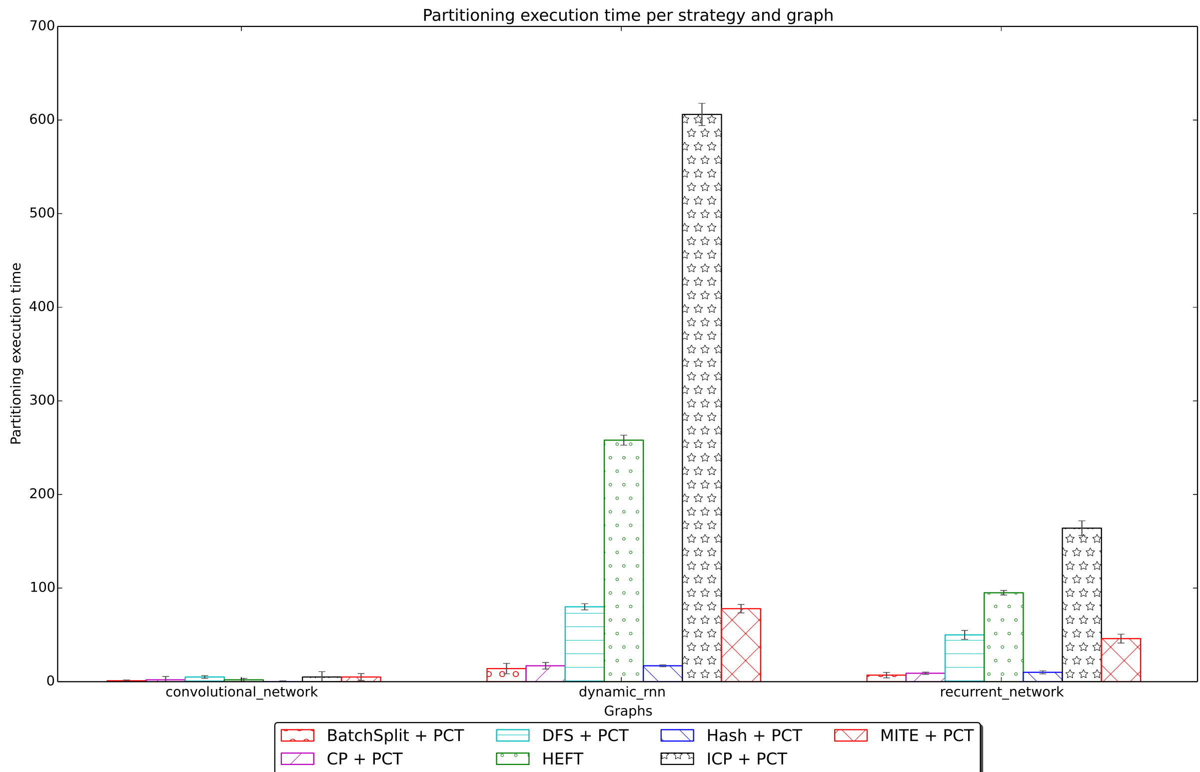



**Figure 7.6:** Partitioning execution time of partitioning and scheduling strategies for three graphs extracted from TensorFlow

better overview the next evaluations are only done on seven devices. The sample graphs 2 and 3 as well as two graphs extracted from TensorFlow are evaluated. At first, we compare the simulated execution time and visualize the results in Figure 7.7. Graph 2 was selected as MITE performed comparatively bad on this graph during the evaluations on the synthetic graphs. To further investigate the reason for this, we will take a closer look on the device utilization later.

Figure 7.7 reveals no surprises with respect to the execution times of the strategies. One of the few differences to the aforementioned evaluations is that the Batch Split performed comparatively worse on the dynamic_rnn for 7 devices than on 50. As Batch Split cuts the node set in equal sized parts for each device the number of devices can have a big impact on the performance of this strategy. Therefore, it is not really suitable for real-world usage as we do not know a good number of devices beforehand. The performance of other strategies is more robust to the arbitrary device number specification.

Graph 3 shows a similar distribution as before. Remarkable is the fact that the execution time is much lower in comparison to graph 2. This was the other way round in the

