Master thesis

Simulation of a Scheduling Algorithm for DAG-based Task Models

Maksym Planeta

19. May 2015

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

Betreuender Hochschullehrer: Prof. Dr. Hermann Härtig
Betreuende Mitarbeiter: Dr.-Ing. Matthias Lieber
Dr.-Ing. Michael Roitzsch
Dr.-Ing. Marcus Völp

TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Systemarchitektur, Professur für Betriebssysteme

Master's Thesis Topic

Dresden, 20th October 2014

Student's Name:	Maksym Planeta
Study Programme:	Informatik (Master)
Student ID (Matrikel):	3930325
Topic:	Simulation of a Scheduling Algorithm for DAG-based Task Models

Driven by the trend towards manycores, an increasing number of applications is developed with parallelism in mind. Asynchronous programming paradigms based on lambdas are used to simplify the expression of parallelism within an algorithm. The dependencies between those individual work items allow the modeling of an application's parallel nature with a directed acyclic graph (DAG).

State-of-the art scheduling algorithms however still model parallelism with opaque threads and therefore cannot benefit from the additional insight available from a DAG-based task description. This thesis should explore DAG-based task scheduling by developing a simulator that allows to experiment with different DAG representations of tasks and corresponding scheduling algorithms.

As a first step, a survey of existing parallel applications should be conducted to analyze their parallel behavior and to extract relevant execution time parameters. Examples from cloud, high performance computing, and real-time workloads can be considered. This analysis should then inform the synthetic simulations performed within the simulator.

The simulator should be capable of operating on single and multiple concurrently running DAG-style applications. It should assign work to simulated CPU cores and thereby generate an execution trace that allows to judge scheduler efficiency, for example by the resulting CPU utilization and makespan.

Having full knowledge of the DAG available with all execution time parameters is an ideal scenario for scheduling, but not realistic in practice. Nevertheless, such a clairvoyant scheduler is useful for benchmarking. At least one additional scenario with a reduced, more coarse-grained representation of task behavior and a consequently less precise scheduling algorithm should be implemented and evaluated.

Dynamically changing the core count, heterogeneous hardware, or unforeseen code execution in applications, for example to recover from a fault, are potential future extensions, but outside the scope of this thesis.

Responsible Professor:	Prof. Dr. Hermann Härtig
Assistant Advisors:	Dr.-Ing. Marcus Völp, Dr.-Ing. Michael Roitzsch
Start:	17th November 2014
Submission:	27th April 2015

Prof. Dr. Hermann Härtig

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Dresden, den date

Maksym Planeta

Abstract

Efficiency of a parallel applications merely depends on a scheduling algorithm which distributes sequential portions of application to multiprocessor systems. State of the art algorithms can be divided in following two classes. Just-in-time (JIT) algorithms are simple algorithms, which can schedule almost any application and have minimal requirements. Full-ahead algorithms are more computation intensive and require more information about a parallel program, but they are able to build more effective schedules.

Directed acyclic graph (DAG) is a typical instrument to model structure of a parallel application, which is used by full-ahead algorithms. DAGs show dependencies between sequential parts of an application and communication between them.

This thesis presents a Horizon algorithm class for scheduling DAG-based parallel applications. The Horizon algorithm extends a JIT algorithm class by providing some restricted knowledge about parallel program structure. With this new knowledge the Horizon algorithm manages to outperform a JIT algorithm. To the best of my knowledge this is the first attempt to fill the niche between full-ahead and JIT algorithms.

Contents

List of Figures

List of Tables

1 Introduction

Effective parallelization of an application requires extensive knowledge about program structure. There exist a variety of methods to declare program structure, that is convenient for the programmer, but also allows to effectively schedule the application a on a multiprocessor system. Examples of parallelization methods are the fork-join pattern [ABB00], message-passing systems [GLS99], map-reduce frameworks [DG08], future-based parallelism [Hal84], and asynchronous lambda approaches [App09].

These methods differ in their API, in their granularity and functionality. But all of them share one common goal: order sequential sections of an application in an order that respects their mutual dependencies. This process is called *scheduling*. Sequential sections represent parts of an actual execution trace of a program and have corresponding start and finish time. The same part of the application code can appear as several sequential sections.

Dependencies between sequential sections represent transfers of data that is calculated in precedent sections and used as an input by subsequent ones. Relations between sequential sections can be modeled as a graph, and thus all the dependencies go from past to future such that the graph has no cycles. Thus, such model is called Directed Acyclic Graph (DAG).

DAGs are a common and well-studied way to represent the execution of parallel application [ZS13; Blu+96]. It is known that scheduling of a DAG-structured program onto a multiprocessor computer is a NP-complete problem in its general form [Ull75]. There exist constrained models, which allow to determine optimal schedule in polynomial time [Hu61; CG72; PY79]. Simplicity of these models barely allows their utilization for scheduling of real parallel applications.

The most popular scheduling techniques do not attempt to find an optimal schedule [KA99b; WG90; ACD74]. Instead, they try to arrange the nodes of a DAG structured application in an order, which respect their sequential dependencies and correlates with an *importance* metric of a node. The importance metric is chosen so that scheduling of a more important job is more critical for the overall progress of execution.

Different scheduling algorithms are more or less successive in their attempt to order nodes of a DAG right. To build a shorter schedule an algorithm has to sacrifice speed of the algorithm itself [KA99a]. Different scheduling techniques allow to achieve other parameters of a schedule as high sustainability towards fluctuations in execution environment, fairness, etc.

On the other side of the spectrum are the algorithms which do not try to get as much information about program structure as possible. Instead, they try to utilize patterns which improve performance within a given execution environment [HL14; HL14]. This makes these algorithms fast and frees from burden of obtaining information about program structure.

The goal of this master thesis is to combine "structure oblivious" and "structure aware" algorithms. The main question behind my work is following. Can we achieve better performance by knowing only a part of program structure?

My initial motivation originates from Apple's GCD technology [App09], which provides a framework for asynchronous execution of so-called *blocks*. The blocks are parts of an application, which the programmer marks as suitable for asynchronous execution. The framework maintains a thread pool and decides itself whether it wants to run a block sequentially or in parallel.

At the implementation level the GCD framework maintains a list of blocks which are available for execution straight away. The GCD has no knowledge which blocks will appear next. I supposed that, if a GCD-like framework has more complete knowledge about program structure, its decision making may become more reasonable and effective.

In thesis thesis shows a variant of a model which supplies a scheduler with restricted knowledge about program structure. I propose an algorithm which acts within this model. I compare the algorithm mostly against the algorithms which have complete knowledge about program structure because this class of algorithms is more diverse and does not depend on the underlying system model that much.

The master thesis has following structure. Section 1 introduces reader to the master thesis topic and justifies its motivation. Section 2 provides the reader with the background knowledge and describes the so-called *Horizon* model, which is the subject of this thesis. Section 3 gives a description of several state of the art algorithms. This section also includes a description of the Horizon algorithm. The algorithm represents only one particular implementation, which is compared with other algorithms. In principle, other variants of the Horizon algorithm are also possible, but not included in the scope of this thesis. Section 4 describes how algorithms were evaluated and explains the results. Section 5 makes concluding remarks, outlines discussion points and gives insight to future work.

2 Technical background

Properties of the environment where the scheduler acts have a major influence on the scheduling algorithm structure. Depending on whether or not the processors are homogeneous, what is the network topology, which information about the program is given, etc. the algorithm can make a decision considering very different pieces of information.

The main aspect, which I want to study in this thesis, is how the properties of the resulting schedule depend on the completeness of the information available to the algorithm. Considering this goal, state of the art algorithms can be divided into following two classes.

Algorithms from the first class have very few information about the program they are scheduling. Their decisions are made in runtime. And minimization of the scheduling overhead often is an important objective for them. This class is called *just-in-time algorithms* (JIT-algorithms) and described in Section 2.3.

Representatives of another class are aware of complex structure of the application. They make each scheduling decision considering possible impact of it in the future. This all allows to make all complicated and time-consuming actions before the execution of the application starts, for example in compile-time. In this case runtime part of the scheduler can be very simple, making the problem of significant scheduling overhead less relevant. This class of the algorithms is called *full ahead algorithms*.

Environment where an algorithm acts is called *system model*. System model consists of three components: *computer model*, *network model* and *scheduling model*. Computer model describes properties of the processors, which run parallel application and the scheduler itself. This thesis differentiates processors as homogeneous and heterogeneous.

Processor are connected via communication links. Properties of the communication links and their topology is described by the network model. As with processors, communication links can be either homogeneous or heterogeneous. If both communication links and processors are homogeneous, the whole system model is considered *homogeneous*, or heterogeneous otherwise.

The network models describes communication links. The link can be either contention free or contention non-free. Contention free means that once a communication operation starts, it does not experience any disturbance from another communication operations. The network model of this master thesis uses contention non-free communication links. This means that a communication operation exclusively occupies the channel between the two processors and can't be interrupted.

Most of the algorithms presented in this thesis are oriented towards the same *homogeneous system model*. I consider this model as a basic one and describe it in Section 2.1. Deviations from this model allow to build algorithms which can show better quality in real life applications, possibly within some constrained application area.

This master thesis contains a description of following three scheduling models: full-ahead, just-in-time and Horizon. The scheduling model describes the information which is available to a scheduler and the way how a scheduler can operate with available information.

The Horizon model is the contribution of this thesis. It is a combination of full-ahead and just-in-time models. The scheduler within this model discovers new information about program structure during the execution. In this sense it is similar to just-in-time algorithms, but in contrast to JIT-algorithms, more information is available at each step and more planning can be made.

Heterogeneity is one of the popular modifications to the basic model. One of the representatives of this class of algorithms is the Lookahead [BSM10] algorithm. I picked this algorithm, because it is the only algorithm found in the literature which utilizes an approach similar to the idea which I present in this thesis. This algorithm uses the model with heterogeneous processors, which I describe in Section 2.2.

Just-in-time algorithms have execution environment compatible to the one of the model from Section 2.1, but they assume different program structure model. The differences are described in Section 2.3. Within the scope of this thesis I present the Horizon algorithm, which scheduling model is something between the model of just-in-time algorithms and full-ahead algorithm. This model is described in Section 2.4.

2.1 Homogeneous model

Parallel program execution consists of series of sequential parts, which have data dependencies between each other. Such structure can be modelled as directed acyclic graph (DAG), where nodes represent sequential parts and edges represent data dependencies. Various parameters of the program can be modelled either as attributes of nodes, edges or the graph itself.

To start with model description I define following graph attributes. Formally, a directed acyclic graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ consists of set of nodes $\mathcal{V}$ and set of edges $\mathcal{E}$. Nodes $v_i \in \mathcal{V}$ represent sequential parts of the program, which are called jobs[1]. Edges $e_{i,j} \in \mathcal{E}$ represent data dependencies which result in communication operations. Symbol v represents the size of set $\mathcal{V}$. Symbol e represents the size of set $\mathcal{E}$. The execution environment consists of set $\mathcal{P}$ of interconnected processor elements, where $p_i \in \mathcal{P}$ are elements of the set. Symbol p represents the number of processors in set $\mathcal{P}$. Figure 2.1 shows an example of a DAG which models a parallel program. For the sake of brevity some of the edge labels are missed in the figure. DAG attributes are summarized in Table 2.1.

Dependencies of the job are designated by an operators $pred(v_i)$ and $succ(v_i)$. Operator $pred(v_i)$ returns a list of the jobs which put data dependency on job v_i. Every node except entry node has at least one parent. Operator $succ(v_i)$ return a list of the jobs which execution depends on the node v_i. When a node is scheduled operator $p(v_i)$ returns a processor which a node was scheduled to.

[1] Terms *nodes* and *jobs* are used interchangeably.

Attribute	Definition	Meaning
$\mathcal{G}$	$(\mathcal{V}, \mathcal{E})$	DAG of the program
$\mathcal{E}$	$e_{i,j} \in \mathcal{E}$	Set of data dependencies
e	$\|\mathcal{E}\|$	Number of edges
$\mathcal{V}$	$v_i \in \mathcal{V}$	Set of jobs
v	$\|\mathcal{V}\|$	Number of jobs
$\mathcal{P}$	$p_i \in \mathcal{P}$	Set of processors
p	$\|\mathcal{P}\|$	Number of processors

Table 2.1: DAG attributes

Operator $w(x)$ designates communication and computation costs. Computation costs of job v_i are $w(v_i)$. Communication costs of data transfer from job v_i to job v_j are $w(e_{i,j})$. If jobs v_i and v_j are scheduled to the same processor, $w(e_{i,j}) = 0$.

Without loss of generality, assume that a DAG has only one entry node and only one exit node. All the nodes, except entry and exit nodes, and edges, except edges incident to entry or exit nodes, have positive weights. Entry and exit nodes, as well as edges incident to entry and exit nodes are allowed to have zero weights. Once a job gets started it runs until the end, i. e. jobs are not preemptable.

Following DAG attributes are important.

s-level (stands for *static level*) of a node v_i is the length of the longest path from node v_i to the exit node, including the weight of v_i. S-level does not consider communication costs, thus it does not depend on particular schedule. In Figure 2.1 nodes encompassed in the s-level of the node v_6 are circled in light gray color. A recurrent relation for s-level is following:

$$sl(v_i) = \begin{cases} \max\limits_{v_j \in succ(v_i)} (sl(v_j)) + w(v_i), & \text{if } succ(v_i) \neq \emptyset \\ w(v_i), & \text{if } succ(v_i) = \emptyset \end{cases}$$

Static t-level (stands for *static top level*) of a node v_i is the length of the longest path from the entry node to a node v_i, excluding the weight of v_i. Static t-level includes all communication costs and computed before schedule is created. Thus st-level computation does not consider possibility to nullify communication costs, when dependant jobs run on the same processor. In Figure 2.1 the nodes encompassed in t-level of the node v_6 are circled in dark gray color. A recurrent relation for t-level is following:

$$stl(v_i) = \begin{cases} \max\limits_{v_j \in pred(v_i)} (stl(v_j) + w(e_{v_j,v_i}) + w(v_j)), & \text{if } pred(v_i) \neq \emptyset \\ 0, & \text{if } pred(v_i) = \emptyset \end{cases}$$

Static b-level (stands for *static bottom level*) of a node v_i is the length of the longest path from node v_i to the exit node, including the weight of v_i. Computation of

static b-level includes communication costs in the same way as with t-level. In Figure 2.1 the nodes encompassed in b-level of the node v_6 are circled in light gray color. A recurrent relation for b-level is following:

$$sbl(v_i) = \begin{cases} \max\limits_{v_j \in succ(v_i)} (sbl(v_j) + w(e_{v_j,v_i})) + w(v_i), & \text{if } succ(v_i) \neq \emptyset \\ w(v_i), & \text{if } succ(v_i) = \emptyset \end{cases}$$

Total work ($\mathcal{W}$) is total complexity of all jobs in the DAG. If all nodes are scheduled to the same processor, the length of the resulting schedule is equal to total work. The formula to calculate total work is following:

$$\mathcal{W} = \sum_{v_i \in \mathcal{G}} (w(v_i))$$

Critical path (CP) designates the length of the longest path from the entry node to the exit node. A DAG can have several critical paths of equal length. Critical path length equals to s-level of an entry node. Critical path is an important parameter of a DAG, because it shows the lower limit of any possible schedule length: even with unbounded amount of processors execution can't take less than execution of critical path requires. Critical path also can include communication costs existing between the nodes in critical path, but in this case, critical path length is not minimal possible schedule length. In Figure 2.1 critical path is shown with thick arrows.

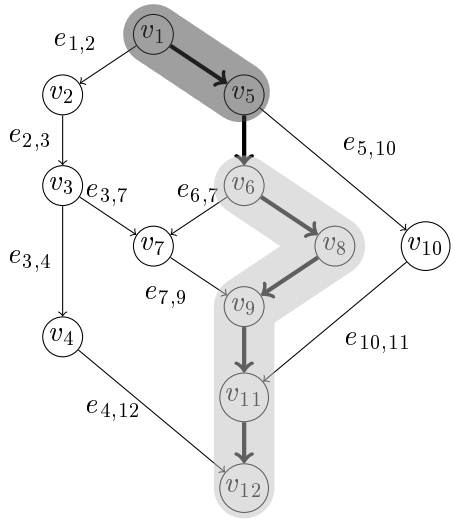


Figure 2.1: DAG structure

Parameters mentioned above do not depend on particular schedule which can assign jobs to processors in different way. These parameters called *static*. If a DAG parameter depends on concrete a job-to-processor assignment, it is called *dynamic*. Dynamic

parameter can also be considered as a parameter of particular schedule. Example of values for DAG parameters for a DAG in Figure 2.1 is given in Table 2.2a. Dynamic parameters are defined in next subsection.

i	$w(v_i)$	$sl(v_i)$	$sbl(v_i)$	$stl(v_i)$	$ALAP(v_i)$
1*	10	130	270	0	0
2	20	110	200	40	50
3	20	90	170	70	80
4	40	60	70	100	180
5*	10	120	250	20	20
6*	30	110	220	50	50
7	20	70	140	100	110
8*	30	80	170	100	100
9*	10	50	110	160	160
10	30	70	120	40	130
11*	20	40	60	210	210
12*	20	20	20	250	250
	CP_{static}				130
	$CP_{dynamic}$				270
	$\mathcal{W}$				280

(a) Node parameters. Nodes with asterisk are on the critical path.

	$w(e_{i,j})$
$e_{1,2}$	30
$e_{1,5}$	10
$e_{2,3}$	10
$e_{3,4}$	10
$e_{3,7}$	10
$e_{4,12}$	10
$e_{5,6}$	20
$e_{5,10}$	10
$e_{6,7}$	20
$e_{6,8}$	20
$e_{7,9}$	10
$e_{8,9}$	30
$e_{9,11}$	40
$e_{10,11}$	30
$e_{11,12}$	20

(b) Edge parameters

Table 2.2: DAG parameters

2.1.1 Schedule parameters

A schedule is a mapping of jobs of a parallel application to timeslots of processors, which respects system model constraints. A scheduling algorithm performs assignment of the jobs to timeslots of processors. An example of a schedule of the DAG from Figure 2.1 with parameters from Table 2.2a is presented in Figure 2.2.

Start time of a node within particular schedule is designated as $ST(v_i)$.

Finish time of a node within particular schedule is designated as $FT(v_i)$. Between start time and finish time holds following relation:

$$FT(v_i) = ST(v_i) + w(v_i)$$

Makespan denotes the finish time of the exit node. Also called *schedule length*.

Dynamic t-level (stands for *dynamic top level*) of a node v_i on a processor P_i is the length of the longest path from the entry node to a node v_i, excluding the weight of v_i. Dynamic t-level designates the earliest possible start time of a node v_i, when

parent nodes of the node v_i are already scheduled.

$$tl(v_i) = \begin{cases} \max\limits_{v_j \in pred(v_i)} (FT(v_i) + w(e_{v_i,v_j})), & \text{if } pred(v_i) \neq \emptyset \\ 0, & \text{if } pred(v_i) = \emptyset \end{cases}$$

Dynamic t-level is computed within a context of particular schedule, meaning that if $p(v_i) = p(v_j)$, then $w(e_{v_i,v_j}) = 0$. Dynamic t-level does not consider the availability of a processor ready for execution of the node v_i, hence actual earliest possible time can be bigger.

Dynamic b-level (stands for *dynamic bottom level*) of a node v_i on a processor P_i is the length of the longest path from node v_i to the exit node, including the weight of v_i. Dynamic b-level is computed as follows.

$$bl(v_i) = \begin{cases} \max\limits_{v_j \in succ(v_i)} (bl(v_j) + w(e_{v_j,v_j})) + w(v_i), & \text{if } succ(v_i) \neq \emptyset \\ w(v_i), & \text{if } succ(v_i) = \emptyset \end{cases}$$

Critical path also can be considered as a dynamic attribute. In this case it is the longest path in a schedule from entry to exit node and equals to dynamic b-level of an entry node. As with dynamic b-level, critical path length can change depending of mutual placement of adjacent nodes comprising the critical path. Even nodes encompassed in the critical path can change for different schedules. If not mentioned otherwise, dynamic critical path is assumed.

Activity $(f(\tau_i, t))$ of a node v_i indicates whether a job is being executed at time t.

$$f(\tau_i, t) = \begin{cases} 1, & t \in [\tau_i - w(v_i), \tau_i] \\ 0, & \text{otherwise} \end{cases}$$

Ready time $(R_i(v_j))$ of a processor p_i means earliest possible time when a processor can run a job of size $w(v_j)$. Ready time takes into account time required to satisfy job dependencies. For example, ready time $R_2(v_2)$ in the schedule in Figure 2.2 equals 40, assuming only node v_1 already scheduled.

