Linköping University | Department of Computer Science
Master thesis, 30 ECTS | Datateknik
2019 | LIU-IDA/LITH-EX-A--19/004--SE

Optimizing a software build system through multi-core processing

Robin Dahlberg

Supervisor : August Ernstsson
Examiner : Christoph Kessler

External supervisor : Anna Gustavsson

Linköpings universitet
SE–581 83 Linköping
+46 13 28 10 00 , www.liu.se

Abstract

In modern software development, *continuous integration* has become a integral part of agile development methods, advocating that developers should integrate their code frequently. *Configura* currently has one dedicated machine, performing tasks such as building the software and running system tests each time a developer submits new code to the main repository. One of the main practices of continuous integration advocates for having a fast build in order to keep the feedback loop short for developers, leading to increased productivity. Configura's build system, named *Build Central*, currently uses a sequential build procedure to execute said tasks and was becoming too slow to keep up with the number of requested builds.

The primary method for speeding up this procedure was to utilize the multi-core architecture of the build machine. In order to accomplish this, the system would need to deploy a scheduling algorithm to distribute and order tasks correctly. In this thesis, six scheduling algorithms are implemented and compared. Four of these algorithms are based on the classic list scheduling approach, and two additional algorithms are proposed which are based on dynamic scheduling principles.

In this particular system, the dynamic algorithms proved to have better performance compared to the static scheduling algorithms. Performance on Build Central, using four processing cores, was improved with approximately 3.4 times faster execution time on an average daily build, resulting in a large increase of the number of builds that can be performed each day.

Acknowledgments

I would like to thank everyone at Configura for making me feel welcome at the company while doing my thesis and it was noticeable that everyone wanted the thesis succeed. I would especially like to thank my two supervisors Anna and Gustav for their support during the thesis. I am also grateful for the guidance and support from my supervisor, August Ernstsson, and my Examiner Christoph Kessler.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Background

This thesis was requested by *Configura AB* which is a company located in Linköping and develops software for the space planning industry like, e.g. , kitchen or office planning. Their main product is named *CET Designer* which provides sales personnel with support tools for all stages of a sales process including space planning, designing, cost estimates and more. CET Designer is developed using Configura's own programming language *CM*.

1.2 Motivation

Continuous integration is today one of the most applied practices in modern software development and is often a vital part of the development team's operation tool set. Configura currently relies on an internal build and test tool called *Build Central* to provide continuous integration services for the development process of CET Designer. This tool is used to perform tasks necessary for the development automatically and will build affected files during each submission of new code and run a set of predefined tests to verify that the product is still operational. Configura only had one deployed machine dedicated to running Build Central at the time this thesis was conducted.

Due to increasing size of the product source code, this build procedure had become too slow to handle each submission of code separately and would stack changes from several submits in one combined build. This often caused errors in determining which developer was responsible for a broken build and would give false-positive answers and falsely notify a developer, affecting the reliability of the tool negatively. This build procedure requires between 20 to 60 minutes to fully execute, depending on the extent to which the program needed to be built. Usually, code submissions will arrive more frequently than this, which caused build queues and consequently long wait time for developers. Having such time consuming builds is a violation of one of the main practices in continuous integration, which advocates for a fast build and short feedback loop [20, 17]. Since this build procedure is expected to continue increasing in scale and time consumption, the efficiency of Build Central had to be improved in order to provide more frequent builds. The chosen approach was to attempt to utilize more of the available processing power by *scheduling* individual build tasks to additional CPU cores.

The challenge was to find the most suitable solution which could both provide benefits to the current system but also allow the source code to scale and make Build Central a solid platform to extend upon. For instance if Configura later desires to expand Build Central to utilize several machines in a distributed build system instead of using one powerful machine, this solution should be able to be utilized to a large extent.

Many comparative studies of algorithms concerning this problem are theoretical or are most often conducted on simulated environments only and might therefore not be applicable for many projects that plan to apply scheduling algorithms to their products [38, 40, 27]. In this thesis, algorithms will be evaluated on a practical, in addition to a simulated, scenario in order to provide a good overall comparison between algorithms.

1.3 Aim

The purpose of this thesis is to reduce the total execution time of the procedure which builds and tests the product. This would allow faster builds and tests which would allow CET Designer to increase in size without further escalating time consumption and result in fewer daily builds. The preliminary goal was to reduce the build time by at least half. Stability was also of high importance and the parallel implementation should preferably never cause build errors related to this implementation.

1.4 Research Questions

- How much more efficient can Build Central become by utilizing parallel processing instead of sequential?
- Do different scheduling algorithms perform equally well on simulated environments as compared to actual application programs?

1.5 Delimitations

Due to certain hardware limitations, the number of processing units used in this thesis will be limited to a maximum of four, i.e. one processor with four cores.

1.6 Report Structure

This thesis starts with Chapter 1, motivating the problem to the reader and defining the aim of thesis as well as presenting research questions to be answered at the end of thesis. Chapter 2 will provide a background on subjects related to this thesis. The reader will be introduced to build systems, continuous integration, basic graph theory and parallel processing. In Chapter 3 the reader will be presented with theory regarding scheduling. This includes the algorithms that are later used in the thesis as well as alternative approaches such as branch-and-bound and genetic algorithms. Chapter 4 will feature the method used to acquire the results and will also motivate certain design decisions. The acquired results from the defined scenarios will be covered in Chapter 5 where the experimental results will also analyzed and evaluated. Further, this section will also include a discussion regarding the methodology of the thesis and validity of the sources. Lastly, a conclusion and possible future work will be presented in Chapter 6.

2 Background

This chapter provides an introduction to areas which are deemed important to understand this thesis.

2.1 Software Build Systems

The practice of automated software builds originates from the creation of the *make* system developed by Bell laboratories [18]. Despite its age, *make* is still one of the most used build systems to this day, which is most likely due to the large amount of legacy systems still in use and because it is included in Linux by default, which make it easy to access [53]. Since the introduction of *make* a large number of build tools have been developed based on similar principles, some examples are Ant, CMake, Maven and Ninja.

A build system is a tool used in software development with the primary objective of providing an automated procedure for creating program executables from the source code. This is the traditional view of a software build system and does mainly include compilation and linking of source code. However, modern build systems often provide utilities such as automated tests, document generation, code analysis and other related tools [53]. The increased importance of build system utilities is mainly caused by the recent domination of agile development methods where the product is build incrementally. Consequently, an automated build pipeline is required [3]. This evolution has resulted in build systems becoming increasingly complex and has become collections of tools rather then a single tool, and contains most of the elements necessary to develop a large software application. An example of a modern build system which provides most of the utilities mentioned above is the popular *jenkins* system [32].

A good build system is supposed to have the following characteristics according to Smith [53]:

- **Convenience** – The system should be easy to use and not cause developers to waste time dealing with the system functionality.
- **Correctness** – A build should fail because of compile errors and not because of faulty file linking or compiling of files in the wrong order by the system. The produced executable program should correctly reflect the content of the source files.

- **Performance** – The system should utilize the available hardware to the greatest extent possible to keep the build fast.
- **Scalability** – The system should be able to provide the above mentioned characteristics, even when the system is building very large applications.

Advantages of having a good build system are claimed by Smith to lower product development cost and improve overall product quality due to the removal of human interaction from error prone and redundant tasks. A previous survey has estimated that manually handling the build procedure causes on average 12 % overhead in development processes [37]. Typical problems that most developers encounter and which could potentially cause this overhead are: *dependency issues* where incorrect dependencies could potentially cause false compilation errors or cause parts of the build to generate faulty output; *slow build* where developers will waste time waiting for their build to finish, resulting in productivity losses; *updating build configuration* could be a tedious and time-consuming task, especially if the build system is hard to use and is dependent on a few key personnel who need to be consulted before changes can be confirmed [53]. A build system can also be utilized in different ways depending on the purpose and need of the developer. The three most common types in modern software projects according to Smith are [53]:

- **Developer build** – Private build for a developer in an isolated workspace which will only be built using that specific developer's changes to the source code. The resulting output is not shared with other developers.
- **Release build** – A build with the purpose of creating the complete software package. The resulting software is then exposed for testing and quality control and is released to customers if it fulfills the specified quality requirements.
- **Sanity build** – Similar to release build but will not be released to customers but is rather used to verify that the current state of the project is "sane", meaning that the source should be able to compile without errors and pass a set of basic functionality tests. This is often named nightly or daily build.

2.2 Continuous Integration

Continuous integration (CI) is today a widely applied practice in modern software development where developers are instructed to regularly integrate their code to a common main repository, causing multiple integrations per day. This practice originated from the development method *Extreme programming* (XP) as one of the main practices [20]. CI is often used as a key component in agile development methods, like *SCRUM* or *Rapid application development* (RAD), and often provides benefits towards productivity and product quality [55]. The purpose of advocating early and frequent integration of code is to prevent large integration issues during late stages of development, also referred to as "integration hell" [17]. This integration policy also provides continuous assurance that the code compiles and the application should successfully pass a number of predefined acceptance tests. This ensures rapid feedback on the current state of the project and allow developers to work with a higher level of confidence in the existing code [17]. Further, frequent integration will assist in reducing risks to the project and facilitates early detection of defects, motivated by Fowler in his article on continuous integration [20]:

"...Any individual developer's work is only a few hours away from a shared project state and can be integrated back into that state in minutes. Any integration errors are found rapidly and can be fixed rapidly...". [20]

The overall goal of using CI is to reduce manual work to largest extent possible but simultaneously having consistent and frequent quality assessment. This is achieved by automating all possible tasks in a normal development iteration. An example of a CI-system is illustrated in Figure 2.1. In order to ensure an effective CI-process there are several key practices that should be implemented. A selection of main practices which are relevant to this thesis are described below:

Single Source Repository

Software projects that utilize CI should have one mainline branch where the latest successfully built and tested code shall reside. The work flow should be that developers take a private copy of the main repository and make changes locally. When a developer is ready with their changes, it is the responsibility of the same developer that the program still compiles and successfully passes all tests before adding these changes to the main repository.

Automate The Build

Building the entire program can be a complicated and time consuming endeavor and should to the highest extent be automated and possible to initialize using a single command, either via build system or script. The benefits are shorter time stolen from developers and fewer errors due to human interaction.

Make The Build Self-Testing

Additionally from building the program, the build system should also include predefined tests to validate that the latest version of the program is robust and functioning. This will assist in catching bugs early, which in turn provides a stable base code.

Every Commit Should Build The Mainline On An Integration Machine

To ensure that the latest integration was successful and does not cause any problems in the mainline, the build system should monitor the version control system for incoming code commits and initiate a new build with these new changes. The developer responsible for the latest commit should then be notified of the build results so that eventual errors can be fixed quickly. This will ensure that the mainline remains in an continuous healthy state.

Keep The Build Fast

An essential part of CI is to provide developers with rapid feedback on the latest changes made to the program. That is why it is crucial to have a fast build system and every minute that can be reduced from the build execution results in time saved for a developer. Since a central part of CI is to integrate often, this will result in a lot of time saved with a fast build.

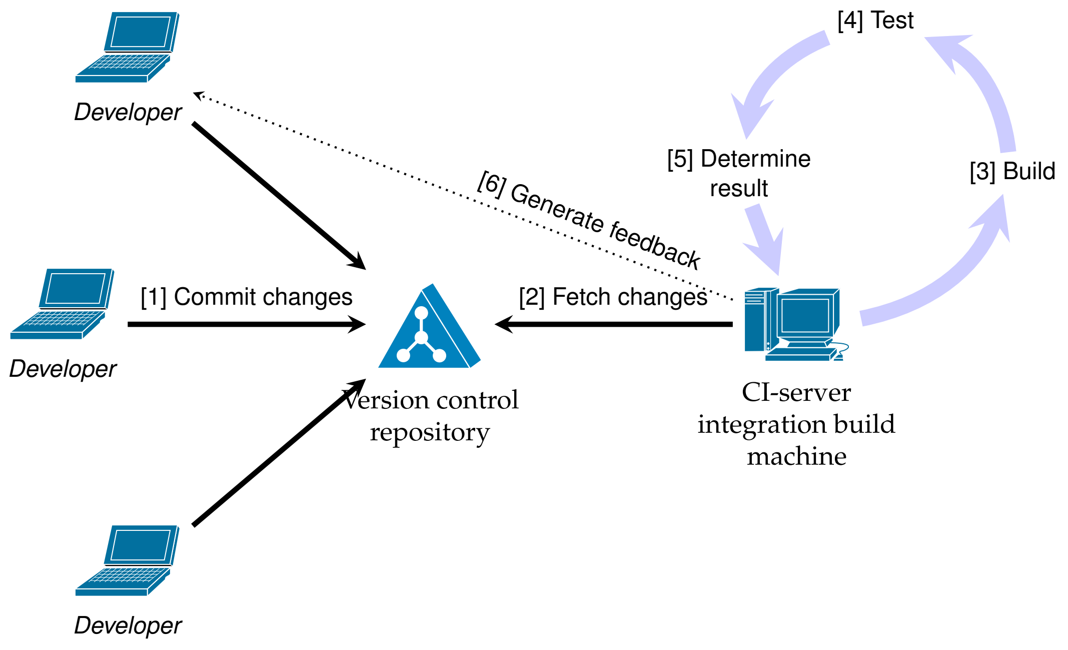


Figure 2.1: Example of a CI-system.

2.3 Directed Acyclic Graph

A *directed acyclic graph* (DAG) is a generic way of modeling different relationships between information. DAGs are extensively used in computer science and are useful when modeling probabilities, connectivity and causality like in Bayesian networks. A DAG consists of a finite set of nodes and a finite set of directed edges, where an edge represents connectivity between two nodes. What defines a DAG is the fact that it will not contain any cycles and connectivity is directional, which consequently means that there is no chance that any path in the graph leads back to a previously visited node, i.e. the graph has a topological ordering. Hence, no starting node of an edge can occur later in the sequence ordering than the end node of the edge. [57]

- **Nodes/Vertices** - Graphically denoted as ovals or circles in a DAG and represents a task, resource or value depending on the information being modeled with the graph.
- **Edges** - Edges are used to represent precedence constraints among nodes and can also be assigned a weight to indicate the level of significance of the edge.