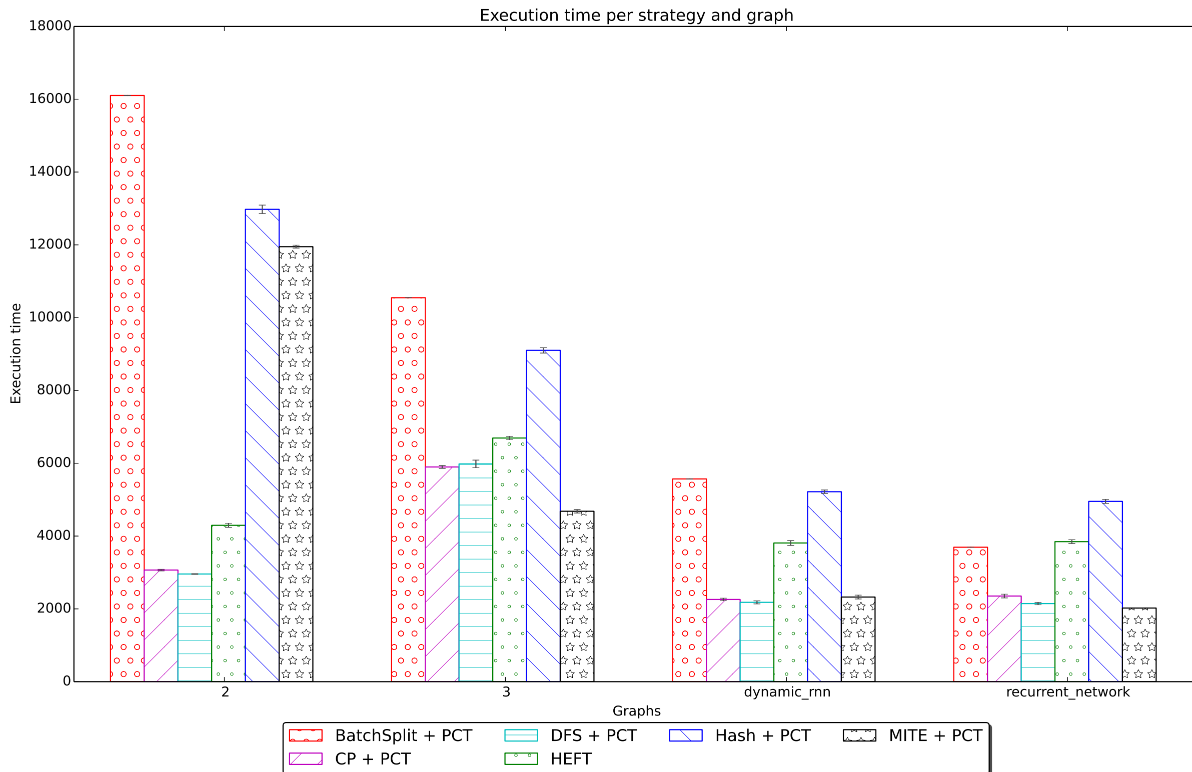



**Figure 7.7:** Execution time of scheduling and partitioning strategies on 7 devices

evaluation on 100 devices. Graph 2 has about twice the number of nodes than 3, but graph 3 is has about twice the number of edges. Graph 3 is a narrow but long dataflow graph. All these observations reveal that the executions on graph 2 can be better parallelized. This is the reason why the computation took longer on 7 devices but was faster on 100 devices in comparison to graph 3. Nevertheless, one can see in Figure 7.8 that the traffic on graph 3 is higher as it has much more (cross-device) edges.

Figure 7.9 (a) shows the utilization of the individual devices when performing the different partitioning and scheduling strategies on graph 2. Figure 7.9 (b) does the same for graph 3. Each time a node executes it is displayed as a bar. The length of the bar reveals the computation duration of the node. White gaps mean that the device was idle at this time period.

When comparing Figure 7.9 (a) with 7.9 (b) one notices that there are much more gaps at the execution of graph 3. Due to the narrow structure of graph 3 there are less nodes executable when the scheduler makes the decision which node is computed next. This leads to idle devices as they have to wait for nodes to become executable. This hypothesis can be confirmed by Figure 7.10 plotting the number of executable nodes for each device before making the decision which node is executed next. For every strategy