Earliest Start Time $(EST(v_i, p_j))$ refers to earliest possible execution time of a node v_i on a processor p_j, with respect to both dynamic t-level of node v_i and ready time of processor p_j.

$$EST(v_i, p_j) = \max(tl(v_i), R_j(v_i))$$

Earliest Finish Time $(EFT(v_i, p_j))$ refers to earliest possible finish time of a node v_i on a processor p_j. Between EST and EFT following relation holds.

$$EFT(v_i, p_j) = EST(v_i, p_j) + w(v_i)$$

Actual Finish Time (AFT) is time when node v_i completes its job within particular schedule. $AFT(v_i)$ represents actual finish time of node v_i. Between parents' AFT and children EFT holds following relation:

$$EFT(v_i, p_j) = \max\left\{R_j(v_i), \max_{v_k \in pred(v_i)}(AFT(v_k) + w(e(v_i, v_k))\right\}$$

As-late-as-possible ($ALAP$) is a metric which indicates how much the node's start time can be delayed without increasing of total schedule length. $ALAP$ of the critical path nodes equals to their t-level. $ALAP$ is computed according to following formula:

$$ALAP(v_i) = \begin{cases} \min_{v_j \in succ(v_i)}(ALAP(v_j) - w(e_{v_i,v_j})) - w(v_i), & \text{if } succ(v_i) \neq \emptyset \\ CP - w(v_i), & \text{if } succ(v_i) = \emptyset \end{cases}$$

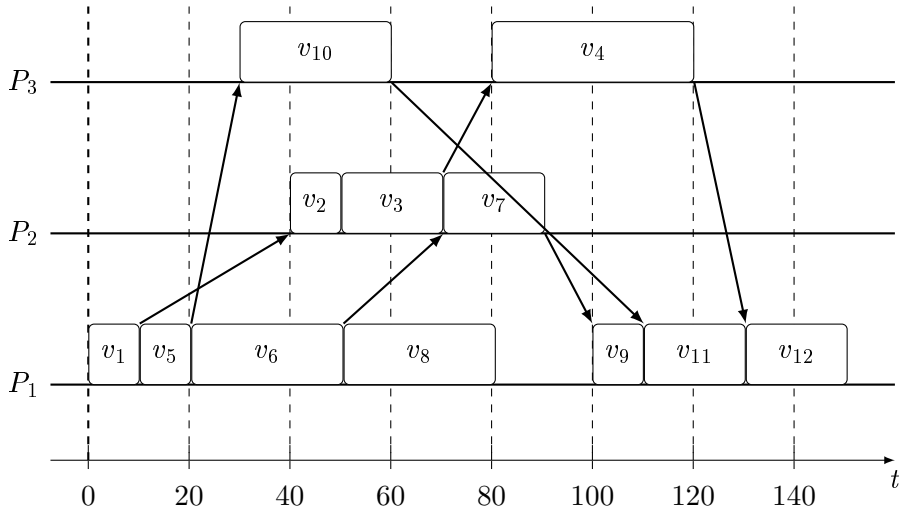


Figure 2.2: Schedule example

A schedule is *optimal* if it has the best possible evaluation parameters among other schedules within given number of processors. Although various *objective* parameters are possible, typically *makespan* is used. This includes my work too. Since optimality is not practical comparison criteria for various algorithms, I compare their *performance*, meaning how good their objective parameters are. In my case, the algorithm having better performance, is the one producing schedule with smaller makespan.

2.2 Heterogeneous model

The heterogeneous system model is an extension of the homogeneous system model. There are several variations of this model possible, but the main differences are the following. Computation costs is not a single number $w(v_i)$ anymore, but a vector $w(v_i)$ of size p. Element j of the vector $w(v_i)$ represents computation costs of a node v_i on a processor p_j. Operator $w(v_i, j)$ represents computation costs of a node v_i on a processor p_j.

Withing this thesis I assume a communication model that consists of a network of fully connected processors. Communication costs are built up of two network parameters:

1. Matrix B of size $p \times p$ of data transfer rates. Element $b_{i,j}$ from matrix B shows data transfer rate from node v_i to node v_j;
2. Vector L of size p of latency created by processors. Element l_i from matrix L designates the latency of processor p_i in communication operation.

The weight of edges in the DAG, instead of time required for data transmit, now indicate size of data to be transmitted. Combining network parameters and edge weight, communication cost of transmitting message from node v_i to node v_j is

$$c_{i,j} = L_{p(v_i)} + \frac{data_{i,j}}{b_{p(v_i),p(v_j)}}$$

It is to be noticed that within this model there exist heterogeneity in two types of resources: processors and network. Variations of system model may have changes in network model and processor model independently. Besides communication costs, the network model may have another topology model. If network topology may have an arbitrary structure, APN scheduling algorithms should be applied [KA99b].

2.3 Just-in-time scheduling model

Just-in-time (JIT) model is oblivious regarding program structure. Schedulers within this model can't afford any planning. It is assumed that at each moment in time a scheduler knows some very restricted information, but this information is up-to-date. Because of this, scheduling decisions are to be made immediately before they are applied, whereas full-ahead scheduler can decide to schedule a job to a processor far before the job is actually scheduled.

Just-in-time model has an advantage by being simpler. Thus it is widespread in real systems. Parallel programming model which is based on threads and mutexes can be seen as such model. Mutex locks can be seen as synchronization points, which are creation of the jobs in full-ahead model. The scheduler knows only about the threads and the mutexes which block some threads, but the scheduler doesn't know which mutex a thread is going to acquire next. Scheduling is done on the fly or *just-in-time* by picking a thread which is not blocked on any mutex.

Similarly to full-ahead model, within JIT model program is assumed to consist of sequential pieces called *jobs*. Jobs have sequential dependencies between each other. If two jobs have noncontradictory dependency sets, they may be run in parallel on different processors. A parallel program can be modeled as a DAG where *nodes* represent jobs and edges represent *edges*. Because nodes in a DAG correspond to jobs in a parallel application, the terms nodes and jobs are used interchangeably.

Within the just-in-time model, scheduling algorithms DAG nodes appear for the scheduler only when they become *ready*, i. e. all their dependencies get satisfied. A job that is currently being executed on a processor is called *active*. If execution of a job A depends on another job B, job A is called a *child* of a job B. And job B is called a *parent* of a job A.

The trivial just-in-time model assumes that there exists single globally accessible list of jobs, which state is *ready*. Such list is called *ready queue* [2]. Whenever there is a processor with no active job, it tries to grab one from a ready queue.

Globally accessible queue puts tough requirements on network latency. This makes model simple, but unrealistic. Global just-in-time model often struggles to achieve high overall performance in large scale systems. Anderson, Lazowska, and Levy [ALL89] have shown that contention for the system bus can drastically decreases both system latency and throughput. Various researches have shown the influence of the data locality on the number of CPU cache misses [Spo+09; HL14; SL93] and page faults [Blu+96]. This increases shared memory bus traffic and contention and bring significant performance penalty. [SL93] has shown that if child jobs tend to stay on the same CPU as parents, performance penalty grows slower with increased number of processors.

2.4 Horizon scheduling model

This section presents a model which was not met previously in the research according to my best knowledge. It combines the ideas both from just-in-time model and full-ahead model. In detail the model is following.

Parallel program is modelled as a DAG, which consists of jobs which have data dependencies between each other. These dependencies determine possibility of parallel execution of the jobs. Scheduling is done by mapping the jobs to available processors in an order that respects data dependencies. This part of the model repeats full-ahead model.

Scheduling of the program is possible only in runtime, because in the beginning only partial information about program execution is available. When an application starts, the scheduler discovers certain number of jobs, within certain depth from the root node. Some of these jobs are ready for execution right after root node finishes, but others depend on root's children descendants. After a job finishes scheduler discovers its descendants within certain depth.

In this aspect horizon model is similar to JIT-model. But in contrast to the later, scheduler within the horizon model discovers more information after each step of execution. This leads to the main hypothesis of my master thesis.

[2] Also known as *ready list*.

- More information about parallel program structure enables the scheduler for making better scheduling decision at each step of the execution. As result, overall schedule will be better.

To my best knowledge there is no know programming instrument that uses this model. Development of such instrument is out of scope of this work. As possible way to provide scheduler with such knowledge I consider static code analysis tools combined with runtime instruments.

Static code analysis tools should be able to recognize sequential parts of the program and dependencies between them. Programmer can advise these tools, by writing a program using special technologies, which have notion of tasks and dependencies. Examples of such technologies are Petri Nets, UML, markup languages [YB05]. Additionally, static code analysis tool should be able to annotate binary image of the program with the information about such sequential parts. These annotations should be recognized by dynamic part of a parallel systems, which provides a scheduler with horizon knowledge.

Dynamic runtime instruments should consider sequential parts of the program as first class citizens and be good in keeping and understanding dependencies between them. Scheduler should be able to fetch this information and use it for making scheduling decisions.

State of the art parallel programming systems, like StarPU [Aug+11], X10 [Cha+05], ZeroMQ [Hin15] understand some of aforementioned concepts, providing higher or lower level of abstraction.

Another possible application of a Horizon model can be introduced by reducing the full-ahead model. Within a full-ahead model all information required by the Horizon model is available. The difference is that in the Horizon model program structure is discovered step by step. This constrains the scheduler within Horizon model to work only during runtime, without a possibility to generate a schedule before program execution.

Various situations are possible, where the Horizon model can be superior to full-ahead one. First, consider a situation when there exist a bunch of n parallel programs, which are to be scheduled on a parallel system. At the same time there can be run up to $m \leq n$ programs simultaneously. Creating full-ahead schedules for simultaneous execution for all possible combinations of application can be practically impossible for sufficiently big numbers of n and m. At the same time JIT-schedulers are believed to be less efficient than full-ahead ones [YB05]. Splitting processor into a set of independent partitions can put tight limit on the best possible utilization bound either. I believe, in this situation, the Horizon scheduler, if it shows better performance than JIT scheduler, can be considered as a preferable choice. It does not require to split the processors and does not require to generate huge number of schedules beforehand.

Another example, where I see the Horizon model being applicable, is improving ability of the schedule to tolerate uncertainties in cost estimations. This property is called *robustness*. After a full-ahead schedule is created, it is not changed anymore. But in runtime it can turn out that certain estimations are imprecise. Jobs which are closer to the exit node tend to be allocated to processors basing on less and less correct information. This can have negative impact on overall performance. Horizon scheduler naturally takes into account changes in cost estimation of the finished jobs. This enables the scheduler

to assign the jobs basing on more precise information. This aspect is to be studied in Section 4.

As a continuation of previous example, processor failures can be seen as another source of unpredictable delays. Sometimes failures can require significant changes in the schedule and here again the Horizon model looks more durable than full-ahead one. Investigation of this aspect is not considered in the scope of this thesis and left for future work.

2.5 Scheduling models

The system model determines variety of algorithms which can be used. The applicability of the model is determined by required properties of the execution environment. Such factors as performance guarantees, robustness, reliability and others determine optimal system model and optimal scheduling algorithm. This section introduces general scheduling principles and paradigms within aforementioned system models.

2.5.1 Work stealing algorithms

Just-in-time scheduling model is popular in state of the art scheduling systems like StarPU [Aug+11] or X10 [Cha+05]. There exist many algorithms within this model. HEFT [THW02], Min-min [HF99], Max-min [HF99], Sufferage [HF99] are among the most popular. These algorithms are mostly opted to run in heterogeneous environments. But since this thesis focuses mainly on homogeneous systems I will give more detailed description for a scheduling paradigm, which basic version works in homogeneous environment. This particular algorithm is called *work stealing*.

There exist two major dynamic scheduling paradigms where jobs tend to stay where parents have been run: *work stealing* and *work sharing* [BL99]. To enforce this principle each processor maintains its own *local ready list*. In work stealing paradigm, processors take or *steal* jobs from ready lists of other processors. And in work sharing ones, processors pass or *share* jobs from their ready list to the ready lists of other processors. If a processor attempts to steal a job only when its own queue becomes empty it is called *parsimonious* [Spo+09].

Principles of work sharing algorithms are similar to work stealing ones, but the latter ensure less communication overhead [BL99].

Work stealing algorithms are subject of extensive research [Spo+09; BL99; ABB00; ABP98], but also have a number of practical implementations [Hal84; Blu+95]. Execution environment where typical work stealing algorithm acts has following structure. There exist a set of processors that can compute parallel tasks independently. Each processor has its own ready list. When a processor finishes the job, some of the children of this job can become ready. If it happens, then these children are added to the local *ready queue* of the processor.

Each processor is capable to communicate with any other processor. This communication could involve data transfers required to complete the job, but also processors are capable to communicate to exchange the jobs in their working queues. The process of

exchanging jobs among ready queues is the essence of scheduling for dynamic scheduling algorithms.

2.5.2 Full-ahead scheduling

If the program structure is known beforehand, it is possible to develop more complicated algorithms. A number of examples are known in the research [WG90; BSM10; Wu00; ACD74; KA99b; ZS13; THW02]. All these algorithms differ not only in the proposed approach, but as also in the details of the models where the algorithms operate. An example of algorithm classification is given in [KA99b]. In Section 3 I describe several particular algorithms and specifics of their models.

Besides structure of the DAG itself, important characteristic of the system model of the algorithm is the information that is known about the jobs and job communication. In simplest case, we can assume that anything, except job precedence constraints is irrelevant. In this situation jobs are assumed to have *unit* computation costs (i. e. all computation costs are equal for all the jobs) [Hu61; ACD74]. If computation costs are under consideration, node weight can have arbitrary value. And this value represents time required to complete a job on a processor. If communication is irrelevant edge weight is either uniform for all the edges or zero. UET-UCT (unit estimated time-unit communication time) is a typical model in research area [Fin+96; AK00], which assumes both computation and communication costs have unit weight.

The popular model which operates with arbitrary communication and computation costs is called *macro dataflow* model [YG92; WG90]. Without explicitly naming it, this model can also be found in numerous other papers [ACD74; KA99b; SZ04b]. Macro dafaflow model works as follows. The execution runs respecting the dependencies between the jobs in a DAG. Each job runs exclusively on a certain processor for the time, which depends on the computation costs of the job. These costs can have arbitrary finite value. Before a job starts, it should receive data from its parent. Time required to accomplish this operation depends on the communication costs of the job and also can have arbitrary finite value. When both parent and child run on the same processor, communication between them takes no time.

Macro dataflow model, yet simple, is often precise enough to approximate execution of parallel program on a multiprocessor system running it. The assumption of zero communication costs within the same processor is realistic because throughput of the network is much lower than the throughput of local memory. Moreover, it is often the case, that data should not be moved, if a job consuming the data resides on the same processor as the job generated the data.

Modern systems are often heterogeneous. This requires algorithms to cope with heterogeneous systems as well. Computing systems can have to kinds of heterogeneity, which can also be combined in the same system: heterogeneity of processors [Gra99; THW02; AB14] and heterogeneity of communication [AB14; BSM10]. Heterogeneity of processors means that instead of single node weight, each job has defined table of execution times for each processor that exist in the system. Different communication channels between processors imply different time requirements for a data transfers, which happen to fulfill job's dependency requirements.

System topology also can bring significant complication for an algorithm. Being fully connected graph (*clique*) in the simplest case, connections between processors can form arbitrary structures. Clique, Hypercube, Fat Tree are popular topologies. But sometimes combinations of famous structures or irregular structures should be handled. The reasons for such a diversity are expenses, throughput, latency and reliability, which vary for different topologies. Irregular structures may occur when there is no single authority which builds up network infrastructure. Internet or SETI@home project are good examples. These networks join together many different subnetworks with various topologies and form arbitrary structures in result.

Depending on existence of the restriction of the DAG, algorithms are divided in *arbitrary graph structure* algorithms and *restricted graph structure* algorithms. Hu [Hu61] requires the program to have a tree-structure and the jobs to have unit computation costs. Coffman and Graham [CG72] allows jobs to have arbitrary computation costs, but number of processors is restricted to two. Finta et al. [Fin+96] proposes an algorithm for arbitrary structure DAG within UET-UCT model, but only for two processors. Papadimitriou and Yannakakis [PY79] proposed to schedule an interval-ordered task graph with uniform jobs computation costs to an arbitrary number of processors. Malewicz [Mal05] propose an algorithm that permits DAG to have complex structure, but requires it to be *narrow* (i. e. the width of the DAG is at most constant). These algorithms represent algorithms with restricted graph structure, but allow to create an optimal schedule in polynomial time.

Another group of restricted graph structure algorithms does not allow to build an optimal schedule in polynomial time, but has softer DAG constraints. Computations where any two jobs which have common parent also have a common child are called *fully strict* or *well-structured* [BL99]. This kind of structure is also called fork-join parallelism, because it can be guaranteed by fork-join paradigm of an operating system. The guarantee is achieved, because the parent thread is the one who always joins with the child thread. These kinds of algorithms put boundaries on the worst case execution time[ABB00; ABP98; BGM99], additional number of cache misses [Spo+09; HL14; ABB00; SL93], additional number of page faults [Blu+96], and memory space requirements [Blu+96; BGM99].

Getting program structure can become a cumbersome task [Wil+08]. Typical way to gather such information is either static analysis [CS; Fer+01; Gus+03] or program execution monitoring [DGH91; PN98]. Getting precise information about future program execution is exaggerated, because of unpredictable and nondeterministic situations that can occur in run-time [Can+08; Ton+00; Mal05; FBB08; MS98]. Ability to sustain unpredictable situations called *robustness* [Can+08]. Various researches define robustness in different manner, comparison of different metrics representing robustness presented in [CJ07].

There are various approaches to tolerate unpredictability of the system. It is possible to make computation and communication time overestimation to improve robustness of the schedule [Can+08], but the disadvantage is a bigger slacking time, and thus increasing scheduling overhead. Just-in-time algorithms, which make decisions on the fly, basing on the information that is available in current moment of time, are naturally tolerant to unexpected jitters in task execution times. A downside is that dynamic algorithms

usually have worse performance in comparison to DAG aware algorithms, in situations when execution does not suffer from uncertainties.

Dynamic task rescheduling [SZ04b; MS98] is a hybrid approach that combines scheduling based on *a priori* knowledge of the DAG and scheduling based on current information. Such algorithms tend show better performance in presence of uncertainties, if they are based on better DAG aware algorithm [Can+08].

If nature of unpredictability is known, for instance deviations of expected job execution time and expected execution times are defined, *stochastic scheduling* algorithms [ZS13; Tan+11] come into play. These schedulers are similar to convenient list schedulers, but assign job priorities with respect to the level of uncertainty of execution time of each job. In the end they prepare a priority list of the jobs, which is expected to be more robust, than the result of typical scheduling algorithm.

2.5.3 Horizon scheduling

This scheduling model is a derivative of the Horizon system model, which was described in Section 2.4. The Horizon scheduler can be seen as a generalization of just-in-time scheduler, because the horizon can be reduced to contain only ready nodes.

Figure 2.3 shows which data horizon scheduler operates on. The execution state represents a situation when nodes v_1, v_2 and v_5 are already finished. The ready queue contains nodes v_3, v_6 and v_{10}. In the example horizon *depth* assumed to be 2, thus additionally to ready nodes, the horizon contains also nodes v_4, v_6, v_7, v_8, v_{10}, v_{11}. Depth is a parameter which determines how far from the ready nodes the graph structure is visible to the scheduler. Depth 1 means that only ready nodes are visible. If depth is 2, then additionally to ready nodes, children of ready nodes comprise the horizon. If depth is 3, grand children of ready nodes get into the horizon, and so on.

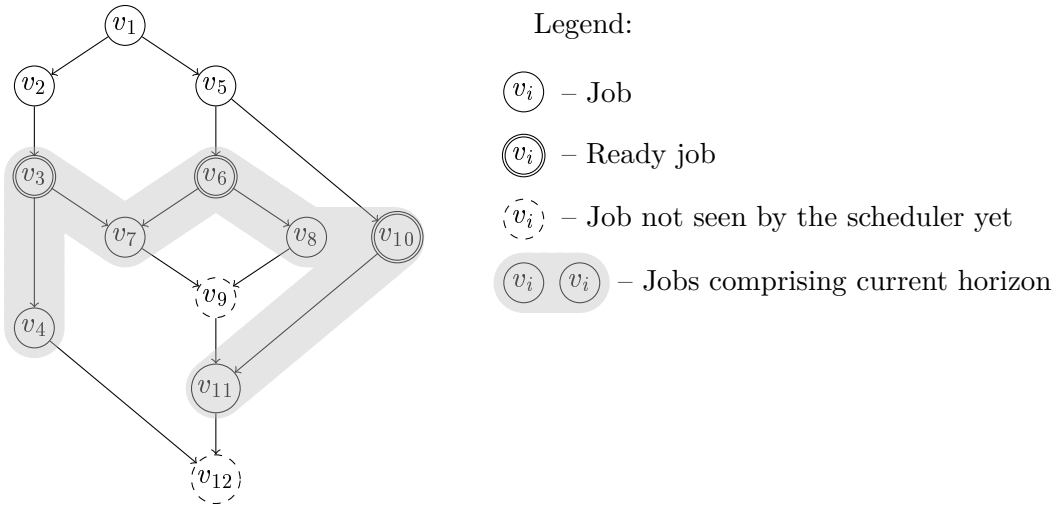


Figure 2.3: DAG as it seen by Horizon scheduler.

For each node in the horizon computation cost estimations are known. Also communication costs between any two dependant nodes are known if both of them are in

horizon. If a node from the horizon depends from a node that is not in horizon only existence of such dependency is known. An example of such dependency is the edge $e_{9,11}$ in Figure 2.3.