There are many examples of real life applications of DAGs, for instance the git commit graph, family trees and Bayesian probability graphs are all DAGs, see Figure 2.2 for a basic illustration. Another relevant example is the *make* build system, which constructs a DAG-model as a dependency graph to keep track of the compilation order of files [18].

DAGs are naturally well suited for scheduling problems where tasks need to respect certain precedence restrains. The operation used to traverse a DAG and make a topological ordering of the tasks is called *topological sorting* and is usually the first step of any algorithm that is applied on a DAG [52].

When describing a DAG in mathematical terms, usually it will be denoted using $G = (V, E)$ where V represents the finite set of vertices and E is the set of edges where $E \subseteq VxV$. Suppose we were to use this to describe the DAG in Figure 2.2, we would get:

- $V = \{1, 2, 3, 4, 5, 6\}$
- $E = \{(1, 3), (2, 3), (2, 5), (3, 4), (3, 5), (3, 6), (4, 6), (5, 6)\}$

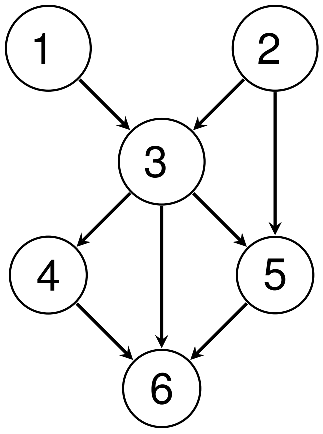


Figure 2.2: Illustration of a DAG.

2.4 Parallel Processing

Single core architecture was for a long time the dominant processor architecture and improvements to the CPU clock frequency followed a similar trend as *Moore's law*, where the number of transistors per mm^2 chip area doubled every two years, and consequently increased the processing power every two years [48]. Then this increase in clock frequency started to diminish, being restricted by memory speed, heat leakage and instruction level parallelism, also referred to as the three "walls" [6]. The rising limitations of increasing single core clock frequency caused computer architects to shift focus towards increasing the number of processing units instead of trying to further improve one powerful processing core [4].

To help distinguish the different processor architectures, one can divide these in four different categories dependent on two different dimensions. One dimension represents the number of *instruction streams* that an architecture can potentially process simultaneously. Similarly, the second dimension is the number of *data streams* that can be processed at the same time. This classification is called *Flynn's taxonomy* and most computer architectures can be inserted in one of these categories, see graphic illustration in Figure 2.3 [19].

SISD is the traditional approach and is entirely sequential where a single instruction stream operates on a single data element. Today, most computers have *MIMD* architectures where several processing elements can execute instructions independently and simultaneously on separate data [8, 4]. This is useful for programs which need tasks to run simultaneously like for instance server applications where the requests from several clients need to be managed separately and concurrently [45]. *SIMD* excels at other types on computations where all processing units execute the same instruction at a given clock cycle but on different data elements. Such architectures utilize data level parallelism but not concurrency and the execution is therefore deterministic. This is ideal for certain tasks with unilateral instructions like signal and image processing [8].

Although multiprocessor solutions have been available for several decades, these powerful improvements has not yet been fully utilized by software developers. The theoretical

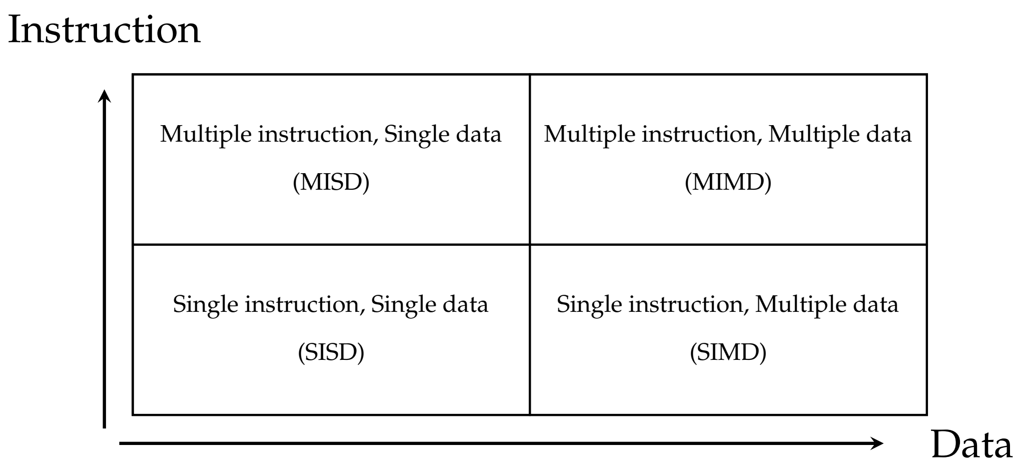

Instruction
Multiple instruction, Single data
(MISD)
Multiple instruction, Multiple data
(MIMD)
Single instruction, Single data
(SISD)
Single instruction, Multiple data
(SIMD)
Data

Figure 2.3: Flynn's taxonomy.

efficiency increase that parallel processing can provide over sequential execution is substantial but also introduces new issues which will prevent maximum hardware utilization. First, concurrency in programming is hard and the absence of viable tools and abstractions were for a long time a problem [56]. This has led to increased expectations on developers to understand the concept of concurrency. In addition, not all sequential applications and problems are suitable for parallelization and receiving full efficiency for every added processing unit is in most cases not realistic in practice. This problem was theorized by Amdahl, who defined upper bounds of the speedup gained from parallelization known as *Amdahl's law* [5]. *Amdahl's law* defines the theoretical speedup of an entire task and that the total speedup of a task is always limited by parts in the program which can not be parallelized. This can be denoted as the following equation:

$$Speedup = \frac{1}{(1-p)+\frac{p}{s}} \tag{2.1}$$

Where p represent the fraction of the task that is parallelizable and s denotes the speedup of the parts that benefit from parallelization. This argument presented by Amdahl still applies to this day with the conclusion that the primary objective when seeking to achieve better speedup is to seek greater parallelism [28].

3 Scheduling

In this chapter the most central element in this thesis is introduced. This includes a background on the area of scheduling as well as detailed description and demonstration of the algorithms which are used in the method of this thesis. Task execution time estimation will also be introduced due to it being closely related to the results from the algorithms in this thesis. Lastly, theory on branch-and-bound and genetic algorithms as alternative approaches will also be presented.

3.1 Background

Scheduling has been extensively studied for several decades all the way back to the 1950s when industrial work processes needed to be managed in a certain order. Computer scientists later encountered this problem when trying to efficiently utilize scarce hardware resources (CPU, memory and I/O devices) in operating systems. Scheduling can be applied to multiple areas such as operations in a manufacturing process, airway planning, project management, network packet management and many more [44]. However, in the context of this thesis it is general task scheduling in computer science that is described.

Scheduling refers to the sequencing problem that arises when there are multiple choices of ordering a set of activities or tasks to be performed [13]. In computer science there are three major types of scheduling problems which are the *open-shop*, *job-shop* and *flow-shop* scheduling. When dealing with these sort of problems there are two central elements to introduce:

- **Job/Task** – This indicates an operation to be processed on a machine. The term ***task*** is used throughout this thesis.
- **Machine/Processor/Core** – Indicates a processing unit which will handle processing of a task. In this thesis we will use the terms interchangeably.

In the *open-shop* scheduling a set T of n tasks will have to be processed for certain amount of time by each of a set M with m machines, in any order without precedence constraints. The *flow-shop* problem is similar to *open-shop* but with the difference that tasks will instead be forced to be processed in a predetermined order by the set of machines. It is however the *job-shop* problem that is implicitly referred to when using the term *scheduling* in this thesis.

These problems can have different variations depending on the problem, nature of the machine architecture or how the tasks are composed. Tasks may for instance have constraints that other tasks have to finish being processed before being able to be started. Such precedence constraints between tasks are often modeled using a DAG and such an example can be observed in Figure 2.2, where task (3) has to wait for task (1) and (2) to finish before being eligible for processing. Machines can vary between being heterogeneous or homogeneous, heterogeneous meaning that the machines in the same system have different capabilities like only being able to handle a specific type of tasks or having different processing power. Homogeneous means that machines have identical properties. Machines can also be restricted to certain tasks communication delays between machines or whether the machines can handle task preemption or not. An example of a scheduling problem is the traveling salesman problem where the salesman can be viewed as the machine (*m*=1) and cities are the tasks with traveling time as processing cost [13].

The scheduling procedure will have an objective of optimizing a certain aspect of the sequencing problem depending of the nature of the problem. The most common objective being the minimization of the schedule *makespan* which is the total length of the schedule e.g. the time when all the tasks have finished processing, mathematically denoted as 3.1.

$$C_{max} = max_{i=1,n}\, C_i \tag{3.1}$$

Where C_i denotes the completion time of task i. By implementing task scheduling where it is applicable, one can achieve more efficient utilization of limited processing resources which in turn can lower the cost of executing a software application [44].

An example of an scheduling algorithm is FIFO ("first-in, first-out"), which is one of the most basic and well known scheduling algorithms. In FIFO, tasks are executed in the order in which they requested resources, so it can basically be compared to a store queue. This basic algorithm may seem fair and will solve many basic scheduling problems, however when these problems become more complex, such an algorithm will most likely not provide a satisfactory solution. For instance when scheduling tasks to execute in a multiprocessor architecture, one would often have to consider aspects like precedence constraints among tasks which increases complexity.

Next, we give an introduction to some terminology regarding scheduling:

- **Weight (w_i)** – Represents the computation cost of task n_i.
- **Entry task** – A task which has no precedence constraints.
- **Exit task** – No later tasks exist that depend on this task.
- **Makespan (C_{max})** – The total schedule length, see Section 3.1.
- **t-level** – The t-level of task n_i is the longest path from an entry task to n_i.
- **b-level** – The b-level is longest path from n_i to an exit task.
- **Static level** (SL) – The b-level of a task but with edge weights excluded.
- **Critical path** (CP) – Heaviest computational path of a DAG.
- **Earliest start time** (EST_i) – The earliest time when task n_i can be started.

Finding an optimal solution for most scheduling problems is known to be NP-hard in the general case, which consequently means that there is no fast procedure for obtaining the optimal solution to the problem [42, 43]. Those few cases where an optimal solution can be found in polynomial time are usually simplified by several assumptions. For instance, all task processing times being equal or restrictions to the number of processors used in the scenario are common assumptions. One such algorithm is explained later in this thesis, see Figure

3.1 [12]. Other common simplifications, besides the two mentioned above, include unlimited number of processors, zero communication costs or full connectivity between processors. However, these scenarios are not regarded as viable in real world scenarios and the task scheduling problem is therefore generally regarded as NP-complete [40].

Scheduling algorithms can usually be divided in two broad categories, static or dynamic. Static scheduling refers to scheduling where the execution queue is produced before the system starts the execution of tasks. Consequently, most parameters concerning the scheduling procedure must be known before task execution in order for the scheduler to make accurate predictions and calculations. This includes task execution times, dependencies between tasks, communication costs and other possible synchronization requirements [40]. Dynamic scheduling will however determine task priority continuously during runtime and will in turn produce a scheduling overhead. As a consequence the dynamic scheduler will have to rely on real time information about the state of the system. Dynamic scheduling is further explained in Section 3.6.

3.2 Task Execution Time Estimation

An important part of any scheduling algorithm is to have an accurate estimation of the execution time of each individual task. For the scheduler to have access to an accurate approximation of this attribute is a major factor in the total execution time of the schedule [22, 10]. Studies involving task scheduling often neglect this part of the problem and usually assume that tasks' execution times are known and are constants. However, in practice, the time required to execute a task is often a stochastic variable [10, 31].

There are three major approaches to solving this problem: *code analysis*, *code profiling* and *statistical prediction* [31]. In code analysis, estimation of the task execution time is acquired through analysis of the task source code on a specific architecture and code type [50].

Code profiling is similar but instead attempts to define the composition of primitive code types that each individual task source code consist of. These code types are benchmarked to obtain the performance and combined with the composition profile of a task, an estimation for task execution time can be produced [21].

Lastly, the approach of statistical prediction involves using past observations to estimate the future execution time of task. A set of past observations for each task are kept at each machine and a new measured value added to this set every time the task is executed. This will increase the accuracy of the estimated value by the statistical algorithm as the set of observations increases. These statistical algorithms can vary in complexity from using the latest measured observation to using distribution functions and probabilities [10, 54].

3.3 List Scheduling

List scheduling is a classic approach when distributing tasks with precedence constraints to processors and there exists a large number of algorithms that are based upon this approach [38, 40, 27]. The main purpose of the list scheduling approach is to define a topological ordering of tasks using priority values and arrange these tasks in a list in decreasing order of priority [24, 1]. If two or more tasks have the same priority value the tie will be broken, depending on the algorithm, by some heuristic or an other priority value [40]. After all tasks have been assigned a priority value, the next step is to schedule the tasks onto a suitable processor or machine which allows the *earliest starting time* (EST).

When assigning priority to a task there are several attributes that can be used to this purpose. Two of the more prominent attributes are *t-level* and *b-level* which are the top and bottom level respectively. The *t-level* of a task n_i is the sum of all weights in tasks and edges along the longest path from an entry task to n_i excluding the weight of n_i , described in Equation 3.2.

$$\textit{t-level}(n_i) = \max_{n_j \in pred(n_i)} \{\textit{t-level}(n_j) + w_j + c_{ji}\} \tag{3.2}$$

Where w_i indicates the weight of task i, c_{ji} represents the communication cost between task j to task i and $pred(n_i)$ is the set of tasks that directly precede task i, e.g., all immediate parent tasks of n_i.