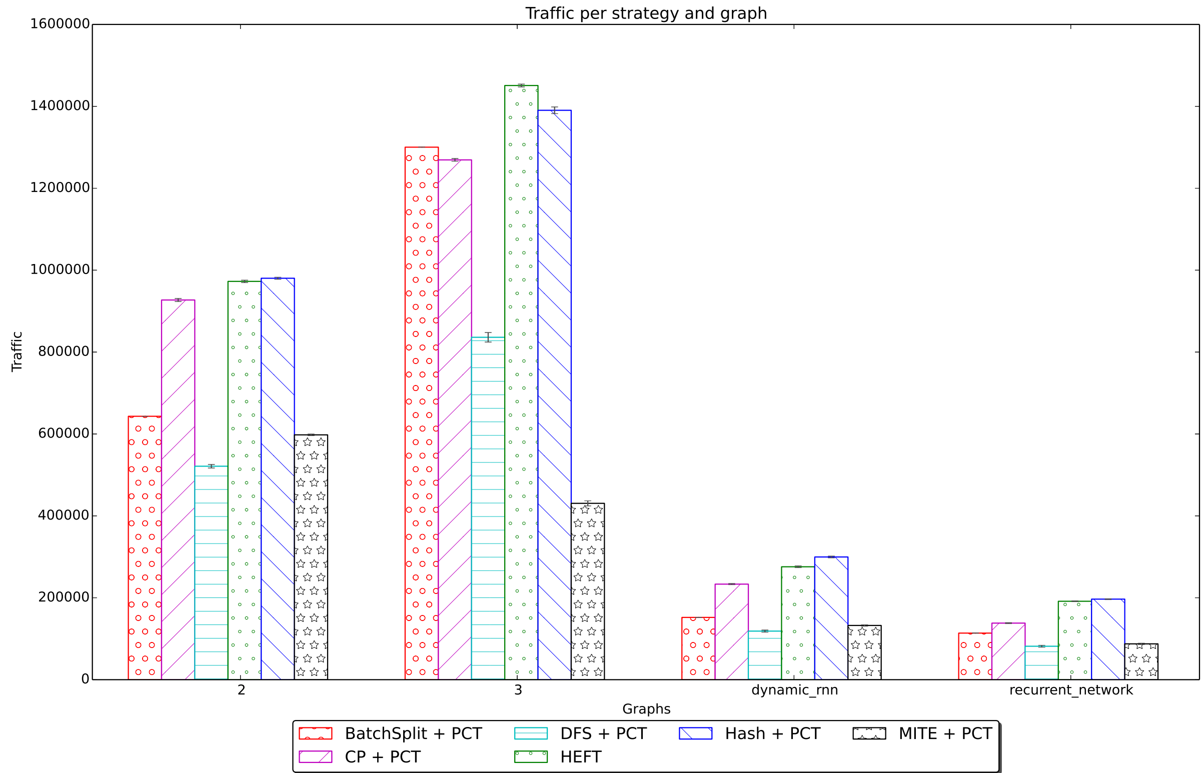
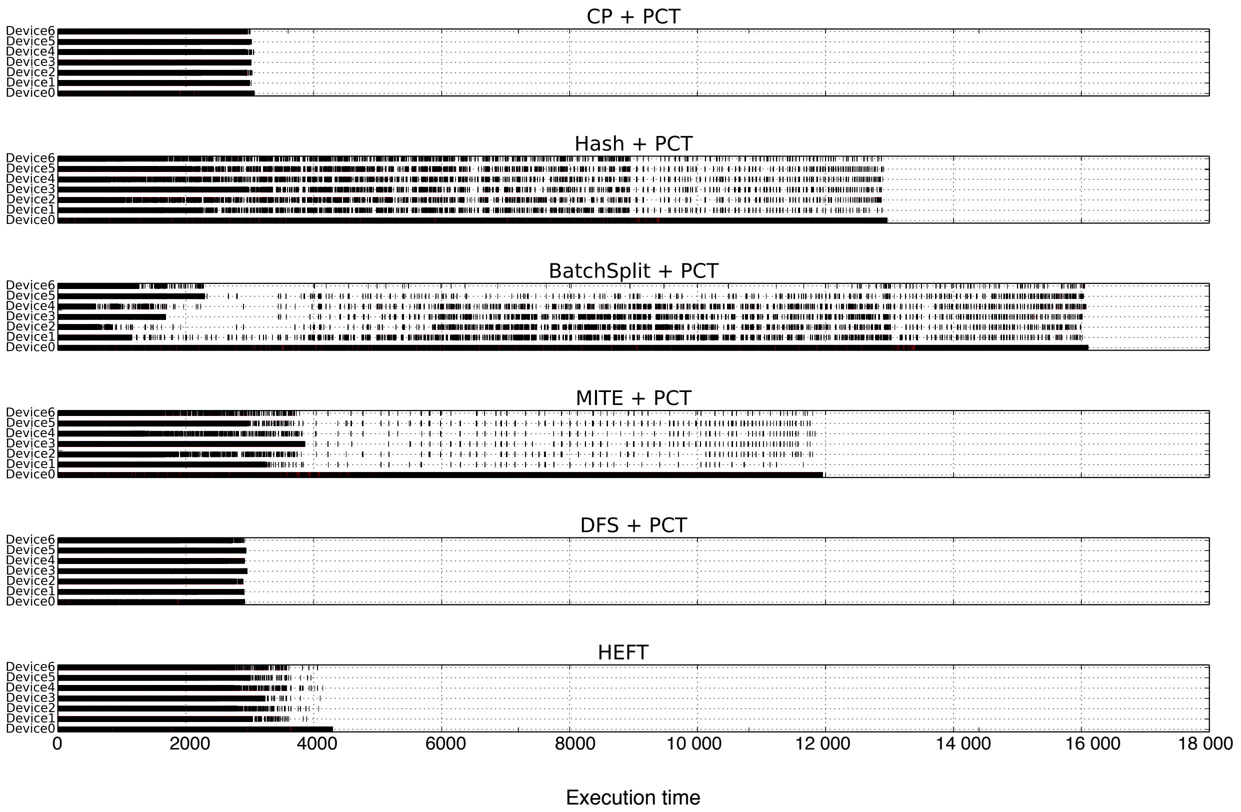