The Horizon scheduling model allows to schedule any task which dependencies either are satisfied or can be satisfied by scheduling only jobs from the horizon. This constraint is required, because otherwise, if it would be possible to schedule a job which has any uncovered dependency (like v_{11} in the example), it could be necessary to change the assignment of jobs to processors. This process is called rescheduling and for simplicity it is not allowed.

Considering the example from Figure 2.3, nodes v_3, v_4, v_6, v_7, v_8, and v_{10} are allowed to be scheduled at this point. Node v_{11} will be allowed for scheduling only after node v_9 is discovered. Node v_9 will be discovered first when either node v_7 or v_8 becomes ready. Node v_7 will become ready after either node v_3 or v_6 finishes and another appears in the horizon. Node v_8 will become ready only after node v_6 finishes.

Horizon scheduling allows to assign jobs to processors in advance. This allows to perform some preliminary steps, which are required to run a job. One such important step is running communication operations. In contrast to the JIT model it is possible to start communication earlier, just after parent nodes finish. This property is taken from full-ahead model and is assumed as in important detail, which improves overall schedule performance. Detailed investigation of this aspect is provided in Section 4.

2.6 List scheduling

After discussion about differences in system models and scheduling models, this section describes implementation details which are often common for all these models. As it was mentioned in Section 1, the scheduling problem is an NP-complete problem. Hence, the algorithms, which are described within the scope of this master thesis, are heuristics. They do not give an optimal solution in a strict sense, but some suboptimal one. The typical scheduling algorithm heuristic is called *list scheduling* [ACD74; PY79; Sch96; KA99b; FL12; AB14].

The goal of list scheduling algorithms is to minimize (or maximize) certain parameter of a resulting schedule: *makespan* [AB14], *fairness* [Zah+10], energy efficiency [Zon+13], etc. The idea behind list scheduling heuristic is that it build a *priority list* of all the nodes in the DAG according to some metric. Later the scheduler performs an assignment of the jobs to processors in the order of jobs priorities. When a job is assigned to a node it is removed from the priority list. The assignment can take place either *statically* (before actual program execution execution) or *dynamically* (while program is actually running). For further details on static list scheduling see Section 2.6.1. For further details on dynamic list scheduling see Section 2.6.2.

If assignment takes place during runtime there can happen following situation. Assume there is a processor available for execution and some jobs from the priority list are in ready state. But the most prioritised job is not ready, because has some dependencies not met yet. This can happen if the top job in the priority queue depends on a job which is currently being run on a processor. There are two solutions for this situation. In the

first one, scheduler waits until the top job becomes ready. In the other one, scheduler takes the most prioritised jobs among the ready ones. If scheduler never intentionally waits it is called *eager* [CJ07].

2.6.1 Static list scheduling

Static list scheduling algorithms are popular in research [ACD74; KA99b; WS97] and are often combined with other scheduling heuristics. Algorithms of this class are the simplest among other list scheduling algorithms.

The scheduling algorithm first arranges the jobs into the list of the nodes. This list has jobs ordered according to some priority. Priority choice depends on the algorithm and its computation complexity varies, but it is important to choose priority in a way that will keep jobs in the list topologically sorted. If this condition is met and if the queue of yet to be scheduled jobs contains at least one ready job, than the most prioritised ready job will be on the top.

After creating a priority queue of the jobs, the scheduling algorithm consists of two steps [KA99a]:

1. Remove the top node from the scheduling queue;
2. Allocate the node to a processor that allows the earliest start-time.

Initial assumption is that jobs are scheduled in a way that every new job appears in the end of the local schedule of a processor. It is simple to implement, but can be improved by *insertion heuristic*[3] [Kru87]. In arbitrary structured DAG dependencies can be such, that schedule will have *holes*. Holes are periods between job executions when a processor has no job to run. Since simple static algorithm does ignores holes, they arise fast. A scheduler with insertion heuristic assigns jobs to processors considering holes and tries to fill them up, when possible.

Figure 2.4 shows an example of assignment of jobs to the processors. Here P_i are the processors, v_i are the computation jobs, e_1 is the communication job. Jobs v_2 and v_3 depend on the job v_1, thus they ought to be executed after v_1 completes.

From the figure one can see that up to the time 30 P_2 has no job running. Consider, that next job to be scheduled is job v_4, which has no data dependencies. Possible time slots for these jobs are marked with dashed boxes. With simple variant of static algorithm it is going to be scheduled to either P_1 or P_2 at time 40. But with insertion approach algorithm is capable to allocate time slot at time 0 on P_2.

2.6.2 Dynamic list scheduling

Dynamic list scheduling algorithms are the extension of static list scheduling algorithms. Dynamic list scheduling algorithms may change the priority queue of the jobs while constructing it. This happens, because after adding the new node the metric used to assign priorities should be recalculated for all the nodes that are already in the priority queue. Thus, a static list scheduling algorithm gets an additional step in a priority queue construction [KA99b]:

[3] Also known as *insertion approach*.

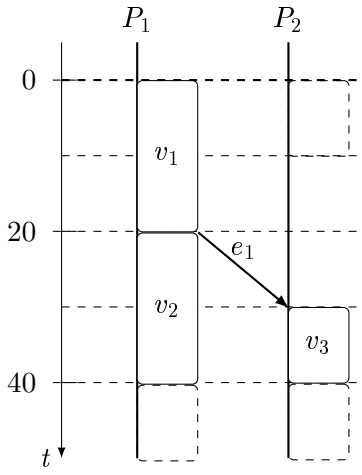


Figure 2.4: Insertion approach explanation

1. Determine new priorities of all unscheduled jobs;
2. Select the job with the highest priority for scheduling;
3. Allocate the node to the processor that allows earliest start time.

Dynamic scheduling has a potential to generate a better schedule than static one, but as drawback it requires continuous recalculation of priorities for the priority queue, thus increasing time complexity of the algorithm.

2.7 Other approaches

This section describes other scheduling models. Detailed description is out of scope of this thesis, but a short description of these related models is important for understanding the research area.

Often differences in the models can be seen as extensions to the basic system model. Although I separate scheduling heuristics in several subsections it is important to mention that they often can be combined in the same algorithm to improve overall performance. Good example of such combination is a combination of clustering heuristic (see Section 2.7.1) and task duplication heuristic (see Section 2.7.2). An example of such combination is LCTD algorithm [CSM93].

2.7.1 Clustering heuristics

Clustering heuristics [Sin+08; LP96; KA99b; GY92] assume that the number of processors is effectively unlimited. Thus the goal of minimizing the makespan is reduced to the problem of optimizing communication costs. In the beginning clustering algorithms assign each job a separate processor (*cluster* in clustering algorithms terminology). In the process of looking for the better schedule an algorithm unites the clusters. The

unification of the clusters means assignment of the jobs from different clusters to the same processor. When two jobs, which have direct dependency between each other, are put on the same cluster, communication costs between these jobs become zero.

When number of physical processors is not less than the number of clusters assigning processors to cluster is a trivial problem. But when number of clusters is bigger than number of processors additional step called *mapping* is required to accomplish scheduling process. During this step a scheduler has to map clusters to physical processors introducing the least possible degradation of the resulting schedule. Considering the fact that the quality of the resulting schedule highly depends on this step [KA99b], it is an open question, if putting the actual number of processors out of scope is worth it.

2.7.2 Task duplication

The problem that task duplication aims to solve called *max-min problem*. Both the heuristic and the problem were presented by Kruatrachue [Kru87]. The essence of max-min problem lies in an observation that within macro dataflow model distribution of the jobs to bigger amount of processors leads to bigger communication delays. At a certain number of processors makespan of the program even tends to increase if maximal number of them is used, because the fraction of communication costs grows and starts to dominate in total execution costs.

Task duplication [Shi+08; AK94] attempts to reduce communication overhead by cloning the task that introduce much of communication costs. These task are distributed among several processors, so that more communication happens locally. Thus task duplication heuristic takes advantage of parallelism and reduces the communication delays at the same time.

2.7.3 Guided random search based algorithms

Deterministic algorithms have efficiency imbalance for different configurations of parallel application workflow. With very few luck performance degradation can be significant. Guided random search based (GRSB) algorithms attempt to solve this problem by introducing randomization making corner cases less likely. Different types of these algorithms include genetic algorithms [Gra99; AKN05; WBS09; Koł+13], Tabu search, Simulated Annealing, etc. [Bra+01].

The idea of GRDB algorithms lies in generating many possible schedules and later selecting the better ones. The process of generation and selection is iterative. Typically a guided random search based algorithm starts from generating some random schedules [Gra99; AKN05; WBS09; Koł+13], but is also possible to base a GRDB algorithm on a schedule obtained by a deterministic algorithm [Bra+01]. The number of iterations of algorithm can be volatile: algorithm runs as long as it is possible to bring a sensible improvement to the schedule. Or it can be fixed to some constant number.

Randomness is not a silver bullet. As with deterministic algorithms, stochastic ones heavily depend on the quality of the metrics that are used to compare intermediate schedules and the process of generating schedules for next generation. Sometimes huge

number of iterations is required to get acceptable quality schedule, which can increase scheduling overhead dramatically.

2.8 Conclusion

This section considers state of the art system models and scheduling models. There exist big variety of such models, thus only some of the aspects were taken into consideration.

Together with full-ahead and just-in-time models I presented a novel Horizon model. This model combines properties of both full-ahead and just-in-time models. I mentioned possible gains of such models and discussed applicability of this model.

In comparison to full-ahead scheduling model, the Horizon model can act in situations, where full-ahead model lacks necessary information. The Horizon model enables scheduling, when there is information about program structure, but for some reasons creating schedule in advance is impossible. The Horizon model is expected to be more robust in presence of imperfection of cost estimations.

In comparison to just-in-time model, horizon model allows to use more information, if available. This allows to assign jobs to better fitting time slots and start communication operations beforehand.

Additionally to the simple models, I described approaches, which can be seen as the extension to basic models. Often these extensions are orthogonal and can be applied independently.

3 Scheduling Algorithms

This section describes concrete algorithms from research. Although these algorithms use various models presented in Section 2, I compare them within the same homogeneous model from Section 2.1.

Section 3.6 presents the Horizon algorithm developed within the scope of this master thesis. It makes additional assumptions on the system model, which requires some changes to the scheduling model. This section describes these changes in greater detail and argues why they are reasonable and applicable in real life scenarios.

Section 3.7 describes a just-in-time algorithm. The algorithm was implemented to provide a baseline for comparison against the other algorithms. It has the simplest system model and thus the performance of this algorithm can be considered as the worst case performance, showing how the other algorithms can improve.

3.1 HLFET algorithm

Adam, Chandy, and Dickson [ACD74] proposed the HLFET algorithm, which is one of the simplest list scheduling algorithms. HLFET stands for highest levels first with estimated times, meaning that the algorithm prioritizes nodes according to their s-level. "With estimated times" means that estimated times for node weights are known. The algorithm itself is oblivious to communication costs.

I use the description of the HLFET algorithm given by Kwok and Ahmad [KA99b]:

1. Calculate s-level of each node;
2. Put nodes in a ready queue according to descending order of their s-levels. Initially the ready queue contains only the entry node. If nodes have the same s-level, ties are broken arbitrarily;
3. Schedule the top node in the ready queue to a processor that allows minimal EST. This algorithm does not use insertion heuristic;
4. Update the ready queue by inserting the nodes that are now ready.
5. If the ready queue is not empty, go to step 2. The algorithm finishes, otherwise.

Complexity of the algorithm is defined by the loop within steps 2–5 plus the complexity of step 1. Calculation of s-levels has complexity $O(v + e)$. Complexity of step 2 is $O(v)$, because one has to update the ready queue and to check up to v nodes. Since, step 2 repeats v times, the total complexity of the algorithm is $O(v^2)$.

Figure 2.2 presents an example of a schedule produced by the HLFET algorithm for the DAG shown in Figure 2.1. And the parameters denoted in Table 2.2 on page 2.2.

3.2 MCP algorithm

MCP stands for *modified critical path*. This algorithm was originally proposed by Wu and Gajski [WG90]. The algorithm arranges nodes in a priority queue in descending order of their ALAP metric. MCP is popular and efficient algorithm. And thus has many implementations. I cite the **simplified** MCP algorithm, proposed by original authors in [Wu00]. It has a smaller asymptotic complexity, but practically the same performance as the original MCP. The steps of the algorithm are:

1. Compute $ALAP$ metric for each node in a DAG;
2. Put the nodes in the priority queue in ascending order of their ALAP times. Ties are broken by the child that has the smallest ALAP times. If children of contending nodes have the same minimal ALAP times, ties are broken arbitrarily;
3. Pop the top node from the priority queue and schedule it to the processor that allows the earliest start time, using the insertion approach (see Sec. 2.6.1). Repeat step 3 until the priority queue gets empty.

ALAP time is computed by traversing all the edges of the DAG, this step has a complexity of $O(e)$. Step 3 performs a sort according to ALAP values and has a complexity of $O(v \log v)$. The complexity of step 3 consists of two parts. In the first part $EST(v_i)$ is determined, by traversing all the parents of node v_i. This part has complexity $O(|pred(v_i)|)$ for a single node. For v nodes the complexity is $\sum O(|pred(v_i)|) = O(e)$. After this the scheduler looks for a time slot for node v_i in the partial schedule using the insertion approach. This part has a complexity of $O(v)$, because the number of possible time slots is less than the number of nodes. Step 3 repeats v times, resulting in $O(v^2)$ complexity. The total complexity of the algorithm is $O(e + v \log v + v^2)$ or $O(v^2)$, because $e < v^2$.

ALAP times for the DAG from Figure 2.1 are shown in Table 2.2.

3.3 ETF algorithm

The ETF (Earliest Time First) algorithm was proposed by Hwang et al. [Hwa+89]. The algorithm consists of following steps [KA99b]:

1. Compute static b-levels of all the nodes in the DAG;
2. Out of the ready queue take the node v_i which has the lowest EST. Ties are broken by selecting the node with a higher static b-level. Schedule node v_i to a corresponding processor without applying the insertion approach.
3. Update the ready queue with those nodes whose dependencies are met after v_i is scheduled.
4. If the ready queue is not empty, go to step 2.

The time complexity of the algorithm is $O(pv^2)$ (see proof in [Hwa+89]).

3.4 HEFT algorithm

HEFT (Heterogeneous earliest time first)[THW02] represents a family of algorithms for heterogeneous systems. This class of algorithms is widespread in state-of-the-art distributed systems.

Given heterogeneity, computation of b-levels or t-levels has no sense anymore, because the longest path in a graph does not characterize upper limit of a schedule length. Thus, HEFT uses another metric.

First, the computation and communication costs of the jobs are aggregated into a single value, to simplify the ordering of the nodes in the priority queue. I introduce the mean computation costs of a node v_i as:

$$\overline{w_i} = \frac{\sum_{j=1}^{p} w(v_i, j)}{p}.$$

The mean communication costs of a node v_i are:

$$\overline{c_{i,j}} = \overline{L} + \frac{data_{i,j}}{\overline{B}},$$

where $\overline{L}$ is the average communication latency and $\overline{B}$ is the average bandwidth of all processors in the system. Most of the parameters defined in Section 2.1.1 are calculated in the same way, but with respect to which processors run specific node and which communication channels are used.

To arrange nodes in a priority list, the algorithm uses for each node an *upward rank*. It is important to note, that nodes are ordered by upward rank are also ordered topologically. The upward rank is defined as:

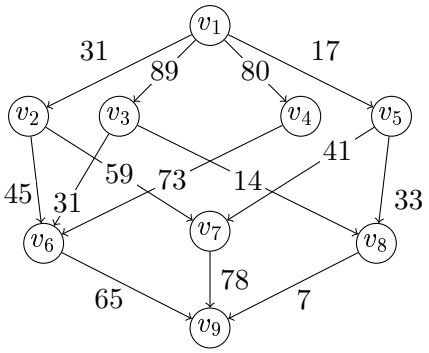
$$r(v_i) = \begin{cases} \overline{w_i} + \max\limits_{v_j \in succ(v_i)} (\overline{c_{i,j}} + r(v_j)), & \text{if } succ(v_i) \neq \emptyset \\ \overline{w_i}, & \text{if } succ(v_i) = \emptyset \end{cases}$$

The algorithm steps are the following:

1. Arrange nodes in a priority queue according to their nonincreasing order of upward ranks. Ties are broken arbitrarily;
2. Pop the first node from the priority queue and compute the EFT for each processor, using the insertion approach;
3. Schedule the node to the processor with the earliest finish time for this node.
4. If the priority queue is not empty, go to step 2.

The HEFT algorithm has $O(v \times p)$ complexity. For a dense graph where the number of edges is proportional to $O(v^2)$, the time complexity is $O(v^2 \times p)$.

To make good comparison with the next algorithm, I provide an example from Bittencourt, Sakellariou, and Madeira [BSM10]. The goal is to schedule an application whose DAG is presented in Figure 3.1. Figure 3.1a shows an example application which



(a) Task precedence graph of a parallel application.

	P_1	P_2	P_3
v_1	20	40	30
v_2	30	45	30
v_3	35	35	65
v_4	15	25	35
v_5	30	50	55
v_6	45	30	60
v_7	15	25	20
v_8	15	20	25
v_9	35	70	75

(b) Parameters of the heterogeneous system required to schedule this parallel application.

Figure 3.1: An example of a parallel application for a heterogeneous system.

	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9
$\overline{w_i}$	30	35	45	25	45	45	20	20	60

(a) Mean weights of nodes.

	v_1	v_4	v_2	v_3	v_5	v_6	v_7	v_8	v_9
$r(v_i)$	356	268	252	246	244	170	158	87	60

(b) Priority list of node ranks (left is the top).

	v_1	v_4	v_2	v_3	v_5	v_6	v_7	v_8	v_9
$EFT_1(v_i)$	20	35	65	100	130	145	186	193	269
$EFT_2(v_i)$	40	125	96	174	87	229	149	205	291
$EFT_3(v_i)$	30	135	81	135	92	259	165	154	309
Processor	p_1	p_1	p_1	p_1	p_2	p_1	p_2	p_3	p_1

(c) Scheduling steps according to the HEFT algorithm.

Table 3.1: Metrics used by the HEFT algorithm.

I are going to schedule. Table 3.1 shows how many timeunits each nodes requires on each of available processors.

For the sake of simplicity we assume communication costs to be homogeneous, all the communication channels have unit bandwidth and introduce zero latency, thus we can assume already that $\overline{c_{i,j}}$ is given by the edge weights in Figure 3.1a. The computed $\overline{w_i}$ are given in Table 3.1a. Computed upward rank $r(v_i)$ is given in Table 3.1b.

When the priority list is created, the scheduler start assigning nodes to processors. At each step HEFT assigns a node to the processor, which allows the earliest finish time, as described above. The process of assignment is traced in Table 3.1c. In this table steps of execution go from left to right. $EFT_j(v_i)$ shows the EFT of a node v_i on the processor p_j. The row labeled *processor* shows to which processor the scheduler decided to schedule this node.

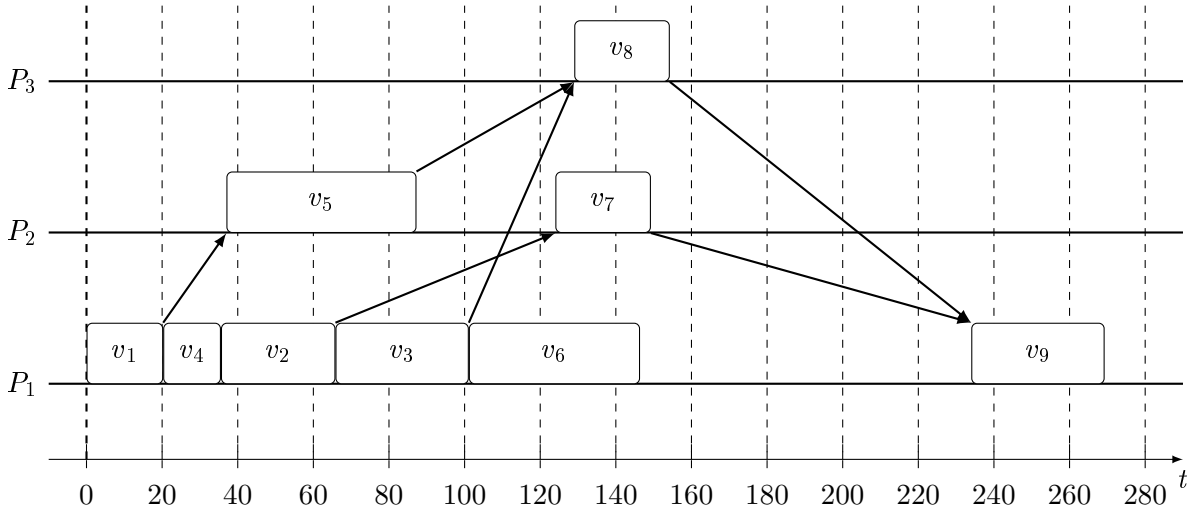


Figure 3.2: Parallel application scheduled by HEFT algorithm.

The schedule produced by a HEFT algorithm is presented in Figure 3.2.