The *b-level* is similar but the result is instead the longest path from task n_i to an exit task including the weight of n_i and communication cost.

$$\textit{b-level}(n_i) = \max_{n_j \in succ(n_i)} \{\textit{b-level}(n_j) + c_{ji}\} + w_i \tag{3.3}$$

Where $succ(n_i)$ is the set of tasks that directly succeeds task i e.g. all immediate child tasks of n_i. Another attribute closely related to *b-level* is the *static level* (SL) which is calculated in the same manner as b-level except that edge weights are excluded, described below in Equation 3.4.

$$SL(n_i) = \max_{n_j \in succ(n_i)} \{SL(n_j)\} + w_i \tag{3.4}$$

Next, the important notion of *critical path* (CP) is introduced. The CP of a DAG is related to b-level and represents the longest path of a DAG [35, 40]. Defined e.g. by Kwok and Ahmad:

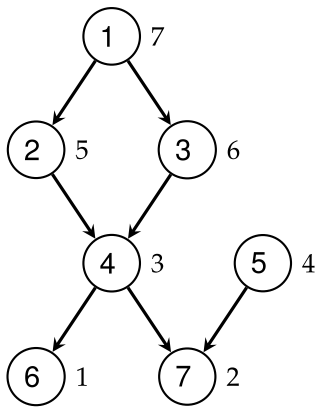
"A Critical Path of a task graph is a set of nodes and edges, forming a path from an entry node to an exit node, of which the sum of the computation costs and communication costs is the maximum."[39]

If we consider the DAG in Figure 3.2, we can see that the CP consists of tasks $\{n_1, n_4, n_9, n_{12}, n_{13}\}$ which can also be observed by examining the b-level which in this example is the same as SL since no communication cost is employed. As a consequence, the CP provides a lower bound for the final makespan length since it will not be possible to provide a schedule shorter than the CP. It is also worth mentioning that a CP of a graph is not necessarily unique, like for instance if w_2 in Figure 3.2 increased by one.

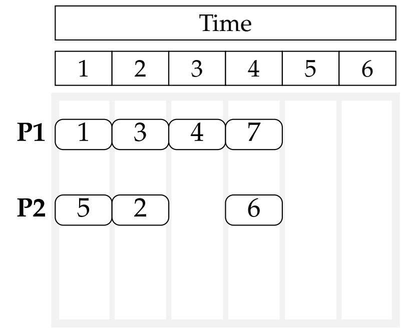
Further, in list scheduling there are two phases: the task prioritization phase and the processor selection phase. Hence, list scheduling algorithms can be divided in two categories, *static* and *dynamic* (not to be confused with dynamic scheduling) which implement these phases differently. In static list scheduling the prioritizing phase is performed before the processor selection phase whilst in dynamic list scheduling the two phases are combined and the priority of a task can be altered [27].

In Figure 3.1, an example of the well known *Coffman algorithm* [12] is presented to demonstrate how a basic list scheduling works. This algorithm is designed to generate optimal schedules for dual-processor system, with all task computation times being one arbitrary unit in size. The first step in the algorithm is to assign numeric labels to each task according to the order which tasks are located in the DAG 3.1(a), beginning from the bottom level and following left to right in horizontal order of the DAG. After all tasks have been assigned a label, the next step is to form a sorted list with labels in descending order. This list represents the order in which the tasks will be scheduled. Next, we iterate over the list and schedule each task to the processor which allows the earliest starting time of the task. The resulting schedule can be observed in Figure 3.1(b). [12]

In the following subsections, the algorithms that are applied in this thesis is described in further detail. These algorithms are applied to the example DAG in Figure 3.2. In this DAG, the numbers located on the right side of the tasks will represent the task execution cost and the arrows represents dependencies between tasks. Observe that this is not the case in the previous example in Figure 3.1, where these numbers on the right side of the tasks represented task priority.

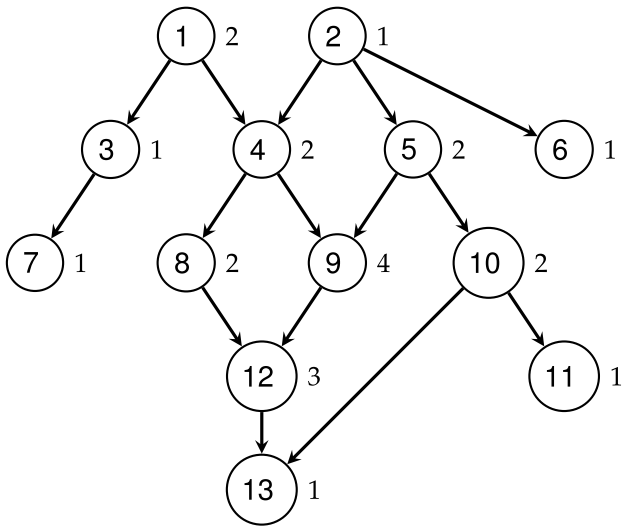


(a)



(b)

Figure 3.1: Illustration of the *Coffman algorithm*.



(a) *Task dependency graph*

Node	SL	t-level
1	12	0
2	11	0
3	2	2
4	10	2
5	10	1
6	1	1
7	1	3
8	6	4
9	8	4
10	3	3
11	1	5
12	4	8
13	1	11

(b) *Table of static priority values*

Figure 3.2: Example DAG.

Highest Level First With Estimated Times

The *Highest level first with estimated times* (HLFET) is an established list scheduling algorithm and introduced by Adam et al. in 1974 [1]. HLFET is a static list scheduling algorithm which determines the allocation priority based on the SL which was introduced earlier. Consequently, this algorithm basically prioritizes the tasks which belong to the critical path of the graph since SL represents the combined weight to an exit task. HLFET uses a non-insertion approach which means that it does not consider utilizing possible idle slots in the schedule. Idle slots are further introduced in Section 3.3.

By applying the algorithm described above in Figure 3.3 on the example graph provided in Figure 3.2 we will get the resulting step by step allocation procedure shown in Figure 3.4. We can observe that the tasks with a path going through task 9 are highly prioritized.

1 Calculate the static b-level of each task.

2 Make a ready list in a descending order of static b-level. Initially, the ready list contains only the entry nodes. No rule to breaking ties.

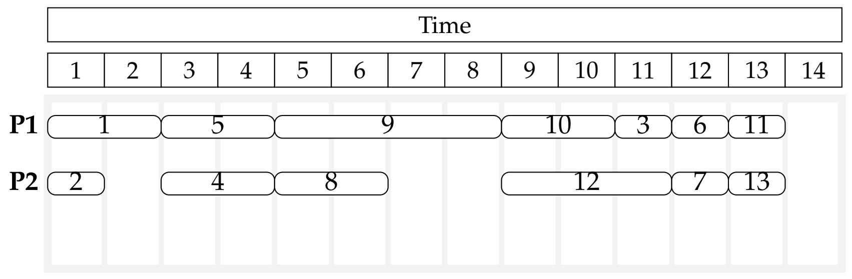
3 Schedule the first node in the ready list to a processor that allows the earliest execution, using the non-insertion approach.

4 Update the ready list by inserting nodes that are now ready for scheduling.
Repeat procedure from **2** until all nodes are scheduled.

Figure 3.3: Highest level first with estimated times (HLFET) algorithm.

Step	Selected task	EST-P1	EST-P2	Selected P	Ready queue	New tasks
1	1	0	0	P1	{ 2 }	{ 3 }
2	2	2	0	P2	{ 3 }	{ 4,5,6 }
3	4	2	1	P2	{ 5,3,6 }	{ 8 }
4	5	2	4	P1	{ 8,3,6 }	{ 9,10 }
5	9	4	4	P1	{ 8,10,3,6, }	{ }
6	8	8	4	P2	{ 10,3,6 }	{ 12 }
7	12	8	6	P2	{ 12,3,6 }	{ 13 }
8	10	8	11	P1	{ 3,6,11 }	{ 11 }
9	3	10	11	P1	{ 6,11,13 }	{ 7 }
10	6	11	11	P1	{ 7,11,13 }	{ }
11	7	12	11	P2	{ 11,13 }	{ }
12	11	12	12	P1	{ 13 }	{ }
13	13	13	12	P2	{ , }	{ }

(a) *Step by step scheduling procedure for HLFET*



(b) *Resulting schedule from step by step procedure presented in (a)*

Figure 3.4: Illustration of the HLFET algorithm applied to the DAG in Figure 3.2.

Insertion Scheduling Heuristic

The *insertion scheduling heuristic* (ISH) is an extension of the HLFET algorithm provided by Kruatrachue [36]. This algorithm employs the exact same procedure as HLFET, prioritizing the SL attribute of a task. The difference being that the ISH algorithm attempts to utilize idle slots that appear in a schedule when a processor is assigned a task which can not be started instantly, i.e. when the task EST is higher than the time when the processor becomes available. An example is the idle time slot created when task 4 in Figure 3.4 is assigned to

1 Calculate the static b-level of each node.

2 Make a ready list in a descending order of static b-level. Initially, the ready list contains only the entry nodes. No rule to breaking ties.

3 Schedule the first node in the ready list to a processor that allows the earliest execution, using the non-insertion approach.

4 If scheduling this task causes an idle time slot in the schedule, find one or several tasks that can fit in this idle slot which can not be scheduled earlier on any other processor.

5 Update the ready list by inserting nodes that are now ready for scheduling.
Repeat procedure from **2** until all nodes are scheduled.

Figure 3.5: Insertion scheduling heuristic.

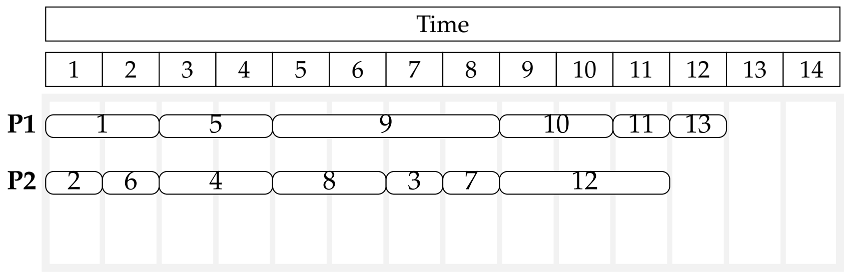


Figure 3.6: Result from applying the ISH algorithm on the DAG in Figure 3.2.

P2 which is available at time 1. However task 4 has an EST of 2 due to the dependency to task 2 and therefore cannot be initiated when P2 becomes available, causing P2 to wait one time unit before executing task 4. When such a situation emerges, the ISH algorithm tries to utilize the newly created schedule hole by inserting other tasks available from the ready queue that can be executed within the time interval and cannot be scheduled earlier on any other processor.

We apply ISH, described in Figure 3.5, to the DAG provided in Figure 3.2. We can observe the result in Figure 3.6 and then compare it to the previous result Figure 3.4. In this specific case, tasks 3, 6 and 7 can be scheduled earlier by utilizing the idle slots and reducing the makespan by one which is the same cost as the CP of the graph, thus the resulting schedule is an optimal solution.

Earliest Time First

The *earliest time first* (ETF) algorithm is considered a dynamic list scheduling algorithm (not to be confused with dynamic scheduling) since it treats the EST attribute as a dynamic attribute and uses it to determine which task gets allocated to a processor [30]. The ETF algorithm determines, at each step of the scheduling procedure scheduling, the EST for every task in the ready list for every processor, i.e. the EST for all task-processor pairs. The task-processor pair that provides the lowest EST is chosen for allocation. In the event that two or more tasks share the lowest EST, the tie will broken using the SL value. This procedure is described in Figure 3.7 and illustrated in Figure 3.8, presented in the same fashion as HLFET with a step

1 Compute the static b-level of each node.

2 Initially, the pool of ready nodes includes only the entry nodes.

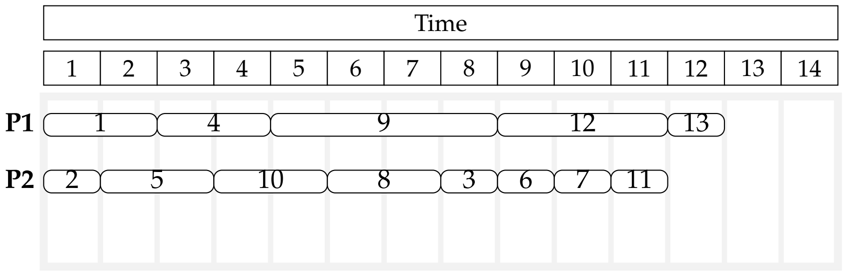
3 Calculate the earliest start-time on each processor for each node in the ready pool. Pick the node-processor pair that gives the earliest time using the non-insertion approach. Ties are broken by selecting the node with a higher static b-level. Schedule the node to the corresponding processor.

4 Add the newly ready nodes to the ready node pool.
Repeat procedure from **3** until all nodes are scheduled.

Figure 3.7: Earliest time first.

Step	Selected task	EST-P1	EST-P2	Selected P	Ready queue	New tasks
1	1	0	0	P1	{ 2 }	{ 3 }
2	2	2	0	P2	{ 3 }	{ 4,5,6 }
3	5	2	1	P2	{ 3,4,6 }	{ 10 }
4	4	2	3	P1	{ 3,6,10 }	{ 8,9 }
5	10	4	3	P2	{ 3,6,8,9 }	{ 11 }
6	9	4	5	P1	{ 3,6,8,11 }	{ }
7	8	8	5	P2	{ 3,6,11 }	{ 12 }
8	3	8	7	P2	{ 6,11,12 }	{ 7 }
9	12	8	8	P1	{ 6,7,11 }	{ 13 }
10	6	11	8	P2	{ 7,11,13 }	{ }
11	7	11	9	P2	{ 11,13 }	{ }
12	11	11	10	P2	{ 13 }	{ }
13	13	11	11	P1	{ }	{ }

(a) *Step by step scheduling procedure for ETF*



(b) *Resulting schedule from (a)*

Figure 3.8: Illustration of the ETF algorithm applied to the DAG in Figure 3.2.

by step table. The format is similar, but we keep in mind that all calculations all performed at the beginning of each step. The resulting schedule was, just as with ISH, an optimal solution but with a different ordering of the tasks.