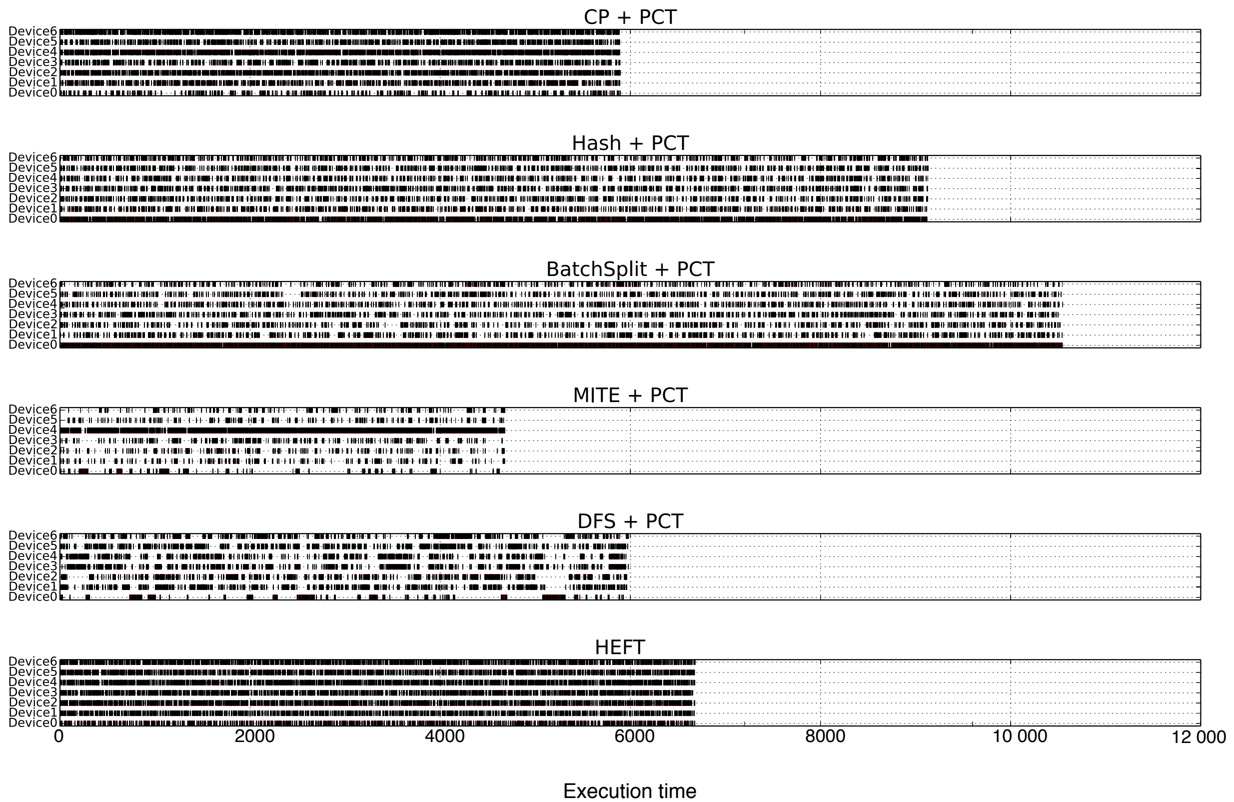

**Figure 7.8:** Traffic of scheduling and partitioning strategies on 7 devices

there were far more executable nodes on graph 2 when making the scheduling decision. Clearly, a lot of scheduling options is an indicator for less idle times on the devices as it is less likely that there is no executable node. Furthermore, this is an indicator that the dataflow graph computation can be parallelized.

In the utilization plot of Figure 7.9 (a) we also can find the reason for the bad performance of the MITE partitioning. Clearly, the computation load is not distributed well. Device 0 has to perform a lot of computations. Within the heuristic function of MITE neither the memory nor the execution time factor can be the reason for this as they have a positive impact on the computation and memory balance. Importance speed boost can not be the reason either as device 0 is the slowest of the three GPU devices. Therefore, the only possible reason for this imbalanced computation is the traffic factor. It is possible that a larger colocation group was assigned to device 0 influencing the assignment of even more nodes to this device by the traffic factor.