3.5 Lookahead HEFT algorithm

An extension of the classical HEFT algorithm was proposed by Bittencourt, Sakellariou, and Madeira in [BSM10]. The original Lookahead algorithm model experienced a major simplification in this thesis, because I consider Lookahead here in the context of homogeneous system model. Thus one can argue that under different conditions this algorithm could have shown different performance in comparison to the other algorithms.

To understand the benefits of the proposed extension, let us consider first the schedule from the previous section. As it turns out, if node v_7 is scheduled to processor p_1, the makespan of the application would be smaller. This happens because the communication costs between nodes v_9 and v_7 introduce a big delay, which delays the start of node v_9. These delays are bigger than the computation costs of node v_7 itself.

Unfortunately classical HEFT can't foresee such an outcome, because in the scheduling phase it considers only one node at a time, ignoring any consequences of such a decision. As stated by Bittencourt, Sakellariou, and Madeira, minor search space expansion can introduce a significant benefit in scheduling efficiency. The idea of considering several nodes at a time, trying to predict the consequences of next scheduling decision was called Lookahead.

There are four variants of the lookahead algorithm proposed in the paper [BSM10]. I am going to introduce all of them, but first start with the basic one, which is called *Lookahead*. The idea is the following. Assume that the node v_i is the current one. The goal is to minimize the maximum EFT among all v_i's children on all processors where the current node is probed. The scheduling steps are the following.

1. Arrange nodes in a priority list according to their upward rank (as in HEFT);

2. Remember the current scheduling state;
3. Take the top node v_t from the priority queue and schedule it to processor p_i;
4. Calculate the EFT for all the children of node v_t, assuming it is scheduled on processor $p(v_t)$ and determine the maximum finishing time among all these children (CFT)

$$CFT_{p_i}(v_t) = max(\{EFT(v_j)|v_j \in succ(v_t)\});$$

5. Return to the scheduling state as in step 2;
6. If the node v_t has not yet been tried on all processors, return to step 2 and try it on another processor.
7. Schedule v_t on processor p_i such that $CFT_{p_i}(v_t) < CFT_{p_j}(v_t), \forall p_j \in P, p_i \neq p_j$

It can happen in step 4 that the children of the task v_t have other unmet dependencies to the point when the task v_t is scheduled, besides the dependency to v_t. If this is the case, the children can't be scheduled just after the node v_t. To overcome this, when the algorithm computes the earliest finishing time of the children, it considers only the current state of the schedule and assumes there are no further unmet dependencies. In this sense the Lookahead algorithm is rather optimistic, because it estimates CFT ignoring any further delays that may arise due to any unscheduled nodes.

Bittencourt, Sakellariou, and Madeira proposed a second approach to estimate the earliest finishing time of the children. They noticed that sometimes children could have a higher priority, because they are on the critical path. Because of this higher priority, children could be treated in a different way. Such a method of CFT estimation is called Lookahead with weighted average EFT. Instead of taking the maximum EFT, the $CFT_{p(v_t)}(v_t)$ is computed as follows.

$$CFT_{p_i}(v_t) = \frac{\sum_{v_j \in succ(v_t)} r(v_j) \cdot EFT(v_j)}{\sum_{v_j \in succ(v_t)} EFT(v_j)},$$

where p_i is the processor where v_t is attempted to be scheduled.

An extension of the basic Lookahead algorithm is presented in the same paper. It attempts to improve scheduling performance by increasing the search space. If the two top nodes of the priority queue are independent of each other, the scheduler attempts to swap them and compare the schedules when the top node is scheduled first and when the top node is scheduled second. The idea of this extension is based on an observation that changing the scheme for rank computation, or even combining different ranks, may give better performance [SZ04a; ZS03] of the scheduler. Since rank computation may change priority ranks only to a limited extent (at least topological order should be preserved), Bittencourt, Sakellariou, and Madeira assumed that small changes in priority order also can bring an improvement.

The steps of the complete algorithm are as follows:

1. Arrange nodes in a priority list according to their upward rank (as in HEFT);
2. Remember the current scheduling state;
3. If the second top node does not depend on the top node in the priority queue, create a supplementary priority queue L of children of the top node, the second top node, and the children of the second top node.

   $$L = succ(v_t) \cup v_s \cup succ(v_s),$$

   where v_s is the second top node in the main priority queue. If v_s depends on v_t, create L only out of the children of the top node.
4. Take the top node v_t from the priority queue and schedule it to the processor p_i;
5. Calculate the EFT for all nodes in the queue L and determine the maximum finishing time among all the children (CFT)

   $$CFT_{p_i}(v_t) = max(\{EFT(v_j) | v_j \in succ(v_t)\});$$

6. Return to the schedule state as in step 2;
7. If the node v_t hasn't been tried on all the processors, return to step 2 and try it on another processor.
8. Restore the scheduling state from step 2.
9. Swap the top and second top nodes in the priority queue and repeat the steps 2–8. If v_s depends on v_t, this step is skipped.
10. Schedule the node with the lowest $CFT_{p_i}(v_j)$, where v_j is one of the two top nodes and p_i is the processor which allows lowest CFT for a given node.

As with the basic variant of Lookahead, the weighted CFT can be used.

An example of schedule of an application with the DAG shown in Figure 3.1a is presented in Figure 3.3.

3.6 Horizon algorithm

In Section 2.5.3 I described the general principles of Horizon schedulers. Here I detail a particular implementation, which has been done as part of this thesis.

The algorithm description is given in Figure 3.4. This algorithm is an attempt to reduce the full-ahead algorithms to the Horizon model. At each step of execution, the scheduler the considers DAG in the horizon in the same way as it is considered by a full-ahead algorithm. Scheduling at each step consists of creating a priority queue of the

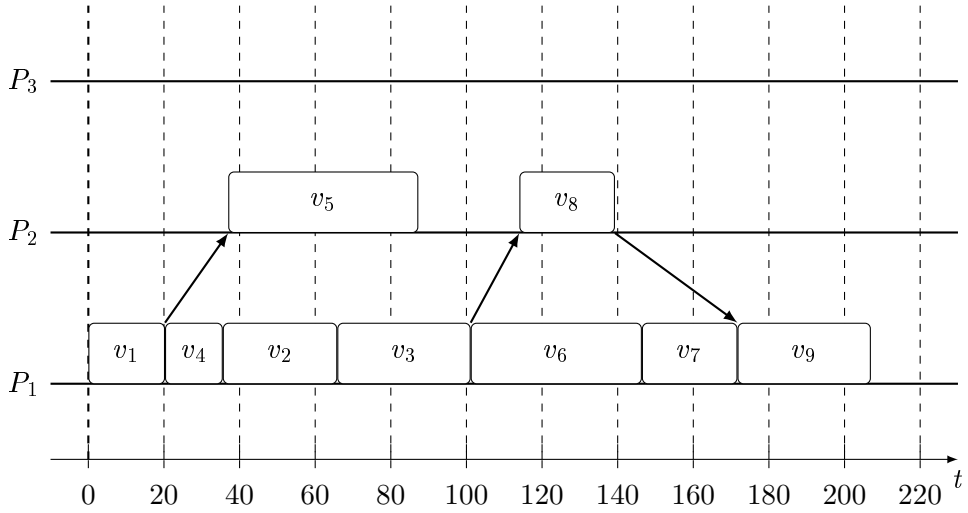


Figure 3.3: Parallel application scheduled by Lookahead HEFT algorithm.

schedulable nodes. To create the priority queue, as with full-ahead algorithms, various metrics can be used: t-level, b-level, difference between t-level and b-level, ALAP and others. My experiments did not reveal a big difference, but ALAP seemed to be the better one.

After the priority queue is created, each schedulable node is assigned to the processor, which allows earliest possible finishing time. Processor allocations which were made during previous iterations should be taken into account to avoid interference. In this context scheduling of one big DAG is reduced to scheduling of many small DAGs.

Operation *Schedule* in Line 22 can be implemented in two ways. The first way is to allocate time on a processor at the end of the processor's schedule. An alternative is to use the insertion approach and allocate time for the job in the processor's schedule as early as possible. The insertion approach has a higher computational complexity, but enables better scheduling towards the end, thus I have chosen the insertion approach in my evaluation.

Figure 3.5 shows a schedule of the DAG in Figure 2.1. States of the scheduler are shown in Table 3.2. The depth is assumed to be equal to 2.

In the example, communication between the jobs v_{10} and v_{11} cannot start immediately after the job v_{10} finishes. First time when it is possible to allocate time slot for such a communication operation, after the job v_{10} finishes is it $t = 110$.

Table 3.2 shows that the scheduler is invoked only 4 times. This happens because after v_5 finishes, all unfinished jobs appear in the horizon. This allows Horizon to schedule the rest of the DAG all at once.

3.6.1 Complexity analysis

The algorithm complexity can be found by combining complexities of all parts of the algorithms. These parts are the loop in Lines 1–3, the loop in Lines 5–11, the sort operation in Line 12, and finally the loop in Lines 13–23.

Require: $Depth \geq 1$
Require: $Horizon \leftarrow$ horizon within depth $Depth$
Require: $Processors \leftarrow$ set of processors
Require: $Horizon$ has just changed
1: **for** $v_i \in Revtop(Horizon(Depth))$ **do**
2: $blevel(v_i) \leftarrow max(blevel(v_j) + w(e_{i,j}) \mid v_j \in (children(v_i) \cap Horizon)) + w(v_i)$
3: **end for**
4: $CP \leftarrow max(blevel(v_i) \mid v_i \in Horizon)$
5: **for** $v_i \in Revtop(Horizon(Depth))$ **do**
6: **if** $children(v_i) \cap Horizon = \emptyset$ **then**
7: $ALAP(v_i) \leftarrow CP - w(v_i)$
8: **else**
9: $ALAP(v_i) \leftarrow min(ALAP(v_j) - w(e_{i,j}) \mid v_j \in (children(v_i) \cap Horizon)) - w(v_i)$
10: **end if**
11: **end for**
12: $Schedulable \leftarrow Sort_{ALAP}(v_i \mid parents(v_i) \subseteq Horizon)$
13: **for** $v_i \in Schedulable$ **do**
14: $WS_{min} \leftarrow \emptyset$
15: $EFT_{min} \leftarrow \infty$
16: **for** $ws_j \leftarrow Processors$ **do**
17: **if** $EFT(v_i, ws_j) < EFT_{min}$ **then**
18: $WS_{min} \leftarrow ws_j$
19: $EFT_{min} \leftarrow EFT(v_i, ws_j)$
20: **end if**
21: **end for**
22: $Schedule(v_i, WS_{min})$
23: **end for**

Figure 3.4: Horizon scheduling algorithm.

Time	Job	ALAP	P_i
0	Horizon: v_1, v_2, v_5		
	v_1	0	P_1
	v_2	40	P_1
	v_5	50	P_2
10	Horizon: $v_2, v_3, v_6, v_5, v_{10}$		
	v_6	0	P_2
	v_3	10	P_1
	v_{10}	20	P_3
30	Horizon: $v_3, v_4, v_6, v_7, v_8, v_{10}, v_{11}$		
	v_4	40	P_1
	v_8	50	P_2
	v_7	60	P_3
50	Horizon: $v_4, v_6, v_7, v_8, v_9, v_{10}, v_{11}, v_{12}$		
	v_9	110	P_2
	v_{11}	160	P_2
	v_{12}	200	P_2

Table 3.2: States of the scheduler, which schedules the DAG from Figure 2.1.

Before continuing with the complexity analysis of the aforementioned parts of the algorithm, I need to prove the following lemmas.

Lemma 3.6.1. *If the unscheduled DAG depth is less than or equal to the Depth, Horizon(Depth) will schedule the DAG in one invocation of the algorithm given in Figure 3.4.*

Proof. If a DAG has depth less than or equal to *Depth*, all jobs are present in the Horizon. This means that there is no job within the horizon, which depends on a job, which is not part of horizon. Hence, all the jobs within the horizon are schedulable. If all the jobs are schedulable, the Horizon algorithm will schedule all of them, like any full-ahead algorithm. □

Before the next lemma, I introduce DAG from Figure 3.6. This DAG represents the worst-case scenario for the Horizon scheduler with depth d. Unfortunately, it was impossible to show all edges of the DAG, thus only some of them are depicted. The worst case DAG has an outgoing edge from every node to every descendant node. Exceptions are the nodes with indices $v'_d, v'_{2\cdot d}, \ldots, v'_{n\cdot d}$. These nodes have only one ingoing edge each from nodes with indices $v_d, v_{2\cdot d}, \ldots, v_{n\cdot d}$, respectively.

Lemma 3.6.2. *If the unscheduled DAG depth is bigger than Depth and the number of unscheduled jobs is also bigger than Depth, at least Depth jobs are schedulable.*

Proof. Consider the DAG from Figure 3.6. When the program starts, the horizon is filled up with all the nodes, except the nodes $v'_d, v'_{2\cdot d}, \ldots, v'_{n\cdot d}$. In the beginning only d nodes are schedulable, because all the prime nodes are out of the horizon and any node with index bigger than $d-1$ depends on a node, which is out of the horizon.

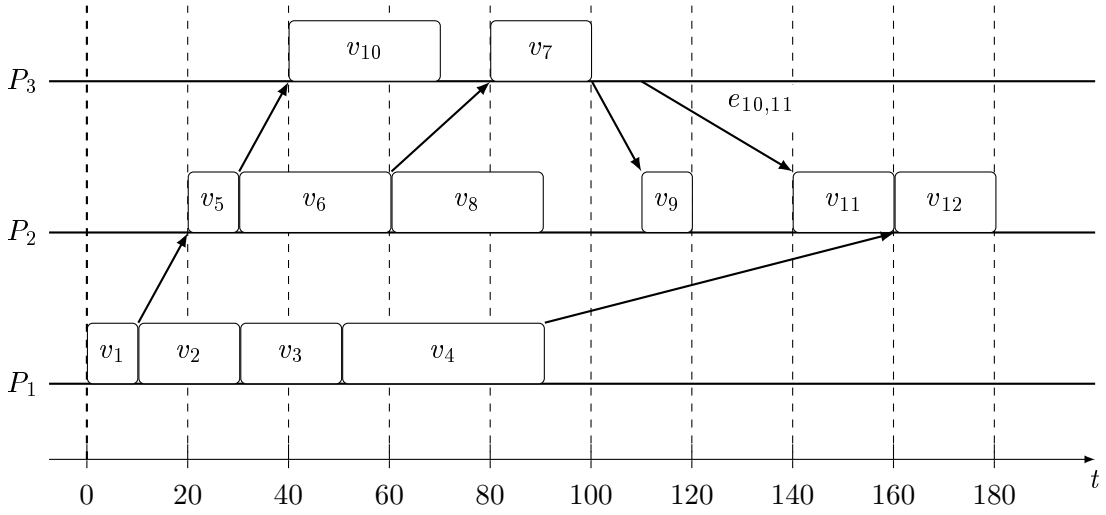


Figure 3.5: Schedule example by the Horizon algorithm.

When node v_1 is finished, node v'_d appears in the horizon. This turns the nodes $v_{d+1}, \ldots, v_{2 \cdot d}$ into schedulable nodes. Together with node v'_d d schedulable nodes appear.

Next time the horizon changes only when the node v_{d+1} finishes. At this point the node $v'_{2 \cdot d}$ appears in the horizon. Similar to the previous iteration, d nodes become schedulable. And so forth and so on until the whole DAG is scheduled.

If the worst case DAG has any additional edge, this edge can end in a v'_{id} node, because other nodes are already in the horizon and have the maximal possible number of ingoing edges. Without loss of generality assume the edge is added from a $v_i, (i = 1 \ldots d-1)$ to the v'_d. This makes the shortest path from the root to the node v'_d shorter than $Depth$. In this case v'_d appears in the horizon and becomes schedulable. Together with the v'_d the nodes $v_{d+1}, \ldots, v_{2 \cdot d}$ also become schedulable. In the result the scheduler schedules $2 \cdot d + 1$ nodes at once. □

Taking into account that each time at least $Depth$ nodes are scheduled, the Horizon algorithm will be invoked at most $\frac{v}{d}$ times.

The complexity of the loop 1–3 depends on the number of the children every job has. In the worst case scenario the top job has $O(v)$ children. Each descendant has one child less. The jobs $v_d, \ldots, v_{n \cdot d}$ introduce a constant mistake and does not change the picture. The last job has only 1 child. Thus, in the beginning the complexity of this loop can be calculated according to the following formula.

$$T(v) = \frac{O(v) + 1}{2} \cdot O(v) = O(v^2)$$

In the beginning, the horizon contains almost all the jobs and after each invocation of the scheduling algorithm at least $Depth$ of them are scheduled. This mean that the complexity of the loop will decrease with each invocation. Using the formula of the sum of arithmetic series, the complexity of the loop 1–3 is the following.

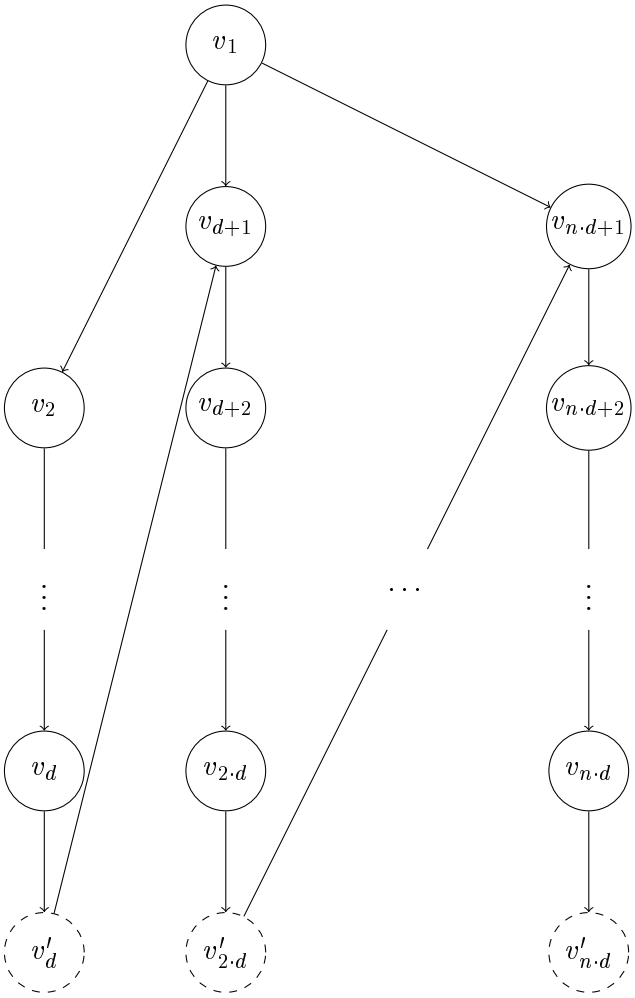


Figure 3.6: Worst case DAG. N. B. Not all edges are shown. Dashed nodes are not in the horizon.

$$T(v) = \frac{O(v^2) + 1}{2} \cdot \frac{v}{d} = O(\frac{v^3}{d})$$

The same reasoning applies to the loop 5–11.

The complexity of the sort operation in line 12 is calculated, assuming that the number of jobs to be scheduled is *Depth* all the time. The complete complexity of the sort operation through all the invocations of the algorithm is the following.

$$T(v) = O(\frac{v}{d} \cdot d \log d) = O(v \log d)$$

The loop 13–23 is invoked once for every job, resulting in v iterations in total through all invocations of the scheduling algorithm. The loop body contains an inner loop and an operation which allocates a time slot in the processor's schedule. The complexity of

Require: $Depth \geq 1$
Require: $Ready \leftarrow$ list of ready nodes
Require: $Processors \leftarrow$ set of processors
1: **while** $Ready \neq \emptyset$ **do**
2: $v_{min} \leftarrow v_0$
3: $p_{min} \leftarrow p_0$
4: **for** $v_i \in Ready$ **do**
5: **for** $p_i \in Processors$ **do**
6: **if** $EFT(v_i, p_i) < EFT_{min}$ **then**
7: $v_{min} \leftarrow v_i$
8: $p_{min} \leftarrow p_i$
9: **end if**
10: **end for**
11: **end for**
12: $Schedule(v_{min}, p_{min})$
13: $Ready \leftarrow Ready \setminus v_{min}$
14: **end while**

Figure 3.7: Greedy scheduling algorithm.

the latter operation is equivalent to determining the EFT, thus the complexity of the inner loop dominates over the complexity of the *Schedule* operation.

The complexity of the EFT operation is equal to $O(v)$, because every gap between each two pairs of scheduled jobs should be probed. The total complexity of the last loop is following.

$$T(v) = O(v) \cdot p \cdot v = O(v^2 \cdot p)$$

The overall algorithm complexity comprises the complexities of the loops 1–3, 5–11, 13–23, and the sort operation at line 12, which is following:

$$T(v, d, p) = O(v^2 \cdot p) + O(v \log d) + O(\frac{v^3}{d}) = O(v^2 \cdot p) + O(\frac{v^3}{d})$$

The first and the last loops have the highest asymptotic complexity, they also determine the complexity of the whole algorithm.