Dynamic Level Scheduling

Dynamic level scheduling (DLS) is similar in many ways to the ETF algorithm but this algorithm utilizes a different attribute as part of the priority calculation [51]. DLS uses an attribute called *dynamic level* (DL) which is calculated by subtracting the EST for a task from the SL value of the task. Next, use the same method as ETF by, at each scheduling step, calculating the DL for every task-processor pair and selecting the pair with the highest DL. This algorithm has a tendency of scheduling tasks with high SL values in the early stages and then tends to schedule with regard to EST in the later stages of the scheduling process. The algorithm is described in Figure 3.9 and just as the case with ISH, only the resulting schedule is presented below in Figure 3.10. The DLS algorithm also provides an optimal solution to the example graph, just as ISH and ETF.

```
1  Compute the static b-level of each task.
2  Initially, the pool of ready tasks includes only the entry tasks.
3  Calculate the DL on each processor for each task in the ready
   pool by subtracting the EST of the task-processor pair from the
   SL value of the task. Pick the task-processor pair that provides
   the highest DL and schedule using the non-insertion approach.
   Schedule the task to the corresponding processor.
4  Add the new tasks which can be scheduled to the ready task pool.
   Repeat procedure from 3 until all tasks are scheduled.
```

Figure 3.9: Dynamic level scheduling.

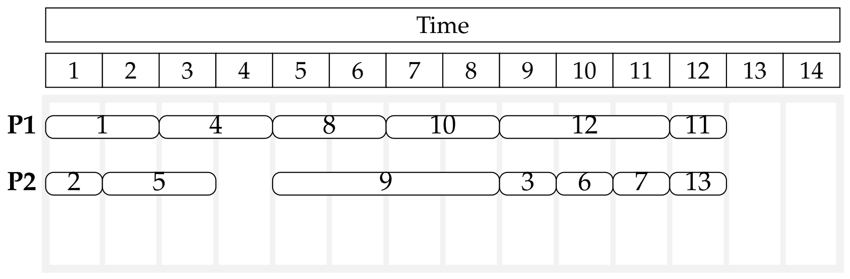


Figure 3.10: Result from applying the DLS algorithm on the DAG in Figure 3.2.

3.4 Branch And Bound

An alternative to the common list scheduling algorithms is to use a branch-and-bound (B&B) strategy to find an optimal or near-optimal schedule. This algorithm paradigm was first developed by Land and Doig [41]. The B&B method is one of the most commonly tools used for solving NP-hard combinatorial problems like the traveling salesman problem and other mathematical optimization problems [11]. The B&B approach consists of systematic exploration of feasible solutions by using the state space search and uses bounds calculations to restrict the search procedure from examining every possible solution which will usually not be a feasible alternative due to computational requirements. The B&B method consists of three different steps which is defined in Figure 3.11 [9].
B&B algorithms also employs different strategies for selecting which subproblem gets explored in the next computation iteration. Determining which strategy to use often involves trade-offs between memory requirements and computational requirements. Examples of such search strategies are *depth-first*, *breadth-first* or *best-first* search.

An example where such an algorithm is used to solve the task scheduling problem is the *depth-first with implicit heuristic search* (DF/IHS) provided by Kasahara [34]. The DF/IHS uses a list scheduling heuristic to provide a prioritized list of tasks which will provide the order in which the search is conducted. At each step, all combinations of task to processor mapping with currently available tasks at the current allocation time are created as new branches. One branch represents one combination of task-to-processor mapping. As the name suggests the algorithm uses a depth first approach where the branch selected to expand is the first node in the list containing unexplored nodes. This way a good solution is found early in the search procedure and unsuitable branches are cut early. This provides benefits towards

1. **Branching** – This step is used to partition the problem into smaller subproblems, where each subproblem is represented as a "branch" of the main problem. These branches will in turn provide a new basis for creating new subproblems, making the procedure recursive. Each generated branch is kept in a list containing all unexplored subproblems and selecting which subproblem to expand depends on the algorithm. This procedure will dynamically create a search tree with each expanded subproblem being represented as a node and will initially only contain the main problem.

2. **Lower bounding** – A function is needed which will estimate the best possible solution that a specific subproblem can provide.

3. **Upper bounding** – The goal of the method is to minimize or maximize a certain aspect of the problem. Without any reduction of the branch exploration would result in a brute-force enumeration of all possible solutions. In order to make the search more efficient, the lower bound value is compared to the upper bound value which is the best solution found so far or an estimated value provided before the search began. If the lower bound value of a subproblem is higher than the upper bound value it will not be necessary to continue the search of that subproblem's branch as it cannot provide a better solution e.g. the branch is "bounded", also called cut.

Figure 3.11: Branch-and-bound algorithm components.

both computation and memory requirements. DF/IHS was found to provide near-optimal solutions.

3.5 Genetic Algorithms

Another approach of finding a suitable schedule is to use a genetic algorithm (GA). GA's are search heuristic algorithms designed to "evolve" a solution through operations inspired by Darwinian evolution. A GA uses a concept of *chromosomes* which represent possible solutions to the specified problem. These chromosomes are assigned a *fitness* value which represents how good a particular solution is. This fitness value is determined using a fitness function which applies heuristics to estimate a fitness score. This fitness function is of vital importance to the GA since a more accurate estimation of a solution suitability will lead to a better selection phase. Using these chromosomes in combination with evolutionary operations such as *selection*, *crossover* and *mutation* will resemble the natural process of "survival of the fittest". With this evolutionary process, the fittest individuals are selected for reproduction and will produce offspring which inherits characteristics of the parent chromosomes. When considering this approach for the scheduling problem, one individual represents a solution e.g. a complete schedule. Some of this offspring must then be randomly mutated in order to find new improved solutions which are added to the population of chromosomes available for reproduction. This process will then be repeated until an adequate solution has been found, e.g. when the offspring produced during several iterations does not provide any gains in fitness or the process is manually terminated [47]. A simple GA typically consists of the steps defined in Figure 3.12 [29]:
Applying GAs to the scheduling problem can be conducted in several ways. The two main approaches are to either use the GA in combination with traditional list scheduling algorithms or to use it to perform the actual mapping of tasks to processors [59]. A GA can for instance be used to evolve the task priority values later to be used in a list scheduling algorithm as suggested in a paper by Dhodh [16].

An example of a GA algorithm which does produce the schedule is proposed by Hou et al. [29]. In this approach, the authors apply a stochastic search based GA where each individual chromosome consists of several lists with each list representing the internal order of computational tasks for each processing unit. This has the advantage of eliminating the need to consider the precedence constraints between different processors. This algorithm utilizes the crossover operator to exchange tasks between different processors while the mutation operator is used for task ordering within the chromosome lists.

1. **Initialization** – First step is to initialize a population of randomly generated problem solutions (chromosomes).
2. **Evaluation** – Evaluate each individual chromosomes using the fitness function applied in the GA.
3. **Genetic operations** – Generate new chromosomes using previously mentioned genetic operations: selection, crossover and mutation.
4. **Repeat from step 2 suitable solution has been found**.

Figure 3.12: Basic genetic algorithm components.

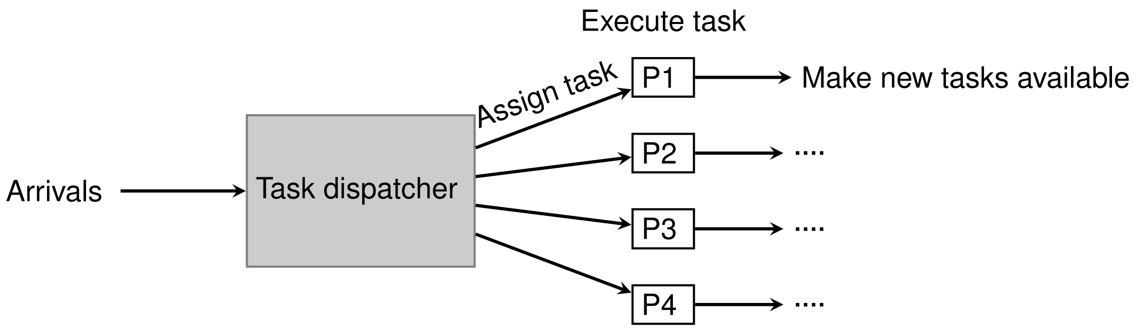


Figure 3.13: Illustration of a dynamic scheduler with centralized allocation.

3.6 Dynamic Scheduling

In contrast to static scheduling, in which all tasks are scheduled before processors start to execute, dynamic scheduling can or will only make a few assumptions regarding the nature of the tasks to execute. A dynamic scheduler will instead have to rely on real time information about the state of the system. This information will become available to the scheduler during the actual execution of the program, e.g. the scheduler will only have access to a limited number of tasks at a time [40, 2, 49]. This means that only the tasks that are ready to execute at the current state of the system is considered for scheduling.

The primary goal of the dynamic approach is to achieve maximum load balancing with the current available information at a certain time which implies that the scheduler has to employ an allocation strategy. In the literature there are two major approaches, centralized or distributed strategy. In a centralized approach there is one single entity which is dedicated to gather information about the current system state and determining which task gets allocated to an available processor [49]. This includes determining priority for the tasks which are currently available for scheduling. Such a model is illustrated in Figure 3.13.

As an alternative to using a centralized model one can also employ a distributed model where all processing units have the ability to communicate regarding the state of the system. In order to efficiently determine task allocation in a distributed approach, a certain policy of how to exchange information must be present [2]. Examples of distributed algorithms are the *bidding* algorithm and the *drafting algorithm*.

A consequence of applying a dynamic strategy is that this would require more computation than the static algorithms but will often generate a better schedule. This will however introduce a secondary goal to the dynamic algorithms to minimize the scheduling time besides minimizing the schedule makespan [40].

3.7 Related Works

An extensive amount of research has been conducted on the area of task scheduling over many years and it is a well known problem in parallel processing. In scheduling there are several different parameters which can affect how the scheduling was conducted. Consequently, there exist a large amount of algorithms which target different scenarios. Due to the large number of algorithms there is an extensive amount of comparative studies addressing different performance parameters [40, 27]. Since this thesis will partly be a comparative study it will share some aspects with other such works and articles. Studies of this nature are numerous and a selected set of similar studies is presented. However, these studies are most often performed in simulated environments which is the main difference to this thesis.

For instance, Hagras and Janeček [27] share several similarities with this thesis. In this article the authors compare two different standard types of list scheduling, static and dynamic, and compare the results with the aid of a random graph generator to provide test scenarios. However, this is also a fully simulated environment with perfect knowledge of task execution time and other factors. The conclusion from this article was that dynamic list scheduling produced better schedules than static list scheduling algorithms. However, it is the recommendation of the authors to prefer static list scheduling over dynamic, the reason being the higher algorithm complexity in dynamic list scheduling algorithms needed to produce a schedule.

Another comparative study, [38] by Kwok and Ahmad who are very prominent researchers in the field and are co-authors of many well cited studies concerning scheduling. In this study the authors compare algorithms on the DAG task graph scheduling problem. This extensive study does, besides comparing performance of certain algorithms, also try to determine ***why*** one algorithm performs better than an other. Besides measuring schedule length, the authors also propose a performance metric called *scheduling scalability* which attempts to estimate the overall solution quality. This study does however examine a class of algorithms that are not relevant to this thesis. Conclusions from this study were that dynamic list scheduling algorithms generally performs better than static ones, though they can cause longer scheduling overhead due to higher complexity. Other findings conclude that insertion based algorithms are better and that a simple algorithm such as ISH, which is used in this thesis, can yield significant performance. Lastly the authors conclude that critical path based algorithms do, in general, yield better performance than other algorithms.

From the same authors is an extensive study [40], in which a large amount of algorithms and classes of algorithms are addressed. This study was one of the main sources when researching scheduling and has had a large influence on this thesis. In this study the authors acknowledge the fact that algorithms are based on a diverse set of assumptions and are consequently hard to describe in a unified manner. As a result, a taxonomy is suggested in which algorithms can be sorted according to their functionality. Examples are, *unbounded number of clusters* (UNC) which is a class of algorithms that assumes an unlimited number of processors, and conversely the *bounded number of processors* (BNP) class assumes a limited number of processors. In addition, there are algorithms that consider a fully general model, where the system is assumed to consist of an arbitrary network topology where communication costs are not negligible, these are defined as the *arbitrary processor network category* (APN) class. This means that APN algorithms also schedules messages on the network in addition to tasks. Lastly we have the class called *task-duplication based* (TDB) algorithms, which make similar assumptions as APN. TDB algorithms will schedule tasks redundantly to multiple processors with the goal of reducing the communication costs overhead. Algorithms used in this thesis can be sorted in the BNP category.

Another related study [15], produced by Davidovi'c and Crainic, acknowledges the absence of established sets for comparing and analyzing heuristic scheduling algorithms. In this study, the authors propose new evaluation sets for use on similar scenarios applied in this thesis. These test-problem instances are produced using the task graph generators which are defined by the authors and are compared to other available sets such as these developed by Kasahara [34], which is used in this thesis. The authors concludes in this work that when communication delays are introduced the performance of scheduling algorithms based on heuristics significantly decline.