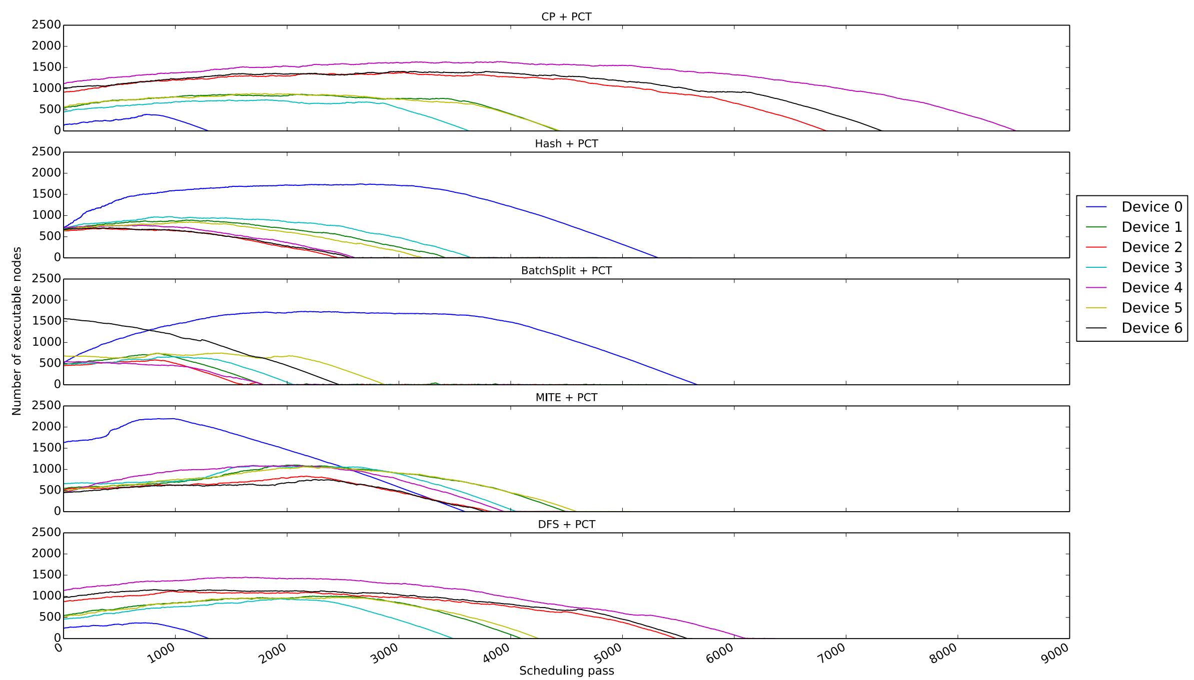



**(a)** Device utilization of graph 2

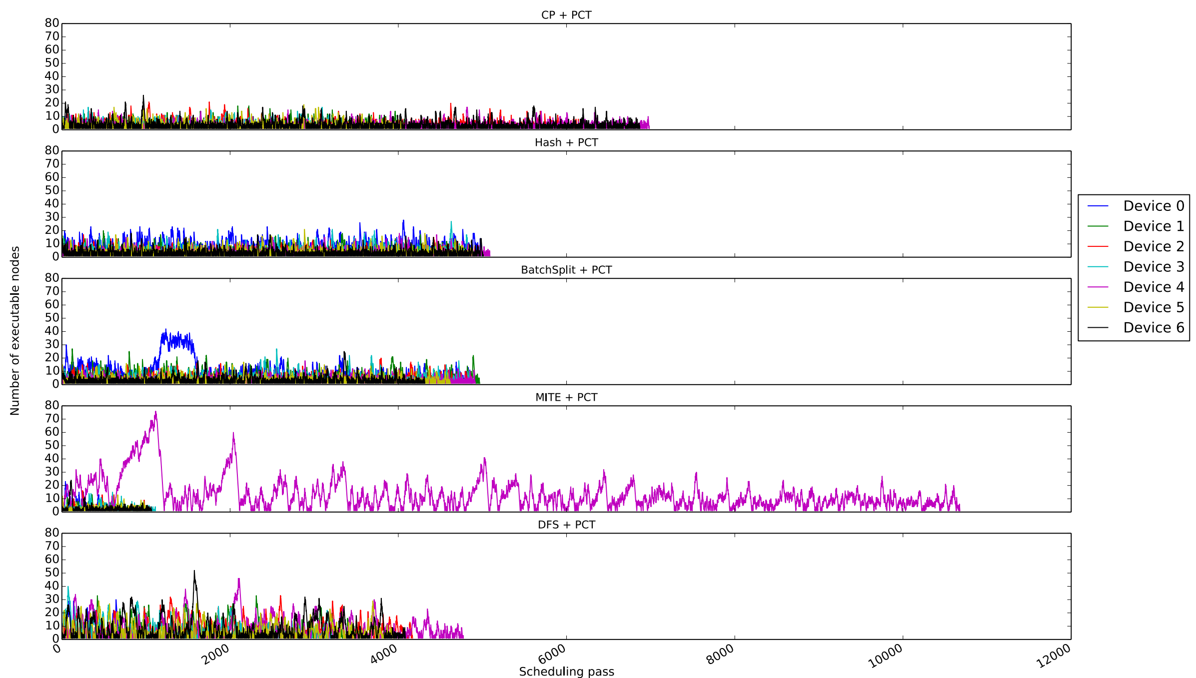



**(b)** Device utilization of graph 3

**Figure 7.9:** Device utilization for the scheduling and partitioning strategies




**(a)** Scheduling options for graph 2




**(b)** Scheduling options for graph 3

**Figure 7.10:** Number of executable nodes per device before making the decision which is node is executed next

# 8 Related work

Graph partitioning and scheduling strategies have been researched in the last decades. A lot of strategies have been investigated considering multiple constraints and system models. In this chapter the work which is particularly relevant to our work is presented.

The TensorFlow whitepaper [MAP+15] presents a placement strategy which is not published in the open source code yet. Their placement algorithm takes a cost model as an input. This cost model contains for example estimates of tensor sizes or the computation time for each node and is created based on heuristics or collected statistics from earlier executions. Starting with the source nodes the graph execution is simulated. Each node is greedily assigned to the feasible device on which it would finish its computation earliest. This heuristic function takes the communication costs as well as computation costs into account. Colocation constraints are considered by grouping nodes and intersect the feasible device sets. Our first plan was to use this strategy as a benchmark algorithm for our work. Unfortunately, the code for this algorithm is not published yet and the description is not precise enough to be able to reimplement it.

The placement algorithm has the same role as the partitioning in our model. Our partitioning algorithms also place nodes on devices, while the TensorFlow placement algorithm implicitly splits the node set by placing the nodes on different devices. The placement algorithm in TensorFlow is work in process according to the paper [MAP+15]. They propose to further investigate the use of machine learning to let the system learn how to make good placement decisions.