3.7 Greedy algorithm

Greedy algorithm is a very simple implementation of a just-in-time algorithm, which was mainly used for comparison with other algorithms. Its algorithm is given in Figure 3.7. It is assumed, that the schedule operation attempts to append nodes at the end of the processors' time line. The goal of the algorithm is to find pairs of processor and ready job, which allows the earliest finishing time, whenever there appear any number of ready jobs.

The complexity analysis of this algorithm is quite simple. The while loop always takes v iterations. The outer for loop requires a number of iterations, which is equal to the ready set. The inner for loop requires a number of iterations, which is equal to the number of processors.

The worst case complexity shows up, when all jobs are ready from the beginning. In this situation, in the first iteration of the while loop, v iterations of the outer for loop are required. In the second iteration of the while loop, $v - 1$ iterations of outer for loop are required, and so on, until the ready queue gets empty. In total, the inner for loop will be executed

$$\frac{v+1}{2} \cdot v = \frac{v^2 + v}{2}$$

times.

The schedule operation and EFT determination takes the same amount of time. Both of these operations check only the end of processor's time line, thus both of these operations have constant asymptotic complexity.

The total algorithm complexity is equal to the number of times the inner loop is invoked multiplied by the complexity of the inner loop.

$$T(v, p) = \frac{v^2 + v}{2} \cdot p \cdot 1 = O(v^2 \cdot p).$$

In the next section, I evaluate the performance of the algorithms presented here.

4 Evaluation

This section shows the comparison of the algorithms from Section 3. Two main aspects are analyzed: performance and robustness. The comparison is done by scheduling various DAGs and measuring the speedup, which every algorithm can provide.

Section 4.1 describes the DAGs, which are used for the comparison. Section 4.2 shows comparison of the algorithms in regard to their performance. Section 4.3 shows the comparison in regard to their robustness. Section 4.4 explains some of the patterns which were discovered during the evaluation. Section 4.5 summarizes the results of the evaluation.

4.1 Benchmark applications description

Two types of graphs were evaluated. Graphs of the first type are traced graphs. Their structure and cost estimation was acquired by gathering run-time information about real parallel program execution. The second type represents generated random graphs.

4.1.1 Traced graphs

The comparison of scheduling algorithms using traced task graphs is common in research [Sin+08; Can+08]. Such evaluations balance between being realistic and being reproducible. The comparison is realistic because the structure of an application is taken from a real-world scenario. It is also reproducible because one can schedule the same DAG using a different algorithm.

Traced graphs for this particular evaluation were collected from a set of HPC-applications. They have different structure, sometimes very simple. However, they show how an algorithm could behave under real-world workload.

The applications for evaluation were taken from a distribution of the StarPU parallel programming framework [Aug+11]. This distribution contains examples of parallel application, which can be run by this framework. The descriptions of applications are as follows:

- Application `mult` is a simple implementation of a blocked matrix multiplication. Its DAG consists of 16 independent pipelines, resulting in 160 jobs in total. An example of the structure is shown in Figure 4.1.
- Application `lu` is an implementation of LU matrix decomposition. Its DAG structure has a high level of parallelism in the beginning, decreasing closer to the end. Another pattern of this DAG is that segments with high level of parallelism are interleaved with segments with lower level of parallelism. The DAG used for

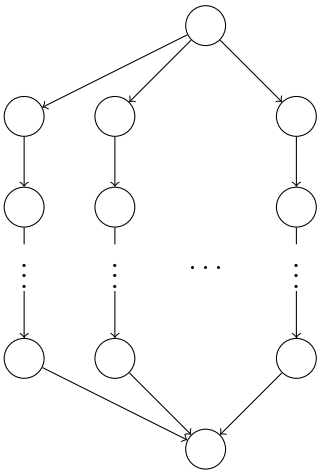


Figure 4.1: Example of `mult` structure.

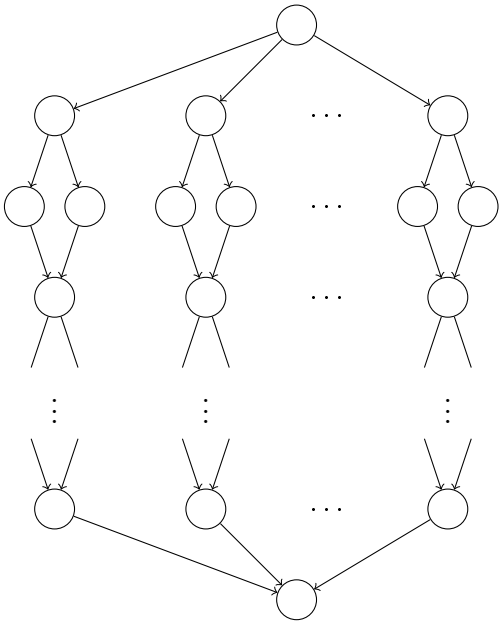


Figure 4.2: Example of `pipeline` structure.

the evaluation contains 1496 jobs. An example with its DAG structure is shown in Figure 4.3.

- Application `heat` has the same structure as `lu`. In fact, it uses the same LU matrix decomposition, but tuned for a particular purpose. However, it differs from `lu` in the computation cost estimations. An example showing a DAG structure of this application is depicted in Figure 4.3, the same as `lu` application.

- Application `cholesky` is an implementation of Cholesky matrix decomposition. Its DAG structure somewhat similar to the structure of the `lu` and the `heat` applications. The DAG used for evaluation has 816 jobs.

- Application `pipeline` consist of series of iterations. Each iteration multiplies two vectors and sums them up. Its DAG structure is very similar to the structure of the `mult` application. The structure of `pipeline` application is given in Figure 4.2. The total number of jobs in this application is 1024.

- Application `stencil` is an example which shows a basic model within the StarPU framework. The application structure resembles so-called nearest neighbor communication pattern. The DAG contains 6144 jobs. The structure is shown in the Figure 4.4.

  Application `stencil` is very communication intensive. The structure of the DAG can be seen as an $N \times N$ matrix. Execution goes from top to bottom. For each row, a job in the i-th column has two outgoing dependencies to a column $i - 1$ and to a column $i + 1$. In the end every job of the last row transitively depends on every job from the first row.

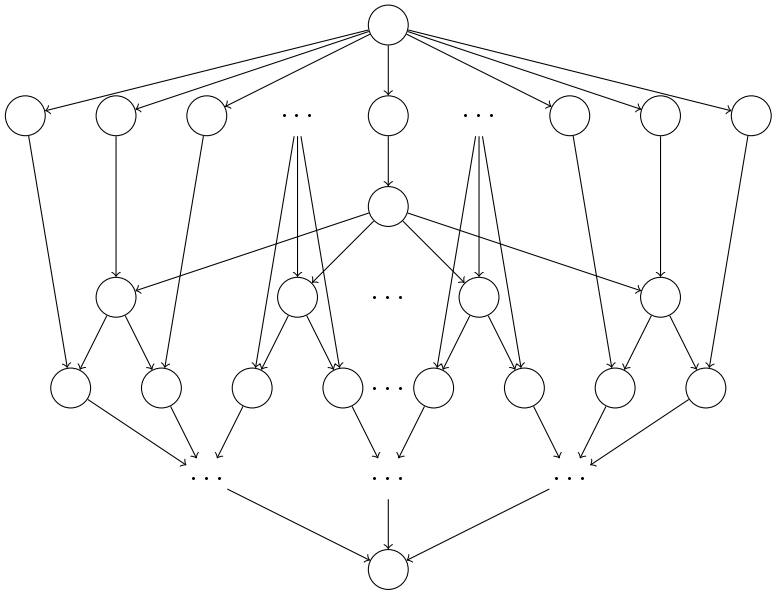


Figure 4.3: Example of an application for LU or Cholesky matrix decomposition.

The framework provides computation costs for all applications, but unfortunately only the average values for each job type are available. As a result, I assigned each job a value corresponding to this job's type.

4.1.2 Generated graphs

Evaluation with randomly generated graphs is a common way to compare algorithms in research [KA99a; Can+08; THW02]. In comparison to traced graph, this approach cannot show how an algorithm would behave in real world scenarios. But the experiments with randomly generated graphs allow to try out a greater variety of DAG structures, which contributes to the completeness of evaluation.

Within the scope of this master thesis, a task graph generator was developed. Its core idea was inspired by Kwok's random generator with optimal solutions [KA99a]. My generator does not guarantee to produce a schedule with a known optimal solution because I do not use optimal solutions in my evaluation.

First the DAG generator allocates chunks of equal range on each of N processors, so-called `timelines`. Next, each timeline is split into a random number of chunks of random size. The length of each chunk denotes the computation costs of a corresponding job. Afterwards, pairs of strictly ordered jobs are randomly connected.

The generator has several parameters, which influence the structure of the resulting DAG. Number of processors `PROCESSORS` determines the initial number of timelines. Number `TASKS` divided by the number of processors is used as an upper bound to generate number of chunks for each timeline. The number of dependencies in the graph is leveraged using parameter `FANOUT`. Parallelism level can be configured using parameter `MAXDIST`. Higher parameter values allow for higher parallelism of the DAG.

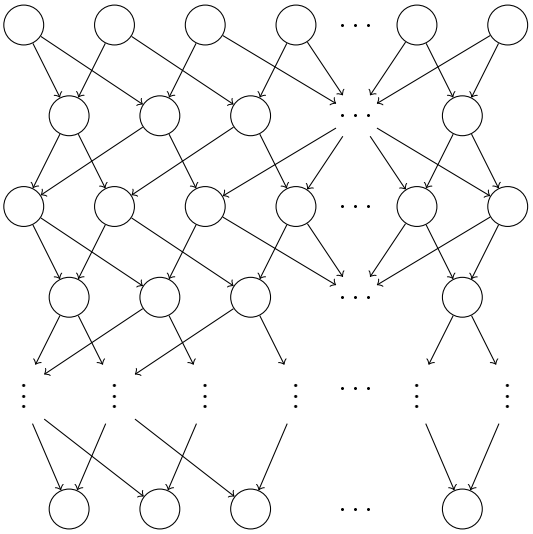


Figure 4.4: Structure of `stencil` application. Start and end nodes are skipped for the sake of brevity.

Name	PROCESSORS	TASKS	FANOUT	MAXDIST	Jobs
gen	4	3000	6	7	1147
gen wide	4	3000	6	15	2062
gen64	64	4000	6	7	2110

Table 4.1: Parameters used for DAG evaluation.

There were three different DAGs generated for the evaluation. Their parameters are given in Table 4.1. The actual number of jobs is given in the last column, because it is different from the value of the `TASKS` parameter.

4.1.3 Communication costs

Communication costs estimation is identical for traced and generated graphs. For both cases communication costs are generated. The main reason why it is done for traced graphs is an unavailability of traced communication costs. Although communication costs generation makes the evaluation less realistic, it is more convenient in certain cases. Generation of communication costs allows to easily test graphs with a different relation of communication costs to computation costs. This parameter is an important characteristic of a parallel program and it varies for different applications.

Communication costs are generated in a following way. First, an average computation cost of a job in a DAG is computed. This value is multiplied with communication to computation ration (CCR). The result is the communication cost for a single communication operation.

4.2 Performance evaluation

I start my evaluation by investigating how the horizon algorithm behaves depending on the value of the depth parameter. The experiment goes as follows. The simulator executes a given DAG using the Horizon scheduler with specific depth. As a performance metric I use the finish time of the end node, which is the length of critical path.

After influence of the depth of the horizon on the performance is studied, the Horizon algorithm is compared against other state of the art algorithms.

4.2.1 Horizon depth evaluation

The simulation is done for each application separately, varying the number of processors and the CCR value. Task assignment obligates simulator to be operable on single and multiple concurrently running applications. The simulator is capable of scheduling several DAGs together. Experiments show that co-scheduling allows for better performance than sequential scheduling in most cases. But such gain was expected and no other non-trivial results were noticed. For this reason experiments with co-scheduling are not presented within the scope of this master thesis.

Experimentation has shown that most of the DAGs exhaust parallelization capabilities with more than 32 processors. For this reason the simulation has been made for all the experiments with 1, 2, 4, 8, 16, 32 and 64 processors.

Three CCR values were taken: 0.1, 1, and 10. These values were chosen since they show symmetric distribution of communication and computation costs. An argument to use these particular values is their prior occurrences in the research [KA99a]. Additionally I was told in a private conversation that the measurements of physical systems graphs in the Modellica language done within the HPC-OM project [HPC] have shown CCR values to be in range from 0.002 to 12.

Not for all the experiments the same attention was paid within the scope of this master thesis, because their results are often similar to the results of other experiments. The summary of all experiments is given in Table 4.2. The table shows speedup of an application in comparison to the speed of sequential execution. For each combination of application and CCR, best and worst speedup are shown for 64 processors. Selected experiments are described further in this section.

First, DAG of a `mult` application was tested. This DAG has very simple structure because each job has at most one child. This DAG parallelizes very well up to 16 processors. With more processors speedup does not increase. The limit is reached because the DAG contains only 16 independent pipelines which cannot run in parallel on their own. CCR value does not change the situation, because there is no inter processor communication.

Application `lu` has a much more complicated structure, which makes scheduling a non-trivial task. Set up with low communication costs, the horizon algorithm shows speedup close to linear: speedup reaches up to 1.97 times for 2 processors and 3.8 for 4 processor. When number of processors increases, speedup grows slower. With maximal number of processors, which is 64, the best achieved speedup is 22.8.

DAG	CCR	*Depth*	Speedup	DAG	CCR	*Depth*	Speedup
`lu`	0.1	8	22.83	`gen`	0.1	8	10.7
		2	20.62			1,2,4	10.63
	1.0	8	18.06		1.0	8	10.01
		2	15.25			1	9.75
	10.0	2	4.53		10.0	8	20.03
		1	3.89			1	1.82
`heat`	0.1	8	5.17	`gen wide`	0.1	8	26.71
		1	5.09			1	25.99
	1.0	8	4.82		1.0	8	25.47
		1	4.69			1	21.93
	10.0	8	3.03		10.0	8	6.64
		2	2.69			2	6.19
`cholesky`	0.1	8	16.6	`stencil`	0.1	1	57.23
		2	15.31			2	54.45
	1.0	8	13.05		1.0	1	33.06
		2	11.4			4	29.42
	10.0	8	2.8		10.0	1	7.5
		4	2.6			8	7.21
`pipeline`	0.1	8	22.61	`gen64`	0.1	8	63.81
		2	22.47			2	56.39
	1.0	any[1]	17.08		1.0	8	59.75
	10.0	any	14.08			2	52.13
`mult`	0.1, 1, 10	any	16		10.0	8	11.52
						1	9.9

Table 4.2: Summary for horizon depth evaluation.

Considering the experiment with $CCR = 0.1$ (see Figure 4.5), with the higher number of processors difference between different depth levels becomes more visible. For example, if the number of processors is equal to 4, Horizon algorithm performs the worst with depth equal 2, showing 3.7 times speedup. With depth equal 8 and the same number of processors the speedup is 3.8. This results only in less than 3% performance gain with increasing depth from 1 to 8. But with 64 processors difference between best and worse schedule reaches more than 10% difference.

The tendency holds when communication and computation costs are equal (see Figure 4.5). With 4 processors best schedule is only 7% better than the worst one. Whereas with 64 processors difference reaches 18% percent.

The trend breaks with CCR equal 10. While with 64 processors difference between the best and the worst depth is close to the results of previous experiments (16%), low number of processors shows different picture. With 2 and 4 processors the schedule actually becomes slower than with 1. I call this effect communication burst problem.

[1] Depth of 1, 2, 4 and 8 were tested.

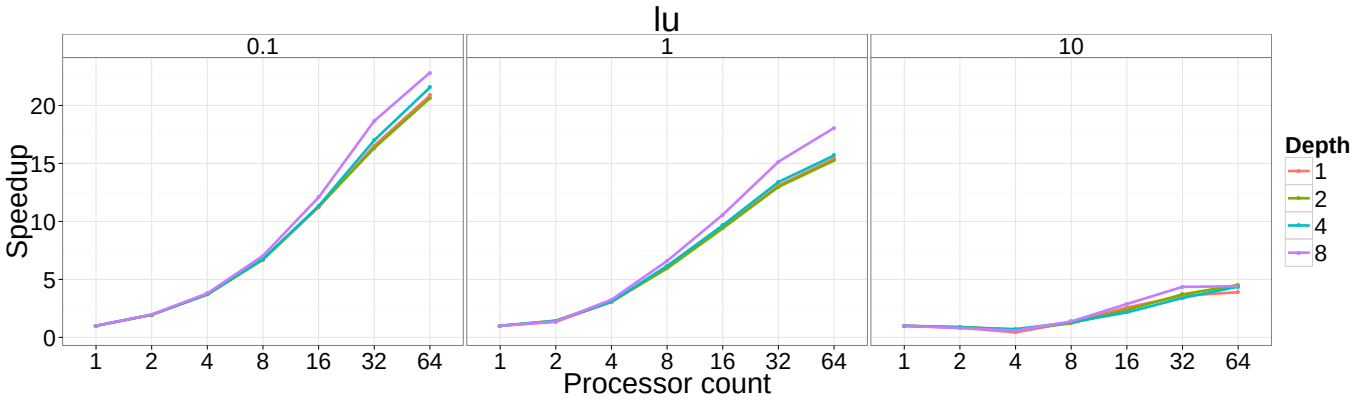


Figure 4.5: Horizon scheduler with application `lu`. Different graphs are for different CCR values.

It reappears in many subsequent experiments. It's detailed explanation is given in Section 4.4.1.

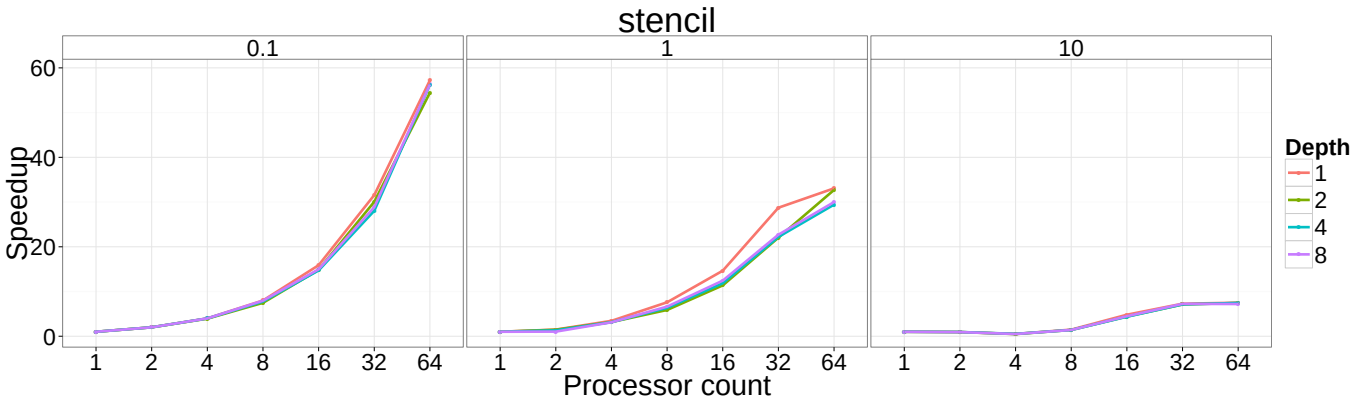


Figure 4.6: Horizon scheduler with application `stencil`. Different graphs are for different CCR values.

Other applications show very similar behavior. Among them extreme communication sensitivity show applications `stencil` and `gen64`. With $CCR = 0.1$ and 64 processors their speedup reaches 57.23 and 63.80 correspondingly. These applications have a lot of communication dependencies (16127 and 7231 correspondingly) and they are extremely sensitive to increase in communication costs. Such, when CCR grows up to ten, best possible speedup with 64 processors reaches only 11.52 times for `gen64` application and only 7.5 for `stencil` application.

If the DAG structure favors, increase in depth level allows to achieve the best performance. In the best case the DAG for `lu` application scheduled on 64 processors with CCR equal to 0.1 was 18.4% faster in the best case, than in the worst case. In

the absolute majority of the cases, including this particular one, the highest gain was achieved with depth equal to 8.

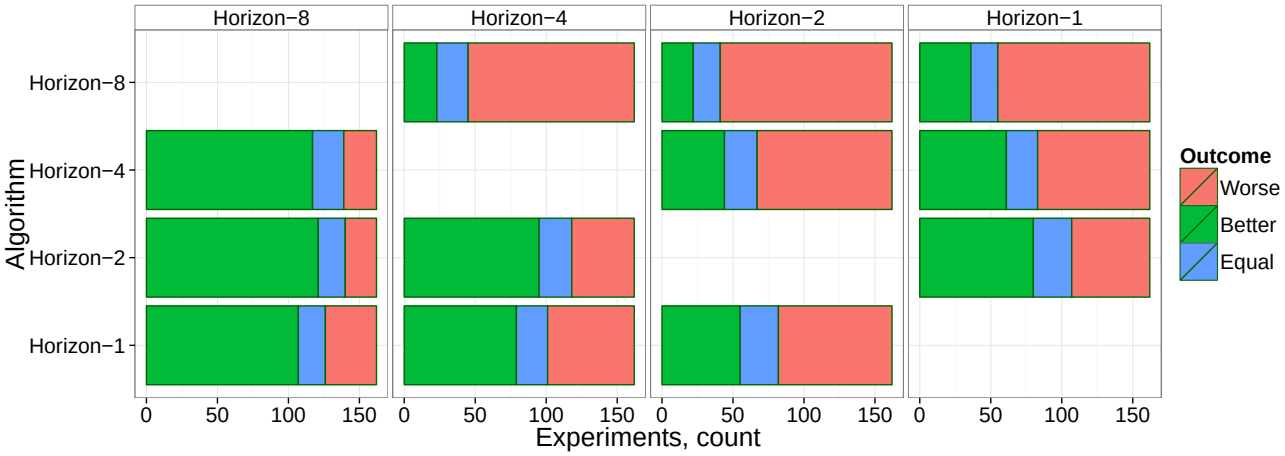


Figure 4.7: Aggregated comparison of Horizon with different depth.