An important procedure in any build system is the compile phase and optimizing this has long been a priority Baalbergen for instance, describes in his article [7] how to parallelize the standard *gnu make* build system. In this work, Baalbergen introduces *pmake* which is intended to be used in a multiprocessor environment. This thesis applies similar assumptions as this, like that the operating system has a processor allocation strategy. Similarly, this work also implements "virtual processors" which controls the execution of an individual task. The conducted experiments showed that considerable speedup could be obtained by parallel compilation. However, the authors could also conclude that linear speed-up would most likely not be achievable due to the overheads produced by the task distribution process.

Also several works involving a parallelization of the compiler itself which is also an alternative executing the build tasks in parallel [25].

4 Method

This chapter contains a detailed explanation of the work process and is written in chronological order. The method is divided in three different phases as sections which represent states in the thesis which had different goals. The choices made during the thesis will also be motivated in this chapter but whether these assessments and choices proved to be correct will however be discussed in Chapter 5.

4.1 Phase 1: Pre-study And Prototyping

The initial phase of the thesis was used to study the field of build systems and how to optimize them. This also involved studying parallel processing and task scheduling. Later in this phase, an early prototype solution in form of a scheduling simulator would also be developed isolated from the target build system.

Pre-study

A pre-study was conducted in the early stages of the thesis with the purpose of determining the appropriate approach to optimizing the build procedure. Even though a parallel build was the primary alternative to speeding up the build procedure, several other alternatives were also considered if a parallel build would not be possible. An alternative could for instance be to minimize the number of redundant build tasks in each build procedure or other build avoidance techniques mentioned by Smith in his book [53].

The first step was to identify which aspect of the building procedure to improve, and with this information to determine which improvements to implement. The optimization of the build procedure was theorized to be bound by either I/O, memory or CPU usage and a simple experiment was conducted to identify which factor would be the first to limit optimization attempts. This test was performed by initiating a build while measuring the parameters mentioned above and was conducted using the task manager and resource monitor which are both default tools included in the Windows operating system. The computer specifications used during the thesis work were the following:

CPU	**RAM**	**Harddrive**
Model: Intel(R) i7-7700K	Model: N/A	Model: SSD
Speed: 4.20 GHz	Speed: 2133 MHz	Speed: N/A
Physical cores: 4	Size: 32 GB	Size: 500 GB
Logical cores: 8		

The results from this test showed that the CPU was using, on average, 12 % of the total CPU power during the sequential execution. However this value may not be entirely accurate as the computer utilized hyper-threading which means that the number of logical processors is double the number of physical processors. Since it is only the physical processors that can actually execute instructions, the total CPU utilization from physical processors should be approximately double the measured value. The I/O usage was very low and hardly rose above 5% and this can be explained by the fact that an SSD hard-drive was used which makes I/O operations very efficient. The memory usage was steady between 8-10% of the total physical memory. To investigate how these values would scale with two build instances, two identical build processes was initialized simultaneously and with the same values measured. This second test provided similar values on I/O while slightly increasing memory usage and the total CPU usage practically doubled.

The conclusion from this experiment was that in order improve performance, the focus should be to utilize the CPU to a greater extent. Since the build machine has a multi-core processing architecture and the build procedure is already decomposed into smaller tasks that are currently executed in sequential order, the most effective solution would be to transform the build to execute in parallel. The most basic approach would be to divide the tasks evenly among all available cores to be executed in parallel. This would however not be possible since some tasks depend on the output from other tasks. Accurate knowledge of task dependencies are consequently an important requirement to be able implement a parallel build. Otherwise, if the build system has faulty or inadequate dependencies, two files might be compiled in the wrong order which could result in a broken build.

To solve this problem one will have to introduce a scheduling algorithm which can achieve an efficient distribution of tasks while still respecting precedence constraints. There are several aspects which have to be considered when choosing a suitable algorithm for this specific problem:

Exact Or Approximation Algorithm

The scheduling problem is a well known NP-complete problem and finding the optimal schedule for every build procedure will consequently not be a fast calculation. It is however possible to achieve the optimal schedule by modeling the problem as a linear optimization problem and then solve it using a ILP or constraint solver. The optimal solution can also be found using some form of exact algorithm but this was considered to be outside the scope of this thesis. But a more suitable solution might be to apply some algorithm which finds a near-optimal schedule. The reasoning for this being a better solution is that an exact algorithm might require a lot more time to find the optimal schedule than might be gained over a faster approximation planning.

Heterogeneous Or Homogeneous Tasks

The Configura build system currently both compiles and tests the software in the same procedure. Several setup commands are also needed before the build procedure can take place. These will however not be included in the parallel execution since the number of setup commands are not numerous enough to benefit from parallelization. However, tasks will still be heterogeneous since test tasks consists of CM-code stubs and compile tasks are shell commands.

Preemptive vs. Non-preemptive

Many algorithms consider a preemptive approach where the scheduler can interrupt tasks and insert other ones at any time. In a non-preemptive algorithm, a task must be allowed to finish before starting the execution of the next task. While certain tasks, like program tests, might be able to be performed using preemption it is only non-preemptive scheduling that will be considered in this thesis since the tasks include compilation tasks which cannot be interrupted and later resumed.

Static Or Dynamic Approach

Both the static and dynamic approach are possible in the current system. A static approach is possible due to all dependencies being mapped before the build system begins to execute and together with a suitable profiler of individual tasks, execution times can be estimated as well. Since these parameters are known the most viable solution would be to use static algorithms. However, implementing a dynamic algorithm was also deemed plausible but was the secondary choice after static algorithms.

Simulation With Random Graph Generator

After the approach to transform the build procedure to execute in parallel was confirmed to be viable a scheduling simulator was to be created and be used as a prototype to the real implementation. This prototype was to be used to obtain a better understanding of scheduling theory and find a good implementation to task-nodes and the DAG structure in general in C#. Further, the simulator was to provide a platform on which the focus was to test the correctness of the priority calculations and the scheduling procedure in a controlled environment. This would distance the initial implementation from unexpected errors such as compilation error and faulty task weight estimations.

The expected benefit with this prototype was to experiment with the implementation of the scheduler and as a consequence avoid a bad implementation in the Build Central. Further benefits were to observe if a certain algorithm behaved as intended, which would be easier to control in such an environment where standard graphs were easy to implement and would also provide a first impression of the effectiveness of an algorithm. This first impression included the ability of testing a certain algorithm in many different scenarios and draw conclusions of good and bad qualities of the algorithm.

Tasks were generated with a randomized execution time to simulate a workload of that task and the execution would be simulated by letting the worker sleep for the assigned duration. Several values could be regulated in order to simulate and evaluate how effective the algorithm was on larger graphs compared to Configura's system, for future usage. Values that could be regulated:

- Max graph levels.
- Max and min graph width.
- The chance of new edges between old node to the newly generated.
- Max and min task workload.

Determine Initial Algorithms

The last part of phase 1 was to select an initial set of algorithms to be implemented in the simulator. The primary focus of the chosen algorithms were to minimize the total length of the schedule, e.g. the schedule *makespan*. The motivation for this being that the developers care about the result from a completed build rather than any specific part of it, which would be the case when minimizing the number of late jobs for instance. A number of algorithms were considered to solve the problem at hand but the algorithms based on list scheduling were assessed to be the most suitable for the initial implementation. This was due to the large amount of scientific literature produced on that subject, with a large range of different algorithms to choose from which can be observed in Chapter 3. List scheduling algorithms also seemed to have lower complexity and to be easier to implement in general. Since the goal was to utilize the CPU in a multi-core shared memory architecture, the algorithms which included communication costs as part of the calculation were neglected and the costs were assigned zero.

The first algorithm to be considered as the established list scheduling algorithm *highest level first with estimated time* (HLFET), described more in depth in Section 3.3. This algorithm was chosen due its simplicity and does in short prioritize the SL of tasks which tends to belong to a critical path of a DAG. This would hopefully provide some early results and feedback on the initial implementation.

The second algorithm to be implemented was a dynamic list scheduling algorithm called *earliest time first* (ETF), for details see Section 3.3 [30, 27]. The purpose of this choice was also that the algorithm was relatively simple but offered a little different approach than HLFET by the fact that in ETF, the scheduler will continuously recalculate priority values in contrast to HLFET which applies static list scheduling. Further, the ETF algorithm desires high load balancing and will always prioritize the tasks that can be started first at any time. Implementing these two algorithm was primarily to facilitate the development of the DAG and task structures.

4.2 Phase 2: Implementation

The second phase of the thesis focused on transferring the scheduler implementation from the prototype to Build Central and dealing with the problems introduced by building actual tasks in parallel and not just simulating task execution.

Migrate Prototype To Build Central

Before proceeding by implementing more algorithms, the existing prototype was to be implemented in Build Central. This first involved getting acquainted with the existing build system implementation and change certain aspects of the system to suit a parallel execution of tasks. This includes for instance, implementing mechanics for awaiting the completion of all parent tasks, single worker implementation representing one processor core, scheduler implementation and other aspects that might cause synchronization issues. Many of these features could be transferred directly from the prototype to the target system with some minor modifications. The purpose of migrating the scheduling prototype early was to hopefully discover potential problems early in development. Such problems was theorized to possibly include:

- Tasks attempting to modify disk files simultaneously which would generate build errors.
- Task execution time being heavily unreliable.
- Circular dependency error.

The expectation was that this step would go smoothly due to the implementation of the prototype. The assessment was that the code structure of tasks and task graph would almost be directly transferable between the two systems. How individual task execution would be initiated had to be changed since threads were only directed to sleep for a duration according to the task weight in the simulator. This is obviously not the case in the build system where tasks have to started differently depending on if the task is a compilation or test. Beyond implementing the primary structure of the graph and constructing the task-nodes from the existing build tasks there were other aspects to implement. One such element was the estimation of the task execution time which would in contrast to the simulator not be entirely correct. The first attempt of such estimation was to simply take the runtime data from each task in the previous build and use it as task weight in the next build.

Testing With Subset Build

The next step in incorporating the prototype to the build system was to create a stable subset of the program to be built. The purpose was to avoid updates in the code from Configura's developers which could have possibly caused errors in the code. This would make it hard to determine if the errors occurred due to problems with the parallel build or if a developer made a mistake. To further increase the reliability of this particular build configuration, all build tasks in the subset were compile tasks. The reason to exclude test tasks were to focus on correctly building the DAG-model from the configuration and achieve continuous reliable builds, also there were some initial concerns regarding running tests in parallel. Further, this would make analysis of the build system easier and provide insights on aspects not covered in the prototype. In order to improve the ability to analyze the structure of the application to be built by the build system, the nodes and dependencies were converted to an image, see Figure 4.1. These observations were also intended to be used to improve the simulator so that it would properly emulate the actual build procedure. Two important conclusions from running the subset build were:

- A few bottleneck tasks would restrict parallelization.
- Large variation in task execution time.

The fact that task execution time was not reliable and could sporadically double in time execution made the total schedule time hard to predict. The impact from these unreliable predictions would have to be mitigated, explained further in the next subsection. The fact that a small number of tasks had long execution time as well as having many tasks depend on their output relates well to Amdahl's law which is briefly explained in Section 2.4. This means that the potential speedup will be limited by these bottleneck tasks and should therefor be highly prioritized.

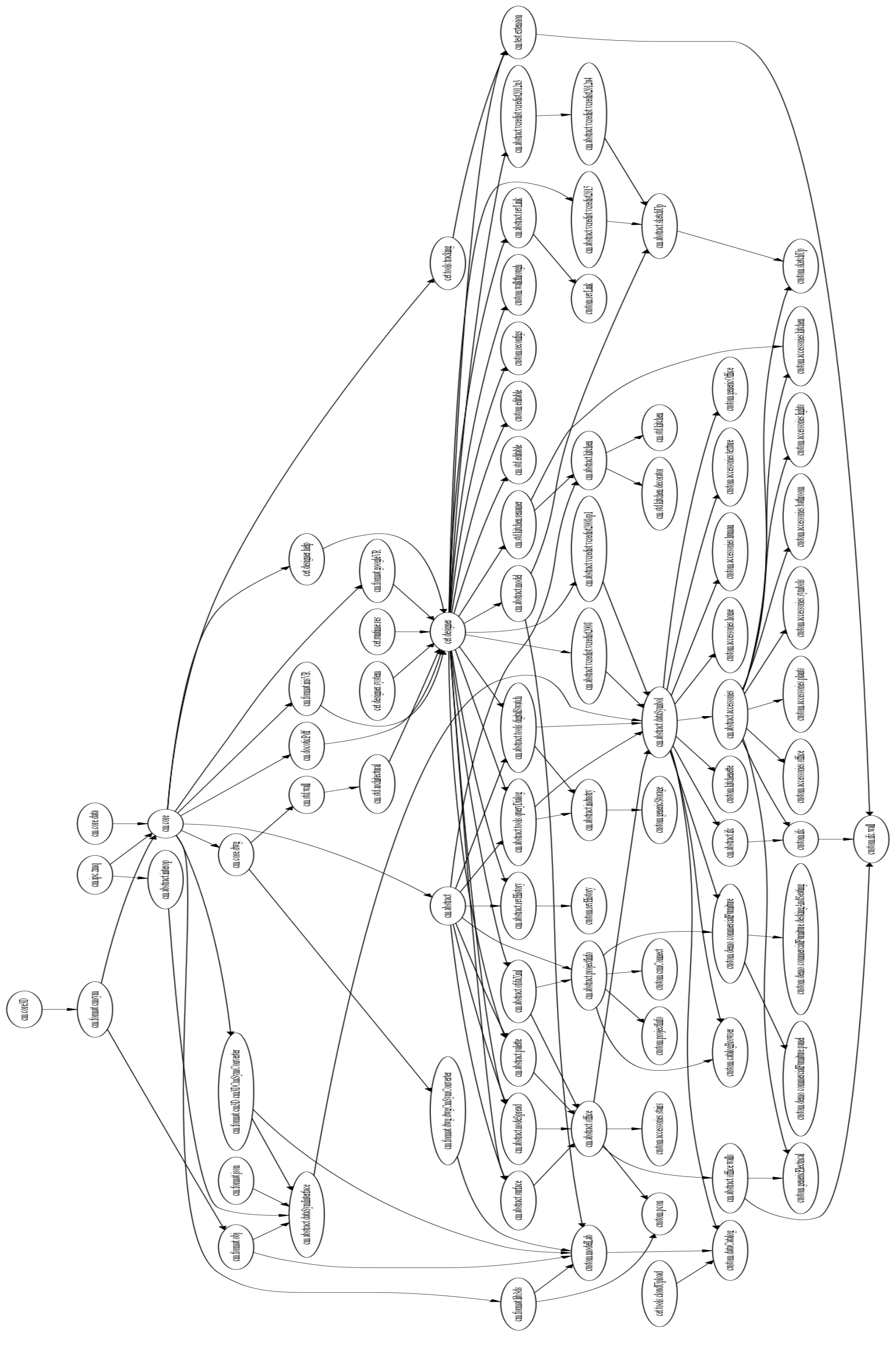


Figure 4.1: Subset of the build procedure represented as a DAG.