Topcuouglu et al. [THW02] addressed the static task scheduling problem with the HEFT algorithm. This algorithm places tasks (equal to nodes) on processors and defines the order of the executed tasks. They also aim to minimize the makespan (overall completion time), consider edge weights as wells as various possibly heterogenous transfer rates. The time to transfer data is also calculated by dividing the edge weight with the transfer rate and contains additionally the communication startup costs of a processor. In contrast, the task execution time can not be computed by dividing the number of operations with the device speed but is provided for each task-processor pair. Colocation constraints, device constraints, send-receive node aggregation and memory limitations are not considered. Another difference is that the node execution order is

defined before the execution starts in an insertion based policy. Although the HEFT strategy is rather old it performs quite well in terms of makespan and robustness when compared to twenty other heuristic based strategies [CJS+08]. This is the reason why it was used as a benchmark algorithm in our evaluation framework.

The HEFT algorithm can be extended to dHEFT [CRB+15] which shall solve the dynamic task scheduling problem and still aims to minimize the overall execution time. dHEFT and CATS are used as a benchmark for the strategies CPATH and HYBRID [CRC+16]. CATS, CPATH and HYBRID divide tasks in critical tasks and non-critical tasks, whereas the critical tasks should be assigned to a fast processor and the non-critical ones to a slow processor. The criticality-aware task scheduler (CATS) [CRB+15] considers task critical or not based on the bottom level. The bottom level is defined as the longest path (measured in the number of tasks) from the current node to a sink node. In this metric the fact that nodes can have heterogenous weights is not taken into account. This concept is extended in the critical path scheduler CPATH in which the criticality is evaluated based on task execution time. It calculates the so-called bottom cost. The Hybrid Criticality scheduler HYBRID considers the bottom level as well as the bottom costs for its decisions.

In comparison to our work, these approaches also have the concept of nodes become ready as soon as the results of all predecessors are conveyed. However, ready nodes are inserted in a non critical or critical ready queue. This is also the main difference to our approach: The node placement is not done until a processor is idle and takes a new task out of a ready queue. Another difference is that the required transfer time for the computation results is not taken into account in all the suggested dynamic solutions.

COLA [KHP+09] is an interesting iterative graph partitioning approach. This method also aims to minimize traffic and balances computation. Thereby, it reduces the overall execution time. COLA also supports heterogenous hosts and capacity limits. Moreover, it enables multiple user-defined constraints like colocation or even exlocation constraints.

# 9 Discussion

In this chapter we discuss possible future work as well as the drawbacks of the existing solutions.

For the creation of the colocation groups the whole graph data is read and kept in RAM. If we read the whole graph data anyway more information about the graph like number of nodes, number of edges or node degree could be collected. These information could be considered in a smarter partitioning or scheduling.

Another weakness of our model is that we assume to have knowledge about the computation costs and tensor sizes before partitioning. However, this knowledge has to be somehow collected. In the case of TensorFlow the number of possible operations is rather limited and therefore the computation costs could be estimated based on data collected in the past. If this is not possible, a simulated execution could generate the data but would lead to additional overhead.

Furthermore, we assume that all machines are inter-connected. In some settings some devices might not be able to communicate with each other. However, one simple modification might be to extend the definition of feasibility in a way that a device is only feasible for node placement if the direct predecessor and successor nodes can communicate with this device. Another option is to use certain devices as relay if two devices can not communicate directly.

In order to make the evaluations of the synthetic graphs more independent of random graph structure the sample graphs should be generated automatically in a larger number and generic way. If the strategies are executed on a large set of graphs we need different metrics like the Schedule Length Ratio [THW02] to still be able to compare them.

It is possible that the system is exposed to a lot of changes. For example the transfer rate might change or a connection could be lost completely. In TensorFlow, it is also possible that the set of nodes to be executed depends on varying variables. In these cases, it might be better to consider more dynamic solutions. Another advantage of dynamic approaches is that we could also perform a first partitioning based on unprecise estimations and adapt the node distribution during runtime. One example how this dynamic partitioning could be realised is the graph processing system Mizan [KAA+13]. It extends Pregel [MAB+10] for dynamic load balancing and introduces a dynamic

partitioning algorithm consisting mainly of monitoring and node migration planning in a distributed system.

# 10 Conclusion

In this thesis, we developed multiple partitioning and scheduling techniques. The goal was to reduce the overall execution time and traffic and consider constraints which were derived from real-world requirements of the machine learning library TensorFlow. Later, we compared and evaluated the developed algorithms.

At first, we developed a system model which conforms to the constraints derived from TensorFlow. Machine learning models can be represented as dataflow graphs. As these models can be very large they are often executed in a distributed system. Therefore, they have to be partitioned onto different devices. On these devices, it is necessary to define a node execution order during computation. These challenges were formulated in the modified scheduling and partitioning problem aiming to reduce the network traffic and the total execution time.

Multiple partitioning and scheduling strategies have been developed matching the additional constraints defined in the system model. To investigate the effects of various partitioning and scheduling strategies on the total execution time we implemented an evaluation framework to simulate the execution. Thereby, our results clearly showed that the combination of scheduling and partitioning strategies is quite important as a bad scheduling strategy can destroy the effects of a smart partitioning and the other way round.