Surprisingly, with 64 processors, Horizon with depth 2 had a smaller speedup more often, than horizon with depth 1. The reasons why this has happened and if it is necessary for `Horizon-2` to have higher performance, than `Horizon-1` are left for future work.

Horizon with depth 1 has a property knowing the same exact information as any just-in-time algorithm. Thus, although its implementation stays the same, it can be characterized as just-in-time algorithm.

The aggregated comparison results for all the experiments are shown in Figure 4.7. In the same experiment the schedule length produced by each algorithm is pairwise compared for each combination of the initial conditions. The initial conditions comprise application type, number of processors and CCR. If one algorithm produces shorter schedule than the second one, it is considered better in this particular experiment. In other words, the algorithms are ranked according to their schedule length, within the same experiment.

In the end the better algorithm has better schedules more frequently. For example, Figure 4.7 shows that in 117 experiments `Horizon-8` produced better schedule, than `Horizon-4`. On the other hand `Horizon-4` were better only in 23 experiments. In 22 experiments schedule lengths turned out to be equal. Experiments with 1 processor were omitted, because in this case schedule length always equals to total computation costs and does not depend on the scheduler.

4.2.2 Comparative analysis of the algorithms

Experiments in this section are done similarly to the experiments from the previous section. The only difference is that instead of running algorithms using the same scheduler, various schedulers are used. When I mention the Horizon algorithm in relation to other algorithms, I assume the Horizon algorithm with depth 8.

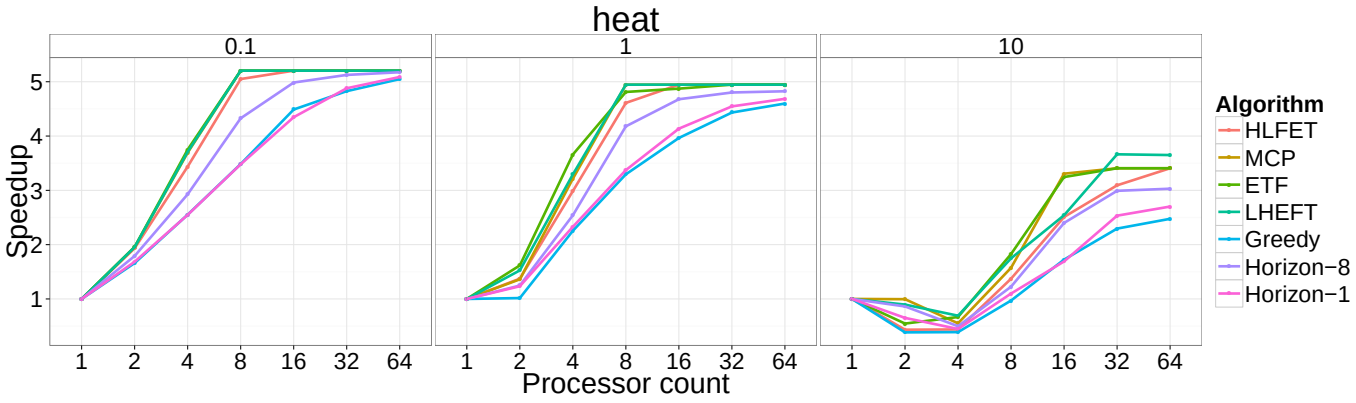


Figure 4.8: Scheduling `heat` application using different algorithms. Different graphs are for different CCR values.

Tendencies discovered for the Horizon scheduler are very similar for other schedulers as well. Speedup bound also turned out to be not only the property of the scheduler, but also it is the property of the application structure. Applications which allowed higher parallelism for the Horizon scheduler allowed higher parallelism for other schedulers as well.

Communication cost sensitivity is peculiar to all the algorithms. One of the most outstanding examples is the `heat` application, which is shown in Figure 4.8. Even when the CCR is low, the highest achievable speedup with 64 processors reached nearly five times. When CCR reaches 10 times the maximal speedup drops to 3.65 times.

As with the Horizon algorithm, some applications with high CCR run faster with single processor, than with 2 and 4 processors. When CCR equals to 1, slowdown, if exists, is observed mostly for 2 processor systems. But with CCR equal to 10, slowdown even with 4 processor becomes frequent.

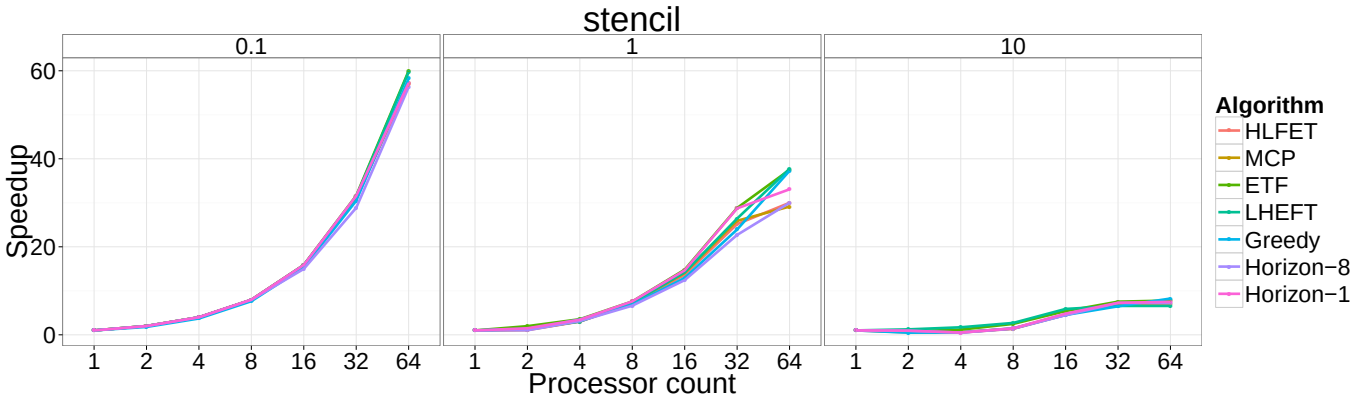


Figure 4.9: Scheduling `stencil` application using different algorithms. Different graphs are for different CCR values.

Depending on the application type, the Horizon algorithm is either slower than full-ahead algorithms, or it has comparable performance. Figure 4.8 shows an application when Horizon algorithm significantly slower in most of the situations with CCR equal to 0.1 or 1. But with CCR equal to 10 its performance degrades less than the performance of full-ahead algorithms.

Scheduling of such highly parallel application, like `stencil`, allows Horizon to show competitive performance in comparison to full-ahead algorithms. Figure 4.9 shows how the Horizon algorithm performance relates to the other algorithm. Although the Horizon algorithm maintains the trend of the other algorithms, it produces slightly longer schedules, when CCR is low. But when CCR is high, overall degradation blurs the difference completely.

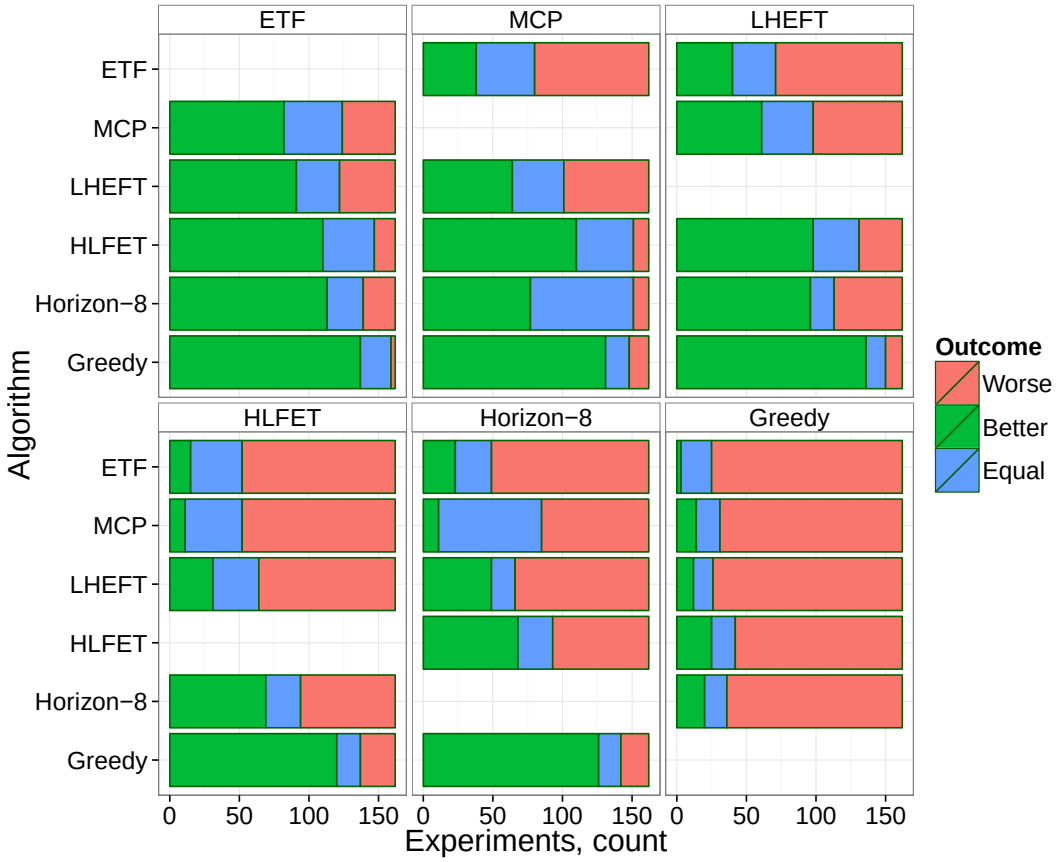


Figure 4.10: Aggregated comparison of the algorithms.

The aggregated comparison presented in Figure 4.10 shows that the best performing are the full-ahead ones. HLFET is the simplest full-ahead algorithm, which is studied in this master thesis. It turned out to be only slightly better than the Horizon algorithm: HLFET has produced shorter schedule than Horizon in 69 experiments and Horizon produced the shorter schedule in 68 experiments.

The main difference of the other three full-ahead algorithms is that they use insertion approach (see Section 2.6.1). I assume that this is the main factor which allowed ETF, MCP and LHEFT algorithms to outperform the HLFET.

The comparison of the MCP and Horizon algorithms gives relatively high number of experiments where these algorithms produced schedules with equal length. The reason for this phenomena can be the fact that both these algorithms use the ALAP metric to order the jobs in the ready queue. The difference is that MCP orders all jobs at once, whereas Horizon orders only the subset of the jobs. But the similarity in these algorithms still seems to bring the high number of equal results.

Although Figure 4.10 does not show how big the difference is in each concrete experiment, it still supports the aforementioned thought that the performance potential of the Horizon algorithm lies between full-ahead algorithms and just-in-time algorithms.

4.3 Robustness evaluation

In this series of experiments I study the impact of imprecise cost estimation upon the overall performance of the algorithm. To simulate such a situation I change the communication and computation costs of the jobs when I set up jobs for execution. I expect full-ahead schedulers to show higher degradation because they are doomed to work with wrong cost estimations without any chance to correct their mistake.

On the other hand, a just-in-time algorithm should be able to adapt to wrong information better, because it is able to learn the actual computation and computation costs of a job, when it finishes its execution. The Horizon scheduler should be somewhere in between, because it is able to learn actual cost information, but it cannot change job-to-processor assignment once the decision is taken.

The actual cost value is generated based on cost estimations, which are known in advance. The degree of imprecision of cost estimations is leveraged by coefficient of variation (CV). The coefficient of variation is a standardized measure of dispersion. It is defined as follows.

$$CV = \frac{\sigma}{\mu},$$

where σ is standard deviation, μ is the mean. Coefficient of variation is a parameter of the experiment. The mean value is set to be the value of the original cost estimation. The standard deviation is calculated as $CV \cdot \mu$.

Actual cost values are generated using Gaussian distribution function before the scheduler invocation, so that all the schedulers schedule identical graphs with the same job costs for corresponding jobs. Since Gaussian distribution can generate very high and very low numbers, the values, which do not fit in the range $[0.01 \cdot \mu; 1.99 \cdot \mu]$, are denied and the cost generation is attempted again.

The robustness of the algorithm is the higher, the less the resulting schedule slows down in the presence of imprecise cost estimations. Figure 4.11 shows how schedule length changed after imprecise cost estimations had been introduced. For each experiment the bars show if the scheduler managed to generate faster or slower schedule after execution

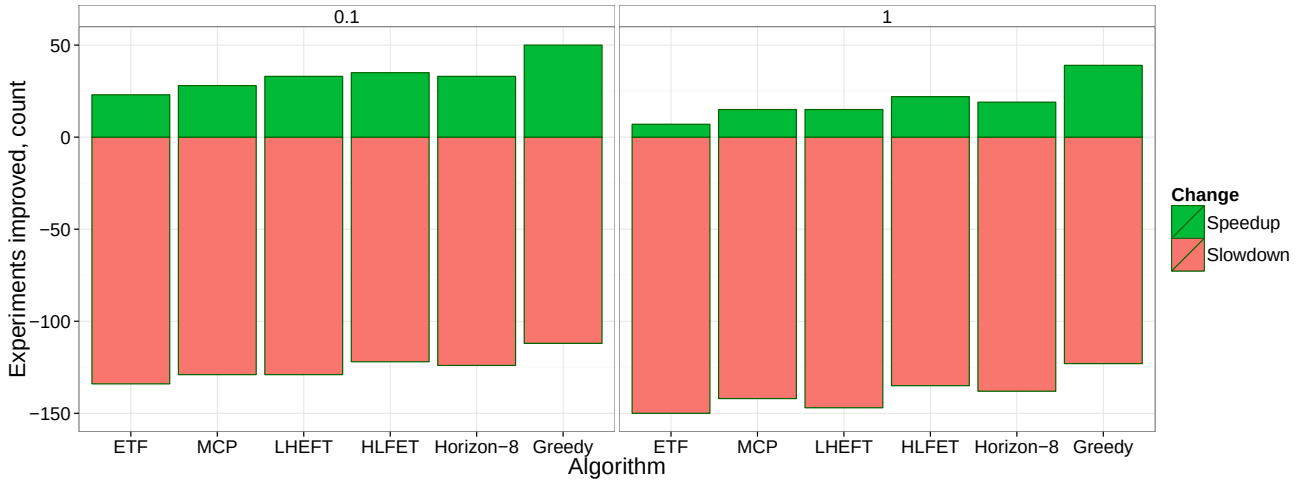


Figure 4.11: Change in rank among all schedulers in presence of imprecise cost estimation. Different graphs show change for different coefficient of variation.

experienced cost fluctuations. Although mean value of a job cost stays the same, all the schedules become slower in most of the cases. The number of slowdowns increased further with increased coefficient of variation.

Still, some of the algorithms suffer less than the others. It is clearly visible, that Greedy algorithm manages to achieve more speedups than others. And it slows down more seldom than others as well. This effect was expected, as it was mentioned in the beginning of the section.

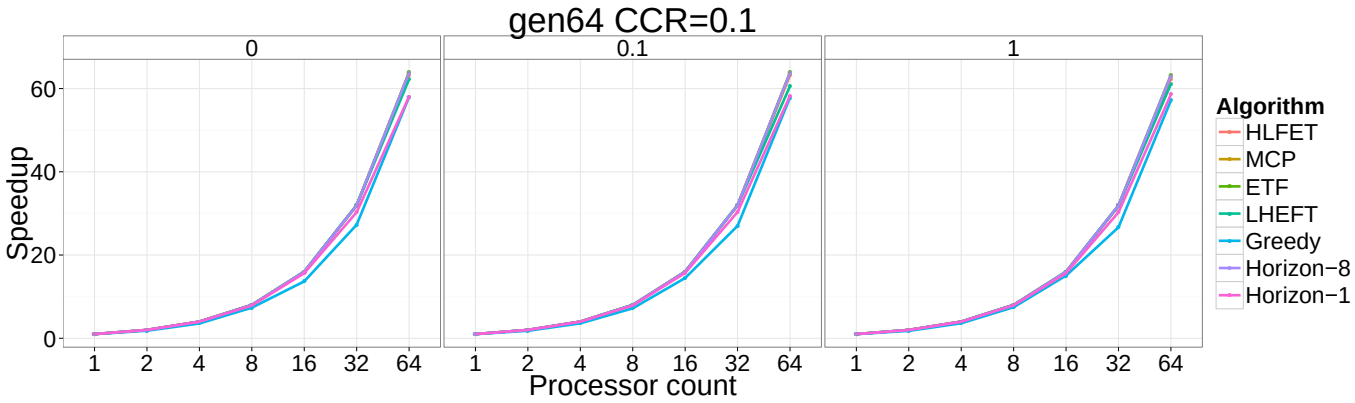


Figure 4.12: Application `gen64` is tolerant to inaccurate cost estimations.

At the same time, performance degradation in presence of imprecise cost estimations turns out to be much smaller, if communication costs are low. But still some graph structures are more prone to suffer from wrong estimations than others. Figure 4.12 shows how application `gen64` reacts on imprecise cost estimations. With increase in coefficient of variation overall performance of all algorithm does not significantly change.

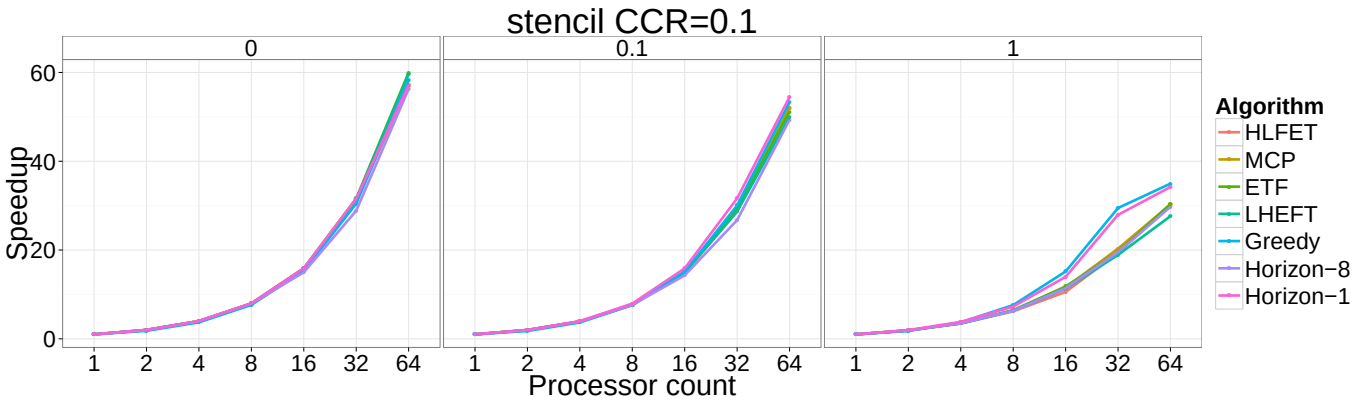


Figure 4.13: Application `stencil` is sensitive to inaccurate cost estimations.

Another example is application `stencil` (see Figure 4.13). It scales very well with higher number of processors, but when cost estimations become inaccurate the maximal speedup drops almost twice. The explanation again is the DAG structure. Application `gen64` represents randomly generated graph without any definite structure. Changes in communication and computation do not break any patterns which are required for high speedup.

On the other hand the DAG structure of `stencil` application is very regular. It consists of many stages of execution. Each next stage depends on many jobs from the previous stage. Even if very few nodes from the previous stage are delayed, the whole next stage is delayed as well. The more stages such an application has, the more delays are accumulated. In the result, overall progress of execution stalls.