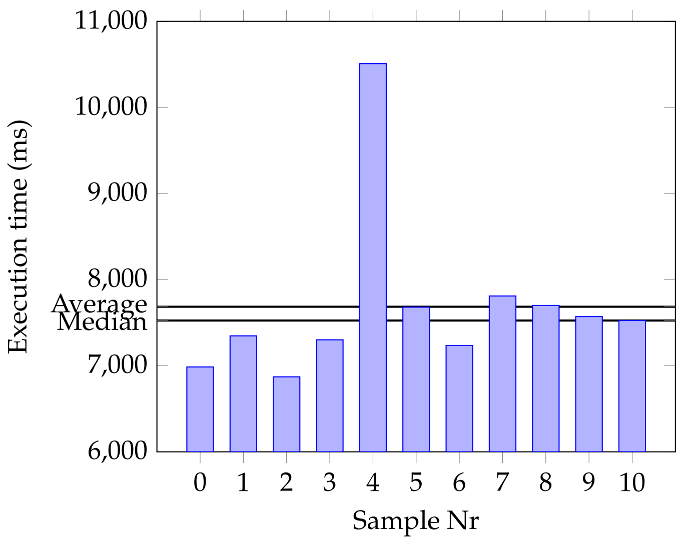


Figure 4.2: Task execution samples of a single task.

Task Execution Time Estimation

Following the initial results and with the observation that task execution times were highly inconsistent, with the consequence of the scheduled time not corresponding to the actual time it takes to execute the entire schedule. Most often this would have a negative impact and in order to make the scheduler more consistent the prediction calculation would have to be changed or improved. The previous technique was, as mentioned in Section 3.4, to save the execution time from the latest build and use that value as prediction for the next build. If we examine Figure 4.2, we can observe that this would not be an accurate prediction for most cases.

There exist a number of different approaches to profiling tasks run time, several are described in the Section 3.2. Implementing a sophisticated profiler to predict the future execution cost would be a time consuming project and not guaranteed to get close to an exact prediction. Instead the primary approach was expanded to get a good enough estimation. The current estimation was expanded by saving samples from each individual tasks build time from the ten latest builds in an XML-document and using the average value of these samples as the prediction value, see Figure 4.2. The results from this expanded prediction calculation were a great improvement on the first approach but the predicted value was still not accurate enough. This was due to the samples that had a significantly higher value than other samples which caused the average value to increase. Tasks would often have a stable execution time (as observed in Figure 4.2) but task execution would at some builds sporadically spike and be far from the usual value. This resulted in the average value not being a good prediction as the goal was to schedule according to the normal case. This could however be easily corrected with the current approach by simply taking the median value instead of the average value from the samples as prediction and number of samples increased to eleven in order order to get a middle value.

4.3 Phase 3: Improvement And Evaluation

In the last phase of this thesis project, the existing solution with the conclusions on the nature of the system would provide the basis for which improvements were to be made. This consisted of expanding the current solutions and creating an alternative solution to the list scheduling algorithms in form of dynamic scheduler. Further, a set of evaluation scenarios were developed in order to compare the algorithms in different situations. Also a optimal ILP solver would be used in order to find the best solution to the evaluation scenarios.

Extending List Scheduling Algorithms

With two list scheduling algorithms implemented the next step was to try and improve performance further. The choice was made to continue on the list scheduling implementation and find improvements to the HLFET and ETF algorithms. One such algorithm was *Insertion scheduling heuristic* (ISH) which basically is an enhanced version of the HLFET algorithm which adds the ability to utilize idle slots in the schedule, this is explained in Section 3.3. The motivation for this is that the bottleneck tasks caused other workers to wait for a long for that task to finish due to the nature of the HLFET algorithm.

A supposed improvement of the ETF algorithm according to some literature [27, 38] was the *dynamic level scheduling* (DLS) algorithm which introduces a new priority attribute to produce a better schedule, see Section 3.3.

Dynamic Scheduler

The motivation to implement a dynamic scheduler was due to the inconsistent execution times of tasks which cause delays in the produced schedules from static schedulers. These delays caused the actual execution time of the schedule to be much longer than the calculated makespan, indicating the negative effect of these delays. A dynamic scheduler would not be affected by inaccurate estimations to the same extent as the list scheduling algorithms since tasks would arrive and be made available to the scheduler during run time, as described in Section 3.6 and illustrated in Figure 3.13. The chosen approach was to use the centralized method and dynamically assign tasks to processors. This approach would require additional overhead caused by conducting task assignment during runtime but was theorized to have a positive effect on the total execution time.

The first attempt was to utilize the same principle as ETF but as a dynamic scheduler instead. Tasks were assigned to processors basically according to the FIFO principle, i.e. first tasks to arrive to the scheduler are the first ones to be processed. This algorithm will be referred to as *dynamic earliest time first* (DETF) further on in this thesis. The implementation of the task distributor for DETF was similar to the algorithm illustrated in Figure 4.3 except for step 3 which will just take the first task that was made available, similar to the FIFO principle. The task processor implementation is identical for both algorithms and is illustrated in Figure 4.4.

The second dynamic scheduler was used in a similar way but would instead combine the dynamic "on-the-fly" task assignment with certain static attributes to provide some values to calculate priorities. This scheduler would use the number of imminent successor tasks with dependencies to a task as the primary priority and break ties with the static level value of the task. Using this allocation priority ordering of available tasks, the scheduler was theorized to provide high load balancing because prioritizing the tasks with the most successor tasks would make more tasks available early and keep the processors occupied. The purpose of using the static level as the secondary priority value was to process the task which is more important to the critical path if two or more tasks had an equal number of imminent successors. This algorithm will be referred to as *dynamic most imminent successors first* (DMISF) further on in this thesis. The centralized task distributor for DMISF was implemented with the following algorithm shown in Figure 4.3.

```
1 Calculate the static b-level of each task.
2 Construct an initial list with entry tasks.
3 Select the task from the list with most imminent successors.
4 Assign the selected task to a random idle processor from a list
  with available processors.
5 Wait for new tasks and processor/processors to become available.
6 Repeat procedure from 3.
```

Figure 4.3: Dynamic task distributor for DMISF

```
1 Wait for task assignment.
2 Execute the assigned task.
3 Check if any descendant tasks can be made available for the dis-
  tributor.
4 Announce to the distributor that this processor is idle.
5 Repeat from 1.
```

Figure 4.4: Dynamic task processor

Optimal Solver

In addition to the implemented list scheduling algorithms and the dynamic schedulers, an attempt was also made of using some sort of optimization solver to find the optimal schedule. Having access to the optimal solution would be a good reference when evaluating the heuristic algorithms. In order to use an ILP solver to receive the optimal makespan the problem first had be modeled as a linear optimization problem:

Sets:

- T – A set of tasks.
- P – A set of processors.
- R – A set of ranks, rank will represent the internal order of tasks on a specific machine.

Parameters:

- w_t – Estimated execution time for task t.
- $d_{t,t_2} - \begin{cases} 1, if\ task\ t_2\ depends\ on\ task\ t \\ 0, else \end{cases}$

Variables:

- $s_{t,p,r}$ – Start time of task t on processor p with rank r.
- $f_{t,p,r}$ – Finish time of task t on processor p with rank r.
- m – Makespan, will be equivalent to the task with highest finish time
- $a_{t,m,r} - \begin{cases} 1, if\ task\ t\ is\ assigned\ to\ machine\ m\ with\ rank\ r \\ 0, else \end{cases}$

Constraints:

- 4.1b – Each task must be done on one machine.
- 4.1c – Each machine has one task on every rank.
- 4.1d – Tasks may not start before all parents are done.
- 4.1e – Task finish time is equal to task start time plus its computation cost.
- 4.1f – Makespan is equal to highest task finish time.

$$\begin{aligned}
\underset{t,p}{\text{minimize}} \quad & m && && \text{(4.1a)}\\
\text{subject to} \quad & \sum_{p\in P}\sum_{r\in R} a_{t,p,r} = 1, && \forall t \in T, && \text{(4.1b)}\\
& \sum_{t\in T} a_{t,p,r} \leqslant 1, && \forall p \in P, \forall r \in R, && \text{(4.1c)}\\
& s_{t,p,r} \geqslant f_{t_2,p,r} \times d_{t,t_2}, && \forall t \in T, \forall t_2 \in T, \forall r \in R, \forall p \in P, && \text{(4.1d)}\\
& f_{t,p,r} = s_{t,p,r} + w_t, && \forall t \in T, \forall r \in R, \forall p \in P, && \text{(4.1e)}\\
& f_{t,p,r} \leqslant m, && \forall t \in T, \forall r \in R, \forall p \in P && \text{(4.1f)}
\end{aligned}$$

This optimization problem then had to be implemented and integrated into the scheduler using third-party software that could operate on optimization problems. There exist a large number of optimization software which would be suitable for this approach, examples are GLPK [23], Gurobi [26], CPLEX [14], Microsoft solver foundation [46]. However, several of these require an additional license in order to use an unlimited model. The choice was to use the standard GNU GLPK framework, which could also be included directly to Visual Studio. This was partly due to the convenience of implementing the model in the same language as the build system, but also to avoid license issues. Lastly, this approach was not expected to provide an valid method for the build system but was rather a tool to get the optimal schedule length and compare the result with the other implemented algorithms.

Collecting Data

When comparing and evaluating the efficiency of the implemented algorithms one will need relevant data concerning each build. Several data parameters need to be collected:

- Number of cores used.
- Which algorithm was used.
- Makespan (estimated total execution time by the scheduler).
- Actual execution time.

These parameters were collected during a build and saved in a XML-file in order to have access to previous build results. Since the task execution time proved to be unreliable, build results had to be obtained for each algorithm several times in order to determine the average build result. This would in turn make the evaluation procedure more reliable. For each scenario described in the section below, each algorithm would perform a batch of five identical builds which would be used to provide an average value for both the makespan and the total execution time. When evaluating the different algorithms it would also be useful to have access to the optimal schedules in order to examine how well the estimation algorithms perform.

Evaluation

Since the desired outcome of the thesis is to lower the total execution time of the build procedure it will primarily be the schedule makespan that will be evaluated. However other aspects will also be considered with regards to future usage of the build system. The complexity of the algorithms might for instance be an important factor, should the number of tasks increase. The algorithms are compared based on their performance on different subsets of the build procedure, all of which are of varying sizes with a different composition of tasks. Additionally, these algorithms are also examined with the developed simulator in order to analyze if there are any performance differences between a simulated environment and a practical one. This would produce good indications for how much the task execution time estimation affects the effectiveness of the algorithms.

In order to obtain robust data on which to base the evaluation, four different scenarios were developed to test the algorithms. The set of scenarios consisted of two builds from Configura's build system and two were obtained by using standardized task graphs for evaluation developed by Kasahara [33, 58], which are often used in similar studies [15]. This was partly to evaluate if the build system should dynamically change which algorithm to use depending on the properties of the build or if one method could be used universally. The four scenarios are described briefly below:

Scenario 1: 85 Tasks

This is the same build which was used in an earlier phase where test tasks were excluded, see Figure 4.1 to observe the produced graph. The purpose was to evaluate which algorithm performed best when having small builds with heavy bottleneck tasks.

Scenario 2: 262 Tasks

This build was also part of the Configura build scenario, together with scenario 1, but had a higher number of build tasks and now also included tests. The purpose of this scenario was to observe how the build system performed when having both compilation tasks and test tasks in the same build. The build consisted of one third test tasks. This scenario was deemed to be the most important scenario as Configura is currently working on including more tests in their continuous builds.

Scenario 3: 500 Tasks

The following scenarios were used as an alternative to the scenarios obtained from the target build system. The purpose was partly to get a more universal and fair evaluation of the algorithms. But the main reason was to evaluate the algorithms effectiveness on larger graphs than the Configura program can currently produce. Observing the results from this scenario would give some insights into which algorithm would be best suited when Configura's software scales in size.

Scenario 4: 1250 Tasks

Practically the same purpose as the Scenario 3, described above but with an even larger graph. This scenario would also be used to examine how algorithms would scale with larger task sets. It would also be used to evaluate if there were any differences between static and dynamic algorithms in a standard task graph compared to a software product task graph.

5 Results And Discussion

The following chapter will present the acquired results from conducting the scenarios described in the evaluation phase of the method, see Section 4.3. The presented result data will also be discussed and compared to the expected results. Additionally, this chapter will also include a discussion concerning the thesis method and evaluate which aspects that could have been conducted differently. Lastly, eventual flaws in the chosen method will be explored.

5.1 Results

We begin this chapter by presenting the results from conducting the constructed scenarios in Section 4.3. The result data from the four cases will be presented in numerical order. Additional results not directly related to these evaluation cases will be discussed later in this section as these aspects did not produce any quantitative results. This includes the impact of irregular task execution time and computation complexity between the algorithms and these will be discussed qualitatively.