In combination with almost every partitioning algorithm, the scheduling strategy upwards Path Computation Time first (PCT) outperformed the other investigated scheduling strategies by far in terms of the execution time. The idea behind this scheduling strategy is to compute the executable node first which has the most time-consuming path to a sink node.

Together with the PCT scheduler, the partitioning strategies Critical Path, MITE and DFS partitioning performed in particular well on the sample graphs extracted from TensorFlow. Critical Path partitioning aims to map the nodes of the critical path to the fastest feasible device. MITE makes use of a heuristic function to place the nodes and considers the factors memory, traffic and execution time. It also favors important nodes on fast devices. DFS partitioning traverses the graph in depth-first search and maps the nodes also based on a heuristic function. These three partitioning strategies

needed significantly less time to execute the graphs extracted from TensorFlow than the benchmark algorithm HEFT.

As the demand for fast machine learning libraries as well as the collection of graph structured data is constantly growing, it is essential to think about the computation and processing of these graphs. Obviously, distributed systems can have a huge impact on minimizing the total execution time by parallelizing. Efficient scheduling and partitioning plays a major role in executing graphs in an acceptable time. Therefore, we believe that it is definitely worth to do more research in scheduling and partitioning algorithms.

# Bibliography


[ABC+16] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, X. Zheng. "TensorFlow: A system for large-scale machine learning." In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 2016, pp. 265–283. URL: https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf (cit. on pp. 15, 16, 23).

[AC86] Arvind, D. E. Culler. "Annual Review of Computer Science Vol. 1, 1986." In: ed. by J. F. Traub, B. J. Grosz, B. W. Lampson, N. J. Nilsson. Palo Alto, CA, USA: Annual Reviews Inc., 1986. Chap. Dataflow Architectures, pp. 225–253. ISBN: 0-8243-3201-6. URL: http://dl.acm.org/citation.cfm?id=17814.17824 (cit. on p. 11).

[CJS+08] L.-C. Canon, E. Jeannot, R. Sakellariou, W. Zheng, S. Gorlatch, P. Fragopoulou, T. Priol. "Comparative Evaluation of the Robustness of DAG Scheduling Heuristics." In: *Integration Research in Grid Computing, CoreGRID integration workshop*. Ed. by S. Gorlatch, P. Fragopoulou, T. Priol. Also published as CoreGRID Technical Report TR-0120. Apr. 2008, pp. 63–74 (cit. on p. 62).

[CRB+15] K. Chronaki, A. Rico, R. M. Badia, E. Ayguadé, J. Labarta, M. Valero. "Criticality-Aware Dynamic Task Scheduling for Heterogeneous Architectures." In: *Proceedings of the 29th ACM on International Conference on Supercomputing*. ICS '15. Newport Beach, California, USA: ACM, 2015, pp. 329–338. ISBN: 978-1-4503-3559-1. DOI: 10.1145/2751205.2751235. URL: http://doi.acm.org/10.1145/2751205.2751235 (cit. on p. 62).

[CRC+16] K. Chronaki, A. Rico, M. Casas, M. Moreto, r. badia, E. Ayguade, j. labarta, M. Valero. "Task Scheduling Techniques for Asymmetric Multi-core Systems." In: *IEEE Transactions on Parallel and Distributed Systems* PP.99 (2016), pp. 1–1. ISSN: 1045-9219. DOI: 10.1109/TPDS.2016.2633347 (cit. on p. 62).

[CSCC15] R. Chen, J. Shi, Y. Chen, H. Chen. "PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs." In: *Proceedings of the Tenth European Conference on Computer Systems*. EuroSys '15. Bordeaux, France: ACM, 2015, 1:1–1:15. ISBN: 978-1-4503-3238-5. DOI: 10.1145/2741948.2741970. URL: http://doi.acm.org/10.1145/2741948.2741970 (cit. on p. 11).

[CW08] R. Collobert, J. Weston. "A Unified Architecture for Natural Language Processing: Deep Neural Networks with Multitask Learning." In: *Proceedings of the 25th International Conference on Machine Learning*. ICML '08. Helsinki, Finland: ACM, 2008, pp. 160–167. ISBN: 978-1-60558-205-4. DOI: 10.1145/1390156.1390177. URL: http://doi.acm.org/10.1145/1390156.1390177 (cit. on pp. 11, 45).

[CZG+16] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, E. P. Xing. "GeePS: Scalable Deep Learning on Distributed GPUs with a GPU-specialized Parameter Server." In: *Proceedings of the Eleventh European Conference on Computer Systems*. EuroSys '16. London, United Kingdom: ACM, 2016, 4:1–4:16. ISBN: 978-1-4503-4240-7. DOI: 10.1145/2901318.2901323. URL: http://doi.acm.org/10.1145/2901318.2901323 (cit. on pp. 11, 15, 21, 45).

[DCM+12] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, A. Y. Ng. "Large Scale Distributed Deep Networks." In: *Proceedings of the 25th International Conference on Neural Information Processing Systems*. NIPS'12. Lake Tahoe, Nevada: Curran Associates Inc., 2012, pp. 1223–1231. URL: http://dl.acm.org/citation.cfm?id=2999134.2999271 (cit. on p. 11).