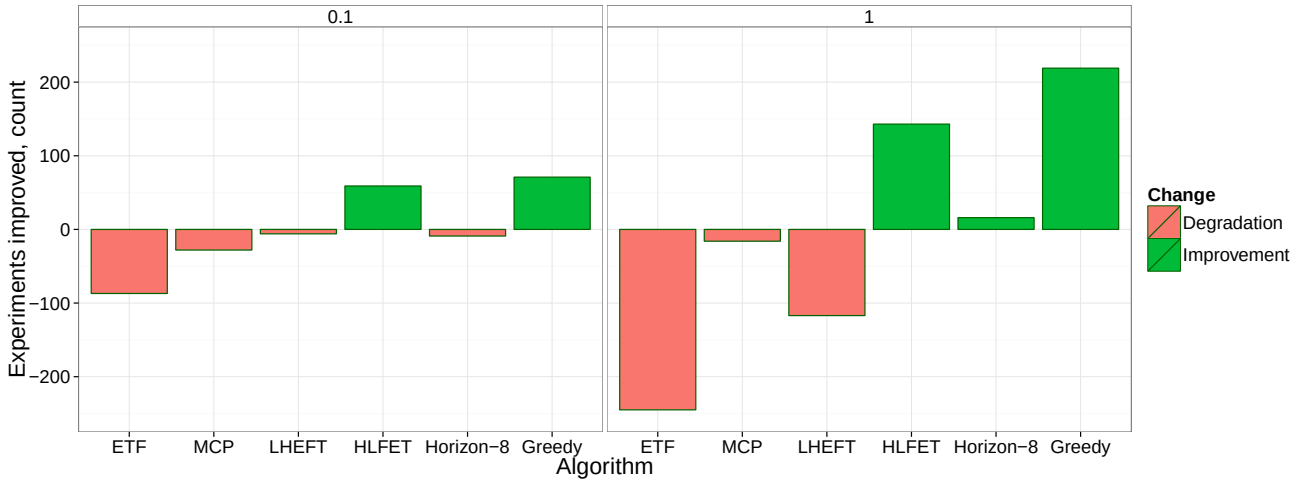


Figure 4.14: Change in rank among all schedulers in presence of imprecise cost estimation. Different graphs show change for different coefficient of variation.

Different algorithms have different sensitivity towards an increase in cost estimation uncertainties. Figure 4.14 shows how algorithm ranking change with different coefficient of variation. Improvement bar means an improvement in the algorithm ranking in the major part of the experiments. On the contrary, degradation bar means deterioration in the algorithm ranking dominated in the major part of the experiments.

No surprise, that three full-ahead algorithms lost their positions, showing bigger sensitivity towards cost estimation accuracy. It is also straightforward, why Greedy algorithm has shown the biggest relative improvement.

Horizon algorithm has shown only minor changes in its ranks. With coefficient of variation equal to 0.1 Horizon algorithm even slightly degraded. It improved its rank in total in 6 experiments, but degraded in total in 15 experiments. In 11 out of 15 experiments, where Horizon degraded, the position was lost to the Greedy algorithm.

The increase of coefficient of variation up to 1.0 made Horizon lose its rank to the Greedy algorithm in more experiments. But even bigger degradation of full-ahead algorithms allowed to maintain overall improvement.

Algorithm HLFET degraded much less, than other full-ahead algorithms which allowed it to significantly improve its relative positions. The reasons why this could happen I discuss in detail in Section 4.4.2.

Figure 4.15 shows pairwise comparison of the algorithms when coefficient of variation equal to 1.0. Greedy algorithm stayed the worst performing algorithm despite of being the most robust. Also number of the experiments where different schedulers produced schedules with the same length significantly decreased.

4.4 Results interpretation

Some of the results and patterns discovered during experimentation require additional explanation. These results are: high level of coincidence in schedule lengths produced by MCP and Horizon algorithms; performance degradation with bigger number of processor; high robustness of HLFET algorithm.

I proposed a short reasoning to explain frequent equality of MCP and Horizon schedulers. Further investigation of this phenomena requires additional experimentation and is left for future work.

Other two patterns also should be supported by exhaustive evaluation. But in the scope of this master thesis I propose only some argumentation, which explains why these patterns occur. This reasoning has to be supported by experiments, but this part is also left for future work.

4.4.1 Communication burst problem

To understand the communication burst problem, consider a following example. The goal is to schedule a very simple DAG from Figure 4.16. For the sake of brevity assume that the scheduler takes ready jobs in random order and schedules them on the processor which allows earliest start time.

The first job to be scheduled is job v_0. Because the schedule is empty yet, the job can be assigned to any processor. Choose processor p_0. Next job is v_1. Processor p_0 allows

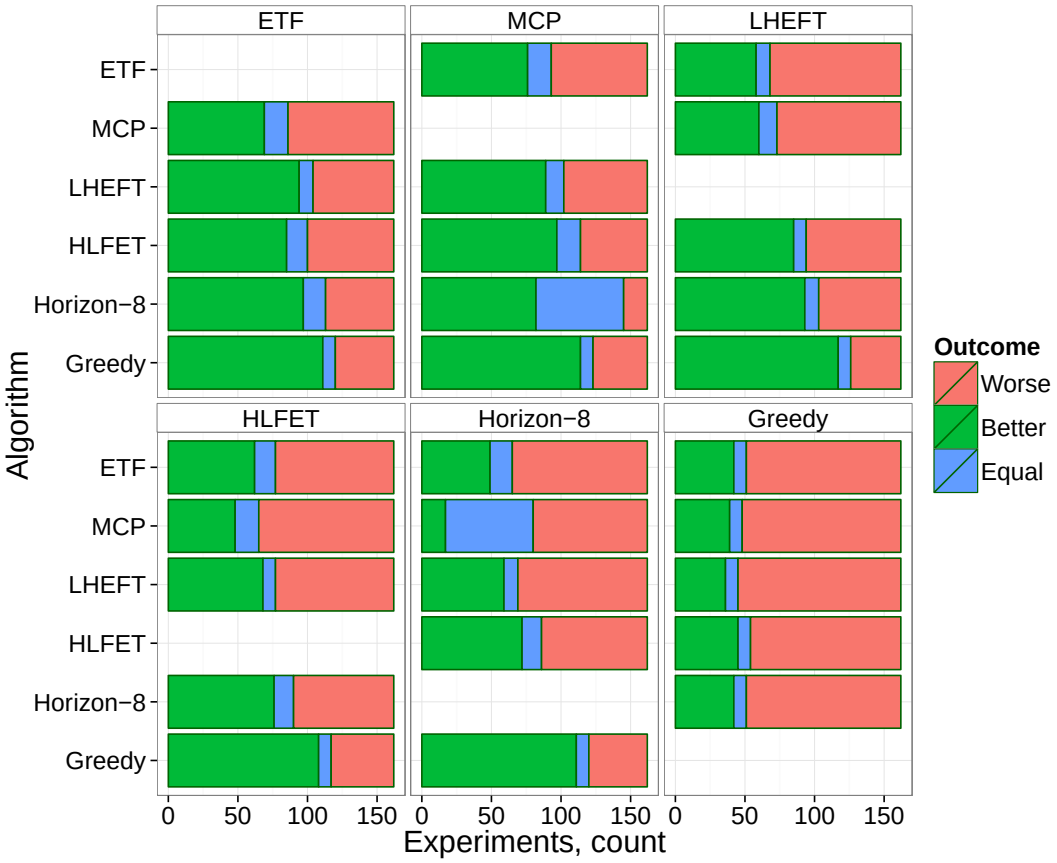


Figure 4.15: Aggregated comparison of the algorithms with $CV = 1.0$.

earliest start time $w(v_0)$. Processor p_1 allows earliest start time 0. Thus, processor p_1 is chosen.

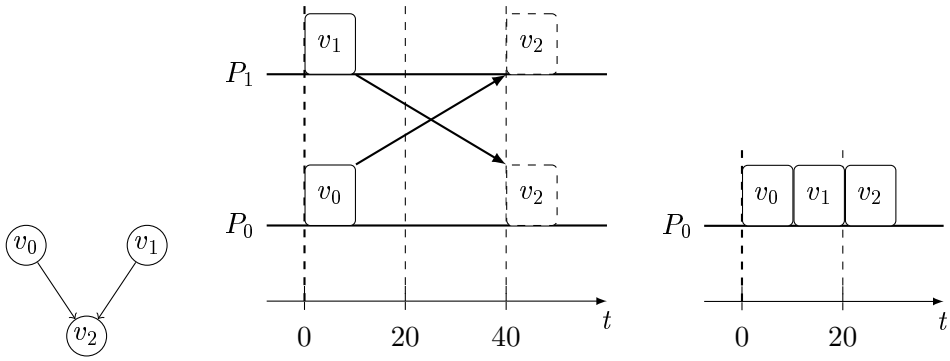
Now, the last job v_2 is to be scheduled. This job depends on v_0 and v_1. If job v_2 is scheduled to the processor p_0, EST for v_2 is equal to $max(w(v_0), w(v_1) + w(e_{12}))$. On the other hand if job v_2 is scheduled to p_1, EST for this job is equal to $max(w(v_1), w(v_0) + w(e_{02}))$.

If scheduling is done only with one processor, EST for v_2 is equal to $w(v_0) + w(v_1)$. If communication costs are low, they can be ignored. In such situation it is easy to achieve speed up, because

$$\frac{w(v_0) + w(v_1)}{max(w(v_0), w(v_1))} > 1.$$

But if communication costs get higher and cannot be ignored anymore, the relation

$$\frac{w(v_0) + w(v_1)}{min(max(w(v_0) + w(e_{02}), w(v_1)), max(w(v_1) + w(e_{12}), w(v_0)))}$$



(a) Example DAG. (b) Schedule with burst. (c) Sequential schedule.

Figure 4.16: Communication burst problem.

is not guaranteed to be bigger than 1. More sophisticated algorithms could require more complex graph structures to run into the same problem, but the pattern stays the same. The bigger the fraction of communication costs the more it is expensive to put jobs, which share descendants, on the different processors. In other words, locality matters. This effect does not always lead to overall performance degradation. Still with high number of processors there is a speedup. But it is visible from `lu` example already, that the higher the CCR, the lower maximal speedup.

4.4.2 Sparse schedule robustness

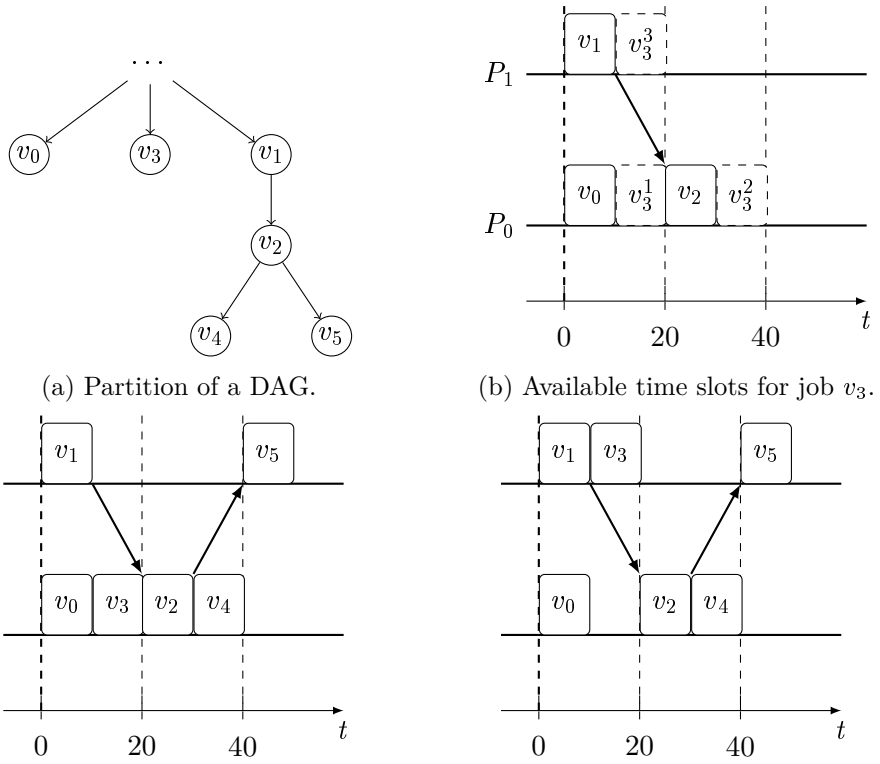
Evaluation has shown that not all full-ahead algorithms are equally sensitive to high accuracy of cost estimations. The algorithm ETF has shown the biggest degradation in comparison to other algorithms, which was superior to other algorithms when coefficient of variation was zero.

On the other hand, the algorithm HLFET manages to sustain this adverse factor better than others. As it was mentioned before, the main distinguishing mark of this algorithm, is that it does not use insertion approach. This approach is an heuristic, which allows to build denser schedules, which results in faster execution in the end.

But the evaluation hints that this heuristic could hurt the robustness of the algorithm. Following is the reasoning, why I think that the insertion approach decreases the robustness of the algorithm. It is more a motivation to investigate this dependency further, rather than formal proof of why HLFET is more robust than ETF.

I illustrate my thought with the following example. Consider DAG segment presented in Figure 4.17a. It is scheduled on two processors and jobs v_0, v_1 and v_2 are already scheduled. Next job to be scheduled is job v_3. Possible time slots are shown as dashed rectangles in Figure 4.17b. If the scheduler uses the insertion approach, it can put job v_3 to slots v_3^1, v_3^2 and v_3^3.

If the insertion approach is not used, the scheduler attempts to allocate the time slot only in the end of each processor's time line. This allows the non-inserting scheduler to schedule job v_3 only to time slots v_3^2 and v_3^3.



(a) Partition of a DAG.

(b) Available time slots for job v_3.

(c) Scheduling with insertion approach.

(d) Scheduling without insertion approach.

Figure 4.17: Robustness of dense and sparse schedules.

Now consider what happens when the inserting scheduler picks, for example, slot v_3^1 and proceeds. Figure 4.17c shows a possible situation after jobs v_4 and v_5 are scheduled. When the application starts execution and it turns out that job v_0 runs a bit longer than it was expected, jobs v_2, v_3, v_4 and v_5 are delayed as well.

Assume a non-inserting scheduler chooses to assign job v_3 to slot v_3^3. Figure 4.17d shows a possible schedule state after the scheduler also schedules jobs v_4 and v_5. Now, if job v_0 executes a bit longer, than it was expected, other jobs are not delayed by v_0.

Filling the holes is good for performance, because it allows to schedule subsequent jobs to an earlier time. But this approach has the higher probability to trigger a chain reaction, where a delay of one job delays many other subsequent jobs. In this sense using an insertion approach requires a tradeoff between higher robustness and faster execution.

In the end even with coefficient of variation equal to 1.0 inserting schedulers (ETF, MCP, LHEFT) turned out to be faster, than non-inserting one (HLFET).

4.5 Conclusion

The goal of this section was to show how the Horizon algorithm, and the Horizon scheduling model in general, relates to other known models. Being an intermediate model, the Horizon model proposes a tradeoff between full-ahead and just-in-time models.

Performance is the first characteristic which was measured. The evaluation has shown that the Horizon scheduler is able to provide the performance that is mostly lower than performance of full-ahead algorithms. On the other hand the Horizon scheduler turned out to show better performance, than the just-in-time algorithm.

The Horizon algorithm can tune its characteristics by changing the depth parameter. This parameter determines how many jobs the scheduler takes into consideration before making a decision. Evaluation has shown that an increase of this parameter tends to improve performance, but small change of this parameters can bring very small improvement, if any. For example, Horizon with depth 1 turned out to be more often better than Horizon with depth 2.

Applications have shown different propensity for parallelization, which for the most applications decreased rapidly with the increase in the fraction of communication costs. Application appeared to be so much sensitive to communication costs, that sometimes adding additional processors lead to performance slowdown. This phenomena was identified as a communication burst problem and described in Section 4.4.1.

The communication burst problem uncovers somewhat counter-intuitive results, because, according to Amdahl's law, the major limiting factor for application parallelization is a high fraction of sequential code. But within macro dataflow model it turns out that a bigger limiting factor are high communication costs. This factor turns out to be very significant, even in the system with the network model of fully connected processors.

Another aspect, which was the subject for comparison is an ability of a scheduling algorithm to produce a schedule, which is tolerant to inaccurate cost estimations. Such an ability is called robustness.

Just-in-time and Horizon algorithms have a possibility to actualize costs of finished jobs, because these algorithms run in parallel to an application execution. Full-ahead algorithms plan everything ahead, and thus have to rely on inaccurate estimations. Because of this difference full-ahead algorithms turned out to be less robust than just-in-time and horizon ones.

As an exception, full-ahead HLFET algorithm has shown second highest robustness among all considered algorithms. This algorithm was the slowest full-ahead algorithm and even high robustness did not allow HLFET to outperform other full-ahead algorithms. In Section 4.4.2 I made an attempt to explain the nature of high robustness of this algorithm.

5 Conclusion And Outlook

In the scope of this master thesis I have described the Horizon scheduler which is built as a combination of full-ahead and just-in-time algorithms. The evaluation has proven the new scheduling model indeed posses intermediate characteristics among state of the art algorithms.

In this section I make concluding remarks, mention discussion points about shortcomings of my work and sketch out new goals towards further research of the Horizon model.

The work which was done in the scope of this master thesis includes description of a new scheduling model, development of a new scheduler, and comparative evaluation of a new scheduler together with several state of the art algorithms.

The evaluation studied the algorithms with regard to their performance and robustness. These two qualities turned out to be reversely proportional to each other. The Horizon algorithm turned out to strike a balance in both these characteristics.

I do not consider this result as outstanding, but my expectation that this model is beneficial for applications which cannot utilize the better performing full-ahead model. At the same time a horizon algorithm is generally faster than a just-in-time one.

In this master thesis I propose a scheduling algorithm which is based on the MCP algorithm. And it turned out that such relation lead to surprisingly high number of equally long schedules produced by MCP and the Horizon algorithms with the same experiment. Such significant dynamic is unlikely to be just a coincidence, but still requires stronger proof to be considered as a fact. It is even more interesting if other full-ahead algorithms are able to succeed their characteristic to their horizon implementations.

Full-ahead algorithms are able to work in parallel with program execution and create a schedule on the fly. Unfortunately such implementation of these algorithms was omitted, but still it is required to make reasonable conclusions about Horizon algorithms.

The communication burst problem revealed an important issue which shows that communication costs can heavily limit overall performance. This phenomena was discovered with a very simplistic network model, but it is also expected to be found in more realistic environments. I simulated an environment which distinguishes only two types of communication: local and interprocessor. Local communication is free of charge and has no delays. Interprocessor communication is expensive and can delay an execution of subsequent jobs.

Many modern systems have a hierarchical topology. Communication costs between processors within such topology vary in order of magnitude depending on a concrete placement of this processors. Communication within the same socket is considered to be very fast because processors from the same socket share local memory. Often such communication is considered to be local. The next level incorporates processor sharing the same main board, but residing on different sockets.

As an example of hierarchical system, consider cluster used for Google file system [GGL03]. The finest grain of composition of such system called *machine*. Machine consists of a main board with, possibly, several processors. Several machines are put together in the same *rack*. Racks are connected between each other and build up a *cluster*. Communication within a rack experiences smaller delays than communication between clusters. General observation is called *locality principle* and is formulated as follows: the closer processors are to each other, the more resources they share and the faster the communication is.

Future work on Horizon model could include how a locality principle and communication burst problem affect the performance of Horizon schedulers with more sophisticated network architectures. This can be combined with adding heterogeneity to processor and network links. These changes to the system model may enable the Horizon scheduler for a wider area of application and make comparison with Lookahead HEFT [BSM10] algorithm more sensible.

System model, which I use within the scope of my master thesis models network contention in communication. This differs from other research [KA99b], but provides more realistic picture.

Robustness evaluation revealed how different algorithms tolerate fluctuations in computation and communication costs. Insertion approach has been identified as an instrument to trade off performance for robustness. But truly robustness oriented algorithms have not been evaluated.

As another topic for research of the Horizon algorithm, it can be studied in regard to its fault tolerance. The absence of full-ahead planning suggests that an implementation of fault tolerance techniques should be easier.

The Horizon algorithm complexity analysis has shown that this algorithm has much higher asymptotic complexity than both just-in-time and full-ahead algorithms. This makes it impractical to use for large scale systems. I think that decreasing asymptotic complexity is the most important direction of future work.

Luckily, locality principle hints that distant communication is very expensive and can be moved out of scope, when an algorithm makes a scheduling decision. In such incarnation Horizon algorithm gives up the globally visible horizon and maintains the local horizons for each processor separately. A local horizon should include not only the successors of the jobs which run locally, but also the successors of the jobs which run on neighboring processors. This approach should transform the Horizon scheduler to something similar to what gossip protocol based load balancers do.

Even if the total complexity is not getting smaller, the complexity burden is to be distributed among all processors in the system. With this model each processor faces the complexity for making a scheduling decision which is limited by the depth of lookahead in the DAG, the number of processors which share scope of their horizon and average fan out of a job. If the DAG is sparsely connected, which is mostly the case, the Horizon scheduler can be able to schedule arbitrary size DAGs and maintain performance better than performance of just-in-time algorithms.

Another promising research direction is a simultaneous scheduling of several applications (co-scheduling). Although no interesting results were discovered in this work, I think there is an opportunity to utilize properties of horizon model in co-scheduling. In

its essence co-scheduling is identical to scheduling of a single DAG, which is built out of several segments, where each segment represents separate application.

In the simplest scenario all applications start simultaneously. A full-ahead scheduler can create a schedule for the compound DAG as if it was a single application. But it can become problematic to create all schedules for all possible combinations of applications. The solution is to create compound schedule on the fly when set of applications is known, but initial delay to start the execution can be unacceptable.

If applications are allowed to start suddenly and at an arbitrary time, I see no solution how a full-ahead scheduler can deal with this situation, except recreating the scheduler from scratch for the DAG which consists of pending part of already running applications and the newly appearing application. However, a Horizon scheduler, as well as a just-in-time scheduler, are able to deal with such situation without any additional delays.

Both aforementioned scenarios can be combined together into one system with distributed Horizon scheduler, which is able to schedule several applications at once. Such a system can be considered as an improvement over just-in-time schedulers because additionally to all the knowledge which is available for just-in-time algorithm, the Horizon scheduler can benefit from knowing an application structure.

Unfortunately, this model has been examined only in a simulated environment and there is no strong proof that this model is applicable. But model evaluation done through simulation, suggests that there is some place for optimism.

Appendix A

DAG generator

```
#!/usr/bin/python

import random
import operator
from graphviz import Digraph

LENGTH=1000000
PROCESSORS=64
TASKS=4000
FANOUT=6
MAXDIST=7

Timeline = [[0.0, LENGTH] for i in range(PROCESSORS)]

class Task:
    def __init__(self, cpu, (start, end)):
        self.start = start
        self.end = end
        self.cpu = cpu
        self.next = -1

    def __str__(self):
        return repr(self)

    def __repr__(self):
        return "CPU: %d [%f;%f]" % (self.cpu, self.start, self.
    end) + \
               (" n: %d" % self.next if self.next != -1 else "")

all_tasks = []

for p in range(PROCESSORS):
    for t in range(int(round(random.uniform(1, TASKS)/
    PROCESSORS))):
        number = 0.
```

```
34          while number == 0.0:
35              number = random.uniform(0, LENGTH)
36          Timeline[p].append(number)
37      Timeline[p].sort()
38      for interval in zip(Timeline[p][:-1], Timeline[p][1:]):
39          all_tasks.append(Task(p, interval))
40
41  dot = Digraph(format='png')
42
43  for i in range(len(all_tasks)):
44      node = all_tasks[i]
45      dot.node("gen.%d" % i, size=str(node.end - node.start))
46
47  deps = set()
48  for i in range(len(all_tasks)):
49      node = all_tasks[i]
50      for _ in range(random.randint(1, FANOUT)):
51          retry = True
52          while retry:
53              edge = random.randint(0, len(all_tasks) - 1)
54              if (all_tasks[edge].start >= node.end) or (node.
    start >= all_tasks[edge].end):
55                  if (min(abs(all_tasks[edge].start - node.end),
56                          abs(node.start - all_tasks[edge].end))
    > LENGTH / MAXDIST):
57                      continue
58                  retry = False
59          deps.add((min(i, edge), max(i, edge)))
60
61  for (i, edge) in deps:
62      if all_tasks[i].end <= all_tasks[edge].start:
63          dot.edge("gen.%d" % i, "gen.%d" % edge, size='1')
64      elif all_tasks[edge].end <= all_tasks[i].start:
65          dot.edge("gen.%d" % edge, "gen.%d" % i, size='1')
66
67  dot.render()
```

Bibliography

[AB14] Hamid Arabnejad and Jorge G Barbosa. „List scheduling algorithm for heterogeneous systems by an optimistic cost table.“ In: *Parallel and Distributed Systems, IEEE Transactions on* 25.3 (2014), pp. 682–694.

[ABB00] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. „The Data Locality of Work Stealing.“ In: *Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA '00. Bar Harbor, Maine, USA: ACM, 2000, pp. 1–12. ISBN: 1-58113-185-2. DOI: 10.1145/341800.341801. URL: http://doi.acm.org/10.1145/341800.341801.

[ABP98] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. „Thread Scheduling for Multiprogrammed Multiprocessors.“ In: *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA '98. Puerto Vallarta, Mexico: ACM, 1998, pp. 119–129. ISBN: 0-89791-989-0. DOI: 10.1145/277651.277678. URL: http://doi.acm.org/10.1145/277651.277678.

[ACD74] Thomas L. Adam, K. M. Chandy, and J. R. Dickson. „A Comparison of List Schedules for Parallel Processing Systems.“ In: *Commun. ACM* 17.12 (Dec. 1974), pp. 685–690. ISSN: 0001-0782. DOI: 10.1145/361604.361619. URL: http://doi.acm.org/10.1145/361604.361619.

[AK00] Theodore Andronikos and Nectarios Koziris. „Optimal scheduling for UET-UCT grids into fixed number of processors.“ In: *Parallel and Distributed Processing, 2000. Proceedings. 8th Euromicro Workshop on*. IEEE. 2000, pp. 237–243.

[AK94] Ishfaq Ahmad and Yu-Kwong Kwok. „A new approach to scheduling parallel programs using task duplication.“ In: *Parallel Processing, 1994. ICPP 1994 Volume 2. International Conference on*. Vol. 2. IEEE. 1994, pp. 47–51.

[AKN05] M. Aggarwal, R.D. Kent, and A. Ngom. „Genetic algorithm based scheduler for computational grids.“ In: *High Performance Computing Systems and Applications, 2005. HPCS 2005. 19th International Symposium on*. May 2005, pp. 209–215. DOI: 10.1109/HPCS.2005.27.

[ALL89] Thomas E Anderson, Edward D Lazowska, and Henry M Levy. „The performance implications of thread management alternatives for shared-memory multiprocessors.“ In: *Computers, IEEE Transactions on* 38.12 (1989), pp. 1631–1644.

[App09] Apple. *Grand Central Dispatch Technology Brief*. 2009. URL: http://www.opensource.mlba-team.de/xdispatch/GrandCentral_TB_brief_20090608.pdf (visited on May 17, 2015).

[Aug+11] Cédric Augonnet et al. „StarPU: a unified platform for task scheduling on heterogeneous multicore architectures.“ In: *Concurrency and Computation: Practice and Experience* 23.2 (2011), pp. 187–198.

[BGM99] Guy E. Blelloch, Phillip B. Gibbons, and Yossi Matias. „Provably Efficient Scheduling for Languages with Fine-grained Parallelism.“ In: *J. ACM* 46.2 (Mar. 1999), pp. 281–321. ISSN: 0004-5411. DOI: 10.1145/301970.301974. URL: http://doi.acm.org/10.1145/301970.301974.

[BL99] Robert D. Blumofe and Charles E. Leiserson. „Scheduling Multithreaded Computations by Work Stealing.“ In: *J. ACM* 46.5 (Sept. 1999), pp. 720–748. ISSN: 0004-5411. DOI: 10.1145/324133.324234. URL: http://doi.acm.org/10.1145/324133.324234.

[Blu+95] Robert D. Blumofe et al. „Cilk: An Efficient Multithreaded Runtime System.“ In: *SIGPLAN Not.* 30.8 (Aug. 1995), pp. 207–216. ISSN: 0362-1340. DOI: 10.1145/209937.209958. URL: http://doi.acm.org/10.1145/209937.209958.

[Blu+96] Robert D. Blumofe et al. „An Analysis of Dag-consistent Distributed Shared-memory Algorithms.“ In: *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA '96. Padua, Italy: ACM, 1996, pp. 297–308. ISBN: 0-89791-809-6. DOI: 10.1145/237502.237574. URL: http://doi.acm.org/10.1145/237502.237574.

[Bra+01] Tracy D Braun et al. „A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems.“ In: *Journal of Parallel and Distributed computing* 61.6 (2001), pp. 810–837.

[BSM10] Luiz F Bittencourt, Rizos Sakellariou, and Edmundo RM Madeira. „Dag scheduling using a lookahead variant of the heterogeneous earliest finish time algorithm.“ In: *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*. IEEE. 2010, pp. 27–34.

[Can+08] Louis-Claude Canon et al. „Comparative evaluation of the robustness of dag scheduling heuristics.“ In: *Grid Computing*. Springer. 2008, pp. 73–84.

[CG72] Jr. Coffman E.G. and R.L. Graham. „Optimal scheduling for two-processor systems.“ English. In: *Acta Informatica* 1.3 (1972), pp. 200–213. ISSN: 0001-5903. DOI: 10.1007/BF00288685. URL: http://dx.doi.org/10.1007/BF00288685.

[Cha+05] Philippe Charles et al. „X10: an object-oriented approach to non-uniform cluster computing.“ In: *Acm Sigplan Notices* 40.10 (2005), pp. 519–538.

[CJ07] Louis-Claude Canon and Emmanuel Jeannot. „A comparison of robustness metrics for scheduling dags on heterogeneous systems.“ In: *Cluster Computing, 2007 IEEE International Conference on*. IEEE. 2007, pp. 558–567.

[CS] Hugues Cassé and Pascal Sainrat. „OTAWA, a framework for experimenting WCET computations.“ In:

[CSM93] H Chen, B Shirazi, and J Marquis. „Performance evaluation of a novel scheduling method: Linear clustering with task duplication.“ In: *Proceedings of the 2nd International Conference on Parallel and Distributed Systems*. 1993, pp. 270–275.

[DG08] Jeffrey Dean and Sanjay Ghemawat. „MapReduce: simplified data processing on large clusters.“ In: *Communications of the ACM* 51.1 (2008), pp. 107–113.

[DGH91] Helen Davis, Stephen R Goldschmidt, and John L Hennessy. „Multiprocessor Simulation and Tracing Using Tango.“ In: *ICPP (2)*. 1991, pp. 99–107.

[FBB08] Kurt B Ferreira, Patrick Bridges, and Ron Brightwell. „Characterizing application sensitivity to OS interference using kernel-level noise injection.“ In: *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press. 2008, p. 19.

[Fer+01] Christian Ferdinand et al. „Reliable and precise WCET determination for a real-life processor.“ In: *Embedded Software*. Springer. 2001, pp. 469–485.

[Fin+96] Lucian Finta et al. „Scheduling UET-UCT series-parallel graphs on two processors.“ In: *Theoretical Computer Science* 162.2 (1996), pp. 323–340.

[FL12] Geoffrey Falzon and Maozhen Li. „Enhancing list scheduling heuristics for dependent job scheduling in grid computing environments.“ In: *The Journal of Supercomputing* 59.1 (2012), pp. 104–130.

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. „The Google file system.“ In: *ACM SIGOPS operating systems review*. Vol. 37. 5. ACM. 2003, pp. 29–43.

[GLS99] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: portable parallel programming with the message-passing interface*. Vol. 1. MIT press, 1999.

[Gra99] Martin Grajcar. „Genetic list scheduling algorithm for scheduling and allocation on a loosely coupled heterogeneous multiprocessor system.“ In: *Design Automation Conference, 1999. Proceedings. 36th*. IEEE. 1999, pp. 280–285.

[Gus+03] Jan Gustafsson et al. „A tool for automatic flow analysis of C-programs for WCET calculation.“ In: *Object-Oriented Real-Time Dependable Systems, 2003.(WORDS 2003). Proceedings of the Eighth International Workshop on*. IEEE. 2003, pp. 106–112.

[GY92] Apostolos Gerasoulis and Tao Yang. „A comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors.“ In: *Journal of Parallel and Distributed Computing* 16.4 (1992), pp. 276–291.

[Hal84] Robert H. Halstead Jr. „Implementation of Multilisp: Lisp on a Multiprocessor.“ In: *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*. LFP '84. Austin, Texas, USA: ACM, 1984, pp. 9–17. ISBN: 0-89791-142-3. DOI: 10.1145/800055.802017. URL: http://doi.acm.org/10.1145/800055.802017.

[HF99] Debra Hensgen and Richard F Freund. „Dynamic Mapping of a Class of Independent Tasks onto Heterogeneous Computing Systems.“ In: *Journal of Distributed Computing, Special Issue on software support for distributed computing* 59.2 (1999).

[Hin15] Pieter Hintjens. *ØMQ - The Guide*. `http://zguide.zeromq.org/page:all`. [Online; accessed May-2015]. 2015.

[HL14] Maurice Herlihy and Zhiyu Liu. „Well-structured Futures and Cache Locality.“ In: *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '14. Orlando, Florida, USA: ACM, 2014, pp. 155–166. ISBN: 978-1-4503-2656-8. DOI: `10.1145/2555243.2555257`. URL: `http://doi.acm.org/10.1145/2555243.2555257`.

[HPC] HPC-OM. *HPC-Openmodelica for Multiscale Simulations of Techninal Systems and Applications for the Development of Energy-Efficient Working Machinery*. `http://hpc-om.de/`. Accessed: 2015-05-17.

[Hu61] T.C. Hu. „PARALLEL SEQUENCING AND ASSEMBLY LINE PROBLEMS.“ In: *Operations Research* 9.6 (1961), pp. 841–848. ISSN: 0030364X. URL: `http://search.ebscohost.com/login.aspx?direct=true&db=bth&AN=7688983&site=ehost-live`.

[Hwa+89] Jing-Jang Hwang et al. „Scheduling precedence graphs in systems with interprocessor communication times.“ In: *SIAM Journal on Computing* 18.2 (1989), pp. 244–257.

[KA99a] Yu-Kwong Kwok and Ishfaq Ahmad. „Benchmarking and comparison of the task graph scheduling algorithms.“ In: *Journal of Parallel and Distributed Computing* 59.3 (1999), pp. 381–422.

[KA99b] Yu-Kwong Kwok and Ishfaq Ahmad. „Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors.“ In: *ACM Comput. Surv.* 31.4 (Dec. 1999), pp. 406–471. ISSN: 0360-0300. DOI: `10.1145/344588.344618`. URL: `http://doi.acm.org/10.1145/344588.344618`.

[Koł+13] Joanna Kołodziej et al. „Hierarchical genetic-based grid scheduling with energy optimization.“ English. In: *Cluster Computing* 16.3 (2013), pp. 591–609. ISSN: 1386-7857. DOI: `10.1007/s10586-012-0226-7`. URL: `http://dx.doi.org/10.1007/s10586-012-0226-7`.

[Kru87] Boontee Kruatrachue. „Static task scheduling and grain packing in parallel processing systems.“ In: (1987).

[LP96] Jing-Chiou Liou and Michael A Palis. „An efficient task clustering heuristic for scheduling dags on multiprocessors.“ In: *Workshop on Resource Management, Symposium on Parallel and Distributed Processing*. 1996, pp. 152–156.

[Mal05] Grzegorz Malewicz. „Parallel Scheduling of Complex Dags Under Uncertainty.“ In: *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '05. Las Vegas, Nevada, USA: ACM, 2005, pp. 66–75. ISBN: 1-58113-986-1. DOI: 10.1145/1073970.1073981. URL: http://doi.acm.org/10.1145/1073970.1073981.

[MS98] Muthucumaru Maheswaran and Howard Jay Siegel. „A dynamic matching and scheduling algorithm for heterogeneous computing systems.“ In: *Heterogeneous Computing Workshop, 1998.(HCW 98) Proceedings. 1998 Seventh*. IEEE. 1998, pp. 57–69.

[PN98] Peter Puschner and Roman Nossal. „Testing the results of static worst-case execution-time analysis.“ In: *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE*. IEEE. 1998, pp. 134–143.

[PY79] C. H. Papadimitriou and M. Yannakakis. „Scheduling Interval-Ordered Tasks.“ In: *SIAM Journal on Computing* 8.3 (1979), pp. 405–409. DOI: 10.1137/0208031. eprint: http://dx.doi.org/10.1137/0208031. URL: http://dx.doi.org/10.1137/0208031.

[Sch96] JMJ Schutten. „List scheduling revisited.“ In: *Operations Research Letters* 18.4 (1996), pp. 167–170.

[Shi+08] KwangSik Shin et al. „Task scheduling algorithm using minimized duplications in homogeneous systems.“ In: *Journal of Parallel and Distributed Computing* 68.8 (2008), pp. 1146–1156.

[Sin+08] Gurmeet Singh et al. „Workflow task clustering for best effort systems with Pegasus.“ In: *Proceedings of the 15th ACM Mardi Gras conference: From lightweight mash-ups to lambda grids: Understanding the spectrum of distributed computing requirements, applications, tools, infrastructures, interoperability, and the incremental adoption of key capabilities*. ACM. 2008, p. 9.

[SL93] M.S. Squillante and E.D. Lazowska. „Using processor-cache affinity information in shared-memory multiprocessor scheduling.“ In: *Parallel and Distributed Systems, IEEE Transactions on* 4.2 (Feb. 1993), pp. 131–143. ISSN: 1045-9219. DOI: 10.1109/71.207589.

[Spo+09] Daniel Spoonhower et al. „Beyond Nested Parallelism: Tight Bounds on Work-stealing Overheads for Parallel Futures.“ In: *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*. SPAA '09. Calgary, AB, Canada: ACM, 2009, pp. 91–100. ISBN: 978-1-60558-606-9. DOI: 10.1145/1583991.1584019. URL: http://doi.acm.org/10.1145/1583991.1584019.

[SZ04a] Rizos Sakellariou and Henan Zhao. „A hybrid heuristic for DAG scheduling on heterogeneous systems.“ In: *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*. IEEE. 2004, p. 111.

[SZ04b] Rizos Sakellariou and Henan Zhao. „A low-cost rescheduling policy for efficient mapping of workflows on grid systems.“ In: *Scientific Programming* 12.4 (2004), pp. 253–262.

[Tan+11] Xiaoyong Tang et al. „A stochastic scheduling algorithm for precedence constrained tasks on Grid.“ In: *Future Generation Computer Systems* 27.8 (2011), pp. 1083–1091.

[THW02] H. Topcuoglu, S. Hariri, and Min-You Wu. „Performance-effective and low-complexity task scheduling for heterogeneous computing.“ In: *Parallel and Distributed Systems, IEEE Transactions on* 13.3 (Mar. 2002), pp. 260–274. ISSN: 1045-9219. DOI: 10.1109/71.993206.

[Ton+00] S. Tongsima et al. „Probabilistic loop scheduling for applications with uncertain execution time.“ In: *Computers, IEEE Transactions on* 49.1 (Jan. 2000), pp. 65–80. ISSN: 0018-9340. DOI: 10.1109/12.822565.

[Ull75] J.D. Ullman. „NP-complete scheduling problems.“ In: *Journal of Computer and System Sciences* 10.3 (1975), pp. 384–393. ISSN: 0022-0000. DOI: http://dx.doi.org/10.1016/S0022-0000(75)80008-0. URL: http://www.sciencedirect.com/science/article/pii/S0022000075800080.

[WBS09] Xiaofeng Wang, R. Buyya, and Jinshu Su. „Reliability-Oriented Genetic Algorithm for Workflow Applications Using Max-Min Strategy.“ In: *Cluster Computing and the Grid, 2009. CCGRID '09. 9th IEEE/ACM International Symposium on*. May 2009, pp. 108–115. DOI: 10.1109/CCGRID.2009.14.

[WG90] Min-You Wu and Daniel D Gajski. „Hypertool: A programming aid for message-passing systems.“ In: *IEEE Transactions on Parallel and Distributed Systems* 1.3 (1990), pp. 330–343.

[Wil+08] Reinhard Wilhelm et al. „The worst-case execution-time problem—overview of methods and survey of tools.“ In: *ACM Transactions on Embedded Computing Systems (TECS)* 7.3 (2008), p. 36.

[WS97] Min-You Wu and Wei Shu. „On parallelization of static scheduling algorithms.“ In: *IEEE transactions on software engineering* 23.8 (1997), pp. 517–528.

[Wu00] Min-You Wu. „Mcp revisited.“ In: *Department of Electrical and Computer Engineering, University of New Mexico* (2000).

[YB05] Jia Yu and Rajkumar Buyya. „A taxonomy of workflow management systems for grid computing.“ In: *Journal of Grid Computing* 3.3-4 (2005), pp. 171–200.

[YG92] Tao Yang and Apostolos Gerasoulis. „PYRROS: Static Task Scheduling and Code Generation for Message Passing Multiprocessors.“ In: *Proceedings of the 6th International Conference on Supercomputing*. ICS '92. Washington, D. C., USA: ACM, 1992, pp. 428–437. ISBN: 0-89791-485-6. DOI: 10.1145/143369.143446. URL: http://doi.acm.org/10.1145/143369.143446.

[Zah+10] Matei Zaharia et al. „Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling.“ In: *Proceedings of the 5th European conference on Computer systems*. ACM. 2010, pp. 265–278.

[Zon+13] Ziliang Zong et al. „Energy-Efficient Scheduling for Multicore Systems with Bounded Resources.“ In: *Green Computing and Communications (GreenCom), 2013 IEEE and Internet of Things (iThings/CPSCom), IEEE International Conference on and IEEE Cyber, Physical and Social Computing*. Aug. 2013, pp. 163–170. DOI: 10.1109/GreenCom-iThings-CPSCom.2013.49.

[ZS03] Henan Zhao and Rizos Sakellariou. „An experimental investigation into the rank function of the heterogeneous earliest finish time scheduling algorithm.“ In: *Euro-Par 2003 Parallel Processing*. Springer, 2003, pp. 189–194.

[ZS13] Wei Zheng and Rizos Sakellariou. „Stochastic DAG scheduling using a Monte Carlo approach.“ In: *Journal of Parallel and Distributed Computing* 73.12 (2013). Heterogeneity in Parallel and Distributed Computing, pp. 1673–1689. ISSN: 0743-7315. DOI: http://dx.doi.org/10.1016/j.jpdc.2013.07.019. URL: http://www.sciencedirect.com/science/article/pii/S0743731513001573.