Also, the results from the initial experiment in the method will not be presented as these results were presented in the Chapter 4 due to the information being necessary for understanding certain decisions in the method. The results from the evaluation phase will be presented case by case, beginning with the scenarios which involve the target build system with actual build tasks followed by scenarios conducted with the simulator. The results from each scenario will initially be evaluated individually and later summarized and discussed. Table 5.1 is presented so that the reader can get an idea of the improvements compared to the initial state of the system after reading the results.

The results in the two first scenarios will be presented using two graphs, one where the dynamic algorithms are excluded and one where the two best performing list scheduling algorithms and the dynamic algorithms are presented together. The reason for splitting up the algorithms is to better represent gradual progress from the decisions made during the implementation. This split represents the milestones well, with the list scheduling algorithms implemented first and the dynamic schedulers added after evaluating the performance of first algorithms. Notice that the presented results are the combined time of the scheduling and the execution of the schedule. The time required to schedule all tasks will however not be presented since this parameter had a very small impact on the total time, less than 1%.

The last two scenarios will however have a single graph to present algorithm performance but will also include a graph to illustrate the difference in required scheduling time of each list scheduling algorithm. Scheduling time still have little to no impact on the scenarios used in this thesis but is primarily to show the different algorithms' complexity and how this could theoretically impact the total scheduling time and possibly be a factor in other use cases.

Scenario	Sequential time (min:s)
1	13:35
2	38:56
3	8:38
4	21:48

Table 5.1: Sequential execution times for each scenario.

Scenario 1: 85 Tasks (No Tests)

This scenario was, as explained in the Chapter 4, designed to provide simple and reliable results early on in the thesis. This subset build setup served this purpose well and it was very useful to the initial success to have a stable starting platform. This allowed for continuous work on necessary elements of the implementation. It was also during evaluation of this case that the discovery of bottleneck tasks and the unreliable task execution times was made. The fact that this was discovered early allowed for appropriate actions to be taken in order to minimize disruptions.

We will begin by examining the results of the implemented list scheduling algorithms in Figure 5.1. We can observe a clear advantage for the ETF and DLS which were both dynamic list scheduling algorithms. The primary reason for this is believed to be the large variation in task execution times between tasks which is not the case in most literature regarding list scheduling. HLFET prioritizes the tasks belonging to the critical path and, since differences in task weight might be high, causes such tasks to be prioritized over constant load balance. The same will apply for ISH, but was supposed to mitigate this effect. The results show however that ISH does not perform better anyway and this is probably because of the low task count which lowers the number of options to insert into idle slots. With the best speedup result, 2.28 using 4 cores and ETF algorithm, the total execution time was 5 minutes and 34 seconds, which can be compared to the original time requirements in Table 5.1.

Moving on to Figure 5.2, which now includes the dynamic algorithms. The results from these algorithms were surprising as the overhead produced by the real time allocation phase were expected to make this approach inferior to the static algorithms. However, as mentioned in Section 4.3, the purpose of the dynamic algorithms was supposed to mitigate the effect from stochastic task execution times. This appears to have given more benefits than the dynamic allocation overhead caused latency. We can observe that the DETF (which was based on ETF) obtained almost identical performance to ETF. The DMISF algorithm however proved to outperform the DETF, ETF and DLS.

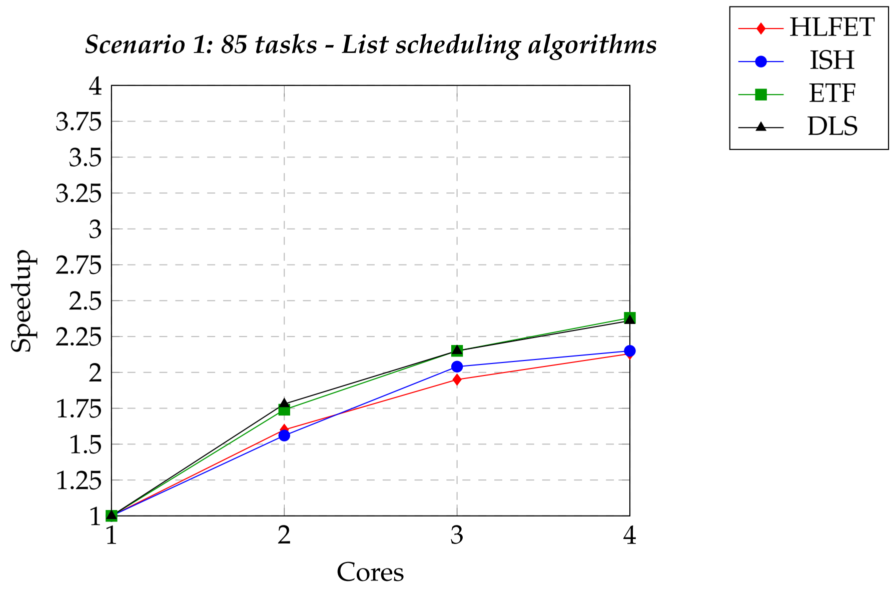


Figure 5.1: Scenario 1 speedup results from list scheduling.

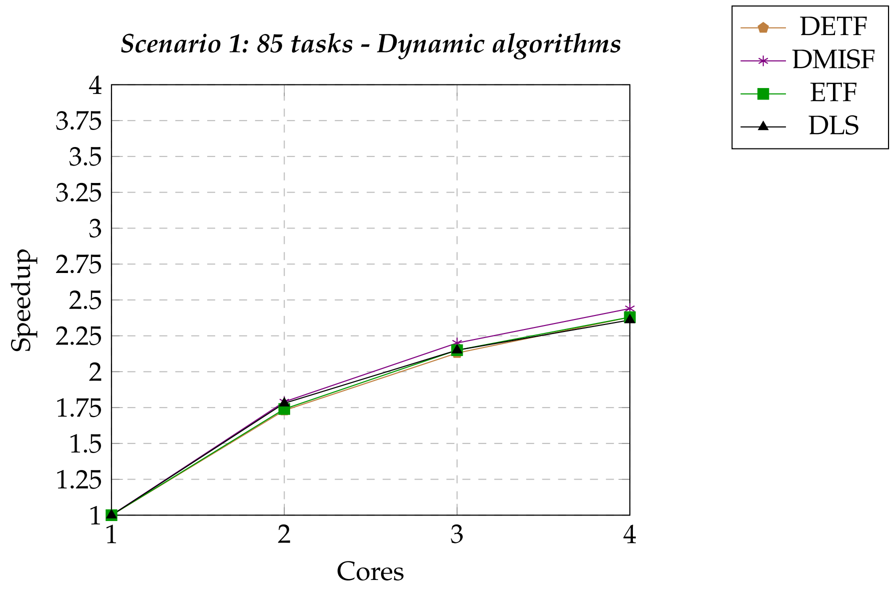


Figure 5.2: Scenario 1 speedup results, including the two best list scheduling and both dynamic algorithms .

Scenario 2: 265 Tasks (Including Tests)

This scenario can be seen as an ordinary build setup to be processed by Configura's continuous builder on a regular basis and will contain a similar number of tasks with the same balance between compilation and tests. As such, this case will have the largest impact in determining which algorithm was the best fit for Configura's build system. This case will be presented in the same manner as the previous one, with Figure 5.3 containing the results of the list scheduling algorithms and Figure 5.4 presenting the two best list scheduling algorithms compared with the dynamic algorithms.

We start by observing Figure 5.3 and the results from the static algorithms. First we can notice that the speedup is considerably higher in this case. This is considered to be due to test tasks having no or only a few number of dependencies which make them easy to schedule while no other important tasks are available, filling idle slots with such tasks. The test tasks did not rely on using any of the output produced by build tasks but instead run code dynamically and did therefor not need to use built libraries.

Just as in the previous case, ETF provides best measured performance. However, in this case, DLS fell a bit behind ETF in term of performance which was not expected since results from the literature suggest that DLS usually performs better [27]. Notably, ISH performed far better in this setup and is not far from having the same performance as ETF. This suggests that the conclusion from the previous case (where ISH performed badly because of low task count) was probably correct. The fact that ETF again had the best performance suggests that prioritizing available tasks over the critical path was beneficial in this environment setup.

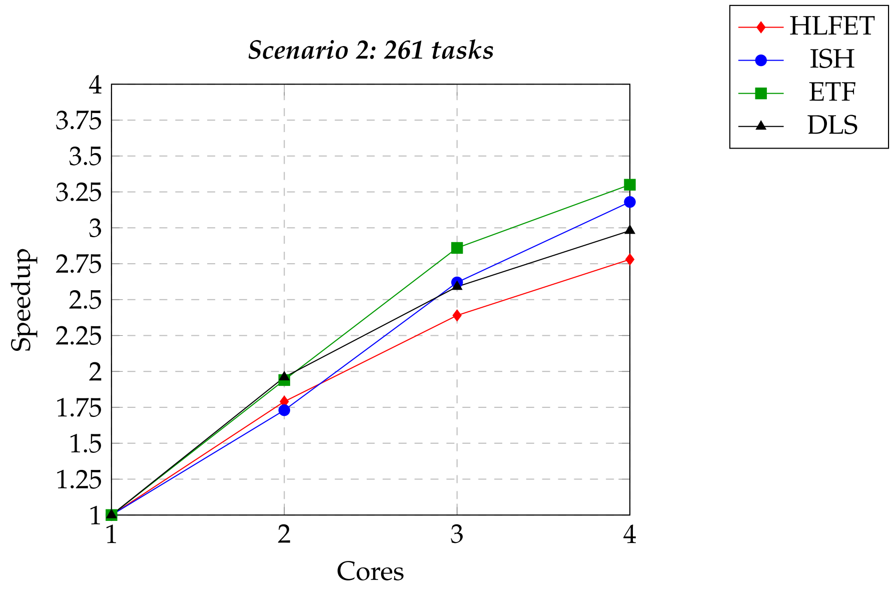


Figure 5.3: Scenario 2 speedup results from list scheduling algorithms.

We continue and now examine Figure 5.4 and the results from the dynamic algorithms. In this case, DETF did not perform as well as it did in the previous scenario and was the worst performing algorithm in this graph. However, the DMISF algorithm produced the best performance again, performing better than ETF. This suggests that the DMISF algorithm be the best current solution to optimizing the build procedure as it had the best performance in both scenarios based on the real build system.

The total execution time when using the DMISF algorithm, with a speedup of 3.43, was 11 minutes and 19 seconds, which can be compared to the sequential time presented in Table 5.1. The impact from having higher complexity in the scheduling phase do not seem to have a large impact on the total execution time. We keep in mind that the results presented in these graphs contain the total execution time from the entire procedure, i.e., both scheduling time and build time.

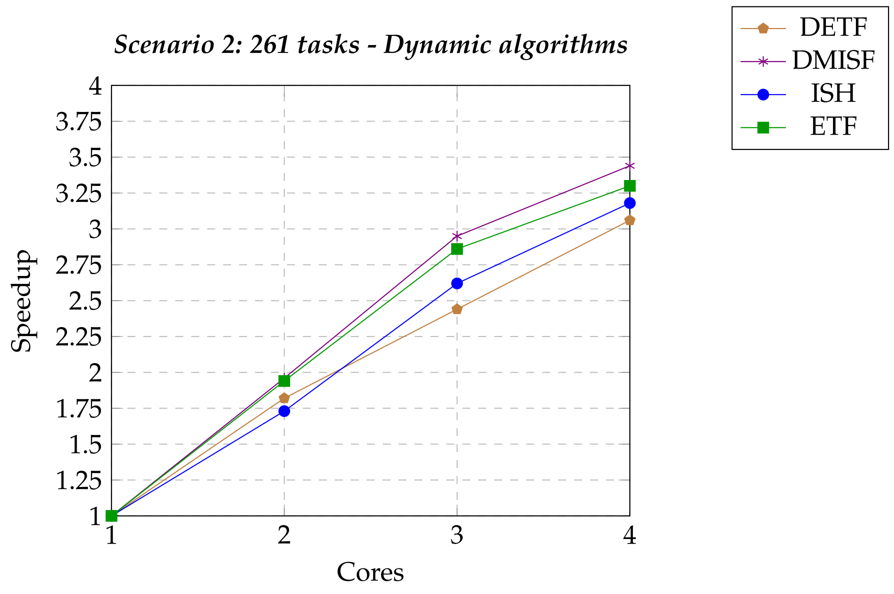


Figure 5.4: Scenario 3 speedup results, including the best list scheduling algorithm and the dynamic algorithms.

Number of tasks	500
Parallelism	3.923194
Critical path length	1315
Optimal schedule length, p=2	2580
Optimal schedule length, p=3	N/A
Optimal schedule length, p=4	1328-1379

Table 5.2: Scenario 3 - Graph properties.

Scenario 3: 500 Tasks (Standard Task Graph)

The following two cases were used obtain data on the algorithms performance on a larger sets of tasks. These results would provide a basis on which to make conclusions of the general performance of the algorithms and make predictions on how the algorithms would perform if Configuras build sets would increase in size. The speedup results from the following two cases will be presented using a single graph instead of splitting it into to two as in the previous cases. However, the time required to schedule these tasks will also be presented to illustrate the impact of the complexity of the algorithms. This was due to the higher number of tasks which would cause a higher scheduling overhead than was observed in the previous scenarios. This task graph is provided by Kasahara's website which hosts a large amount of standard task graphs [33]. Important associated values which are provided on the website are presented in Table 5.2. Parallelism represents the extent which the task graph can be parallelized and is calculated using the sum of all task processing times divided by critical path length. p is the number of processors. Unfortunately the optimal schedules for p=3 are not available.

We start by observing the measured speedup from all the implemented algorithms in Figure 5.5. Similar to scenarios 1 and 2, the HLFET and DETF had the worst performance though not as bad as in the previous scenarios. The optimal speedup for this scenario (provided by Kasahara's website containing standard task graphs [33]), is ˜ 3.88 which make the best performing algorithms having close to optimal schedules for this particular case. Once again, the ETF and DLS algorithms provided the better results and the dynamic algorithms performed worse than in the previous scenarios. The reason for the decreased performance from the dynamic algorithms are most likely caused by smaller individual task size in this standardized task graph, causing the real-time scheduling overhead to have an higher impact.

Secondly, we will briefly analyze the scheduling time for each of the static schedulers, see Figure 5.6. We can determine that the time requirements for the ETF and DLS are much higher than the HLFET, just as described in the literature. However, even if these values may suggest that HLFET would have an advantage due to this, the scheduling time is still negligible when considering the total time of executing the schedule.

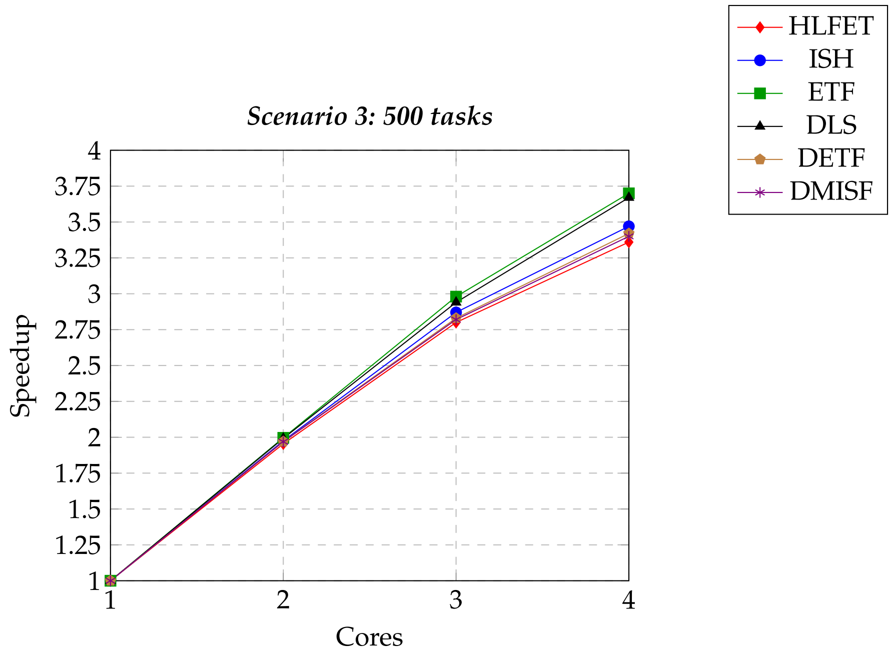


Figure 5.5: Scenario 3 speedup results.

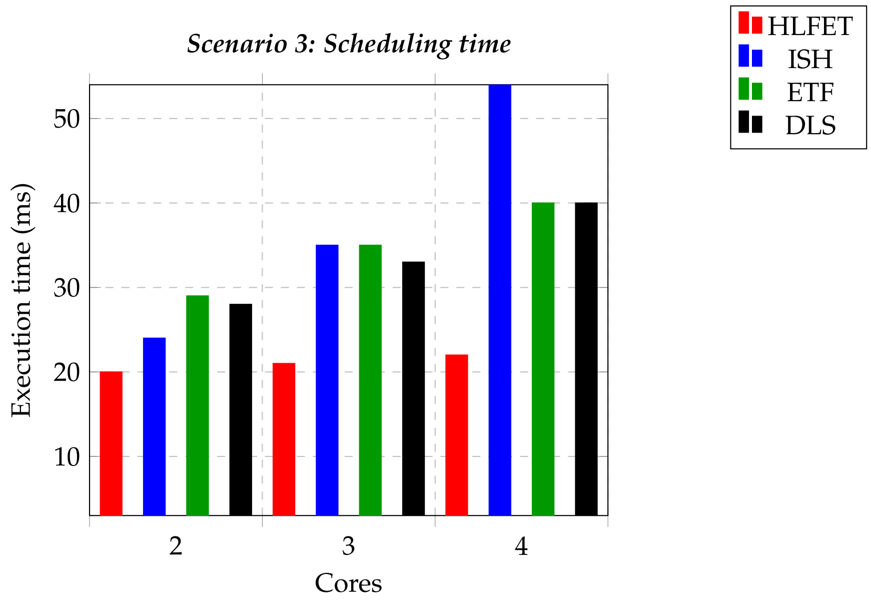


Figure 5.6: Scenario 3 - Scheduling overhead.

Number of tasks	1250
Parallelism	4.392845
Critical path length	2963
Optimal schedule length, p=2	6508
Optimal schedule length, p=3	N/A
Optimal schedule length, p=4	3254-3326

Table 5.3: Scenario 4 - Graph properties.

Scenario 4: 1250 Tasks (Standard Task Graph)

The last scenario were an extension to scenario 3 and was supposed to be used to verify the results in that scenario using a larger task set. The result will be presented in the same way as the previous scenario. This graph was also provided by Kasahara's website with standard task graphs and the graph properties are presented in Table 5.3 [33].

If we start by examining the results in Figure 5.7, we can observe that all algorithms performed fairly well in this scenario with not much difference in speedup between the algorithms. Since the optimal speedup in this task graph is ˜ 3.91, we can conclude the ETF and DLS algorithms are close to producing optimal schedules. We can also observe that Figure 5.7 is very similar to Figure 5.5, which would suggest that ETF and DLS had the best general performance. The fact that all algorithms provides good schedules would suggest a highly parallelizable task graph.

The scheduling overhead is presented in Figure 5.8 and can be observed to quite similar to the result in the previous scenario with the exception of the ISH algorithm, which had a much higher value in the previous scenario. We can see that the dynamic list scheduling algorithms required much more time.

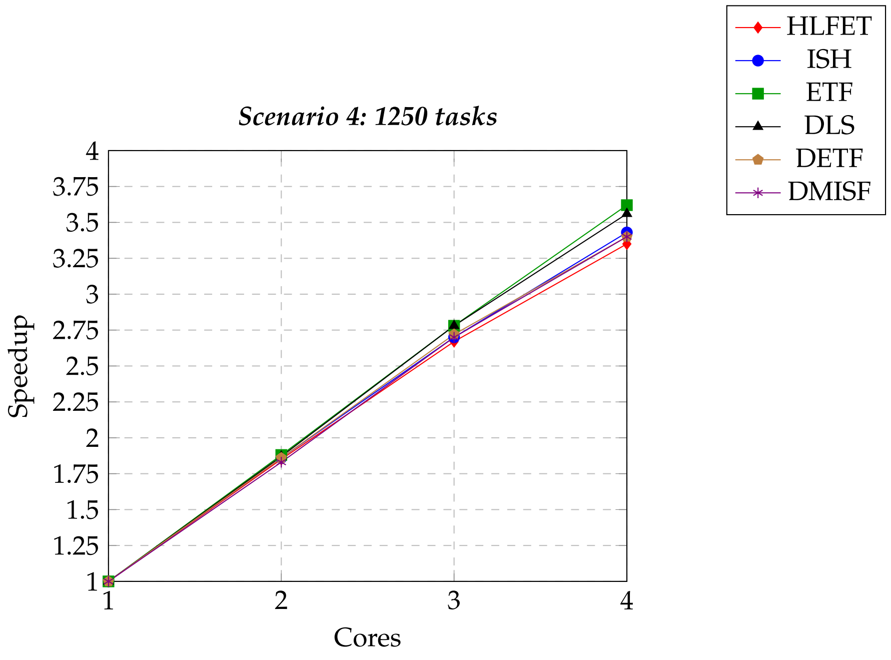


Figure 5.7: Scenario 4 speedup results.

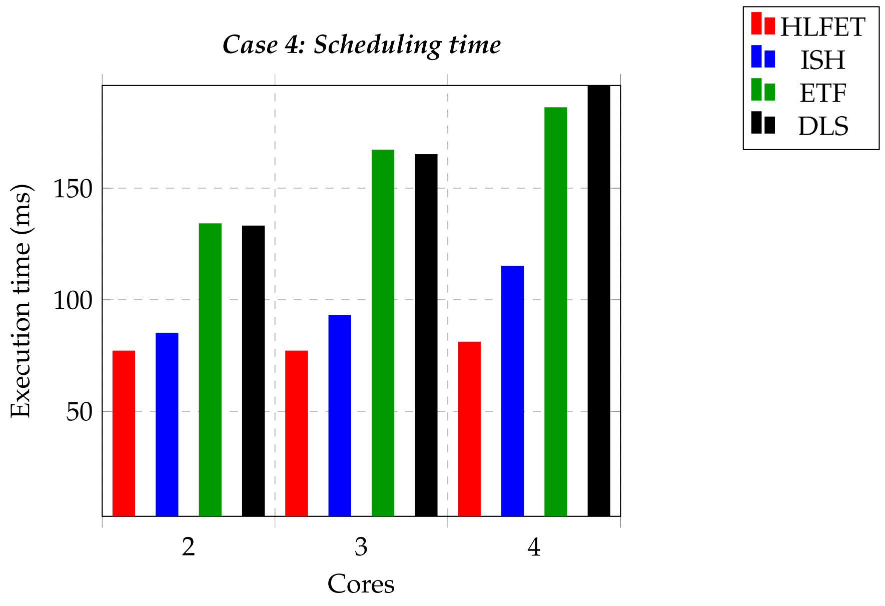


Figure 5.8: Scenario 4 speedup results.

Optimal Solver

The goal of using the optimal solver, described in Section 4.3, was to obtain the best possible solution for the two first cases and compare the implemented algorithms to this solution. The suggested model and solver was working well for smaller task sets (number of tasks less than 20) and was producing optimal schedules within 30 minutes. Unfortunately, as the graph size increased to more than 20 tasks, the solver had trouble to complete within several hours. This caused the decision to abandon this approach due to the time requirements for running scenarios 1 and 2 in the solver. One could introduce a time limit to the optimization solver but the produced schedule would most likely be suboptimal and the whole purpose of using the solver was to get the optimal solution. The causes of this behavior in not clear but it is possible that the problem was translated poorly to a optimization model, resulting in unnecessary branches being processed. Other causes can be that the GNU GLPK solver was not best available alternative and that another solver might have produced faster results.

These results further emphasized the difficulty of finding the best possible solution. These tests also concluded that even if the an optimal solution could be found, it would still not provide better performance than either the heuristic nor the dynamic algorithms.

Discussion And Expected Results

In this section we briefly discuss how the actual results differed from the expected results and the causes of these deviations. We will begin by looking at the overall results where the achieved speedup using four processors could reach up to 3.43 on the target build system, which was higher than first anticipated. The reasons for this would suggest low coupling between build tasks which made it easier to utilize CPU in such high degree. However, if we analyze the graph shown in Figure 4.1 we can observe that most tasks share a common parent early in the task graph which produces a bottleneck. This bottleneck effect can also be observed in the results for scenario 1 where the achieved speedup is far lower then in the following case due to this bottleneck task. The cause is previously analyzed in the results of scenario 2 with the conclusions that the tasks that performs tests can be used to fill schedule idle slots since most of the tests do not have dependencies.

The results from the algorithms were also somewhat unexpected if we compare with results obtained in other algorithm comparative studies. According to most such studies, static algorithms should have the advantage to dynamic ones when most runtime parameters are known. Yet the results did not indicate that this was the case for the setup used in this thesis. This can of course partly be explained by the fact that actual application tasks was processed with the build system and not just a set of simulated tasks which have more predictable time requirements. One can consequently argue that the task execution time parameters are not known before execution in the current setup since these proved to be highly irregular and very hard to predict accurately. This challenges the prediction of static algorithms being better suited than dynamic algorithms in this environment. This single aspect changes conditions on which these assumptions from the majority of studies on this subject are made.

Results were also not entirely unexpected since the proposed extensions were not selected without substantiated reasons. The choices were based on observations from previous builds and were always theorized to improve the performance. The common denominator are irregular task execution times which influenced the decision to implement dynamic algorithms which would later prove to be the best performing algorithm for the target build system.

Next, we will attempt to predict how the current implementation would scale if we were to increase the number of cores in the system. As mentioned earlier in this thesis, the number of cores used to evaluate the system is not optimal since the scalability and load balancing capacity of the algorithms cannot be fully observed, especially in scenario 3 and 4 where the speedup is almost linear. However, the results are not very likely to deviate from the current curve but without quantitative results, there is no way to know for sure. In theory, due to the existence of the bottleneck tasks, the speedup increase for each added core would continue to level out just as in Figure 5.3. Still it would have been interesting to examine the difference in scalability between dynamic and static algorithms. Intuitively we believe that the dynamic algorithms would continue to perform better than the list scheduling algorithm mostly due to the large task sizes and unreliable execution times. Would these aspects however be addressed and improved it is likely that the performance would become more even.

We can also examine the prediction that the overhead caused by higher complexity in the scheduling procedure would impact performance. This was simply not the case when running the scenarios. The differences in scheduling time requirements between static and dynamic list scheduling algorithms proved to have a negligible effect on the total execution time, especially when the number of tasks are small. However, one aspect that impacts the importance of this overhead is the size of each individual task. If tasks take a long or short time to execute will not impact the time required to schedule all tasks. This leads to scheduling time having higher importance in systems were tasks have a short execution time. Because the tasks in the target build system are relatively large, scheduling time was of little importance to performance. Not even in scenarios 3 and 4, which both had smaller tasks, was the scheduling time any major factor although it had a larger impact than in the target system.

5.2 Method Discussion

In this section, the general method will be discussed, analyzed and criticized.

Simulator

We begin by analyzing the decision of implementing a simulator prior