[GLG+12] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, C. Guestrin. "PowerGraph: Distributed Graph-parallel Computation on Natural Graphs." In: *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*. OSDI'12. Hollywood, CA, USA: USENIX Association, 2012, pp. 17–30. ISBN: 978-1-931971-96-6. URL: http://dl.acm.org/citation.cfm?id=2387880.2387883 (cit. on p. 11).

[KAA+13] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, P. Kalnis. "Mizan: A System for Dynamic Load Balancing in Large-scale Graph Processing." In: *Proceedings of the 8th ACM European Conference on Computer Systems*. EuroSys '13. Prague, Czech Republic: ACM, 2013, pp. 169–182. ISBN: 978-1-4503-1994-2. DOI: 10.1145/2465351.2465369. URL: http://doi.acm.org/10.1145/2465351.2465369 (cit. on p. 63).

[KHP+09] R. Khandekar, K. Hildrum, S. Parekh, D. Rajan, J. Wolf, K.-L. Wu, H. Andrade, B. Gedik. "COLA: Optimizing Stream Processing Applications via Graph Partitioning." In: *Middleware 2009: ACM/IFIP/USENIX, 10th Inter-*

*national Middleware Conference, Urbana, IL, USA, November 30 – December 4, 2009. Proceedings*. Ed. by J. M. Bacon, B. F. Cooper. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 308–327. ISBN: 978-3-642-10445-9. DOI: 10.1007/978-3-642-10445-9_16. URL: http://dx.doi.org/10.1007/978-3-642-10445-9_16 (cit. on p. 62).

[KK98] G. Karypis, V. Kumar. "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs." In: *SIAM J. Sci. Comput.* 20.1 (Dec. 1998), pp. 359–392. ISSN: 1064-8275. DOI: 10.1137/S1064827595287997. URL: http://dx.doi.org/10.1137/S1064827595287997 (cit. on p. 11).

[KSH12] A. Krizhevsky, I. Sutskever, G. E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks." In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira, C. J. C. Burges, L. Bottou, K. Q. Weinberger. Curran Associates, Inc., 2012, pp. 1097–1105. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf (cit. on pp. 11, 45).

[LZCS14] M. Li, T. Zhang, Y. Chen, A. J. Smola. "Efficient Mini-batch Training for Stochastic Optimization." In: *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '14. New York, New York, USA: ACM, 2014, pp. 661–670. ISBN: 978-1-4503-2956-9. DOI: 10.1145/2623330.2623612. URL: http://doi.acm.org/10.1145/2623330.2623612 (cit. on p. 11).

[MAB+10] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, G. Czajkowski. "Pregel: A System for Large-scale Graph Processing." In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. SIGMOD '10. Indianapolis, Indiana, USA: ACM, 2010, pp. 135–146. ISBN: 978-1-4503-0032-2. DOI: 10.1145/1807167.1807184. URL: http://doi.acm.org/10.1145/1807167.1807184 (cit. on pp. 11, 63).

[MAP+15] Martın Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, Xiaoqiang Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: http://tensorflow.org/ (cit. on pp. 15, 21, 23, 25, 45, 61).

[MMTR16] R. Mayer, C. Mayer, M. A. Tariq, K. Rothermel. "GraphCEP: Real-time Data Analytics Using Parallel Complex Event and Graph Processing." In: *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*. DEBS '16. Irvine, California: ACM, 2016, pp. 309–316. ISBN: 978-1-4503-4021-2. DOI: 10.1145/2933267.2933509. URL: http://doi.acm.org/10.1145/2933267.2933509 (cit. on p. 11).

[MTLR16] C. Mayer, M. A. Tariq, C. Li, K. Rothermel. "GrapH: Heterogeneity-Aware Graph Computation with Adaptive Partitioning." In: *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. June 2016, pp. 118–128. DOI: 10.1109/ICDCS.2016.92 (cit. on p. 11).

[NSB+15] A. Nair, P. Srinivasan, S. Blackwell, C. Alcicek, R. Fearon, A. D. Maria, V. Panneershelvam, M. Suleyman, C. Beattie, S. Petersen, S. Legg, V. Mnih, K. Kavukcuoglu, D. Silver. "Massively Parallel Methods for Deep Reinforcement Learning." In: *CoRR* abs/1507.04296 (2015). URL: http://arxiv.org/abs/1507.04296 (cit. on pp. 11, 16).

[Sca16] S. Scarpinelli. *TensorFlow for Machine Intelligence*. S.l: Bleeding Edge Press, 2016. ISBN: 978-1-939902-35-1 (cit. on pp. 15, 21, 23, 45).

[THW02] H. Topcuouglu, S. Hariri, M.-y. Wu. "Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing." In: *IEEE Trans. Parallel Distrib. Syst.* 13.3 (Mar. 2002), pp. 260–274. ISSN: 1045-9219. DOI: 10.1109/71.993206. URL: http://dx.doi.org/10.1109/71.993206 (cit. on pp. 32, 33, 61, 63).

All links were last followed on Feburary 18, 2017.

**Declaration**

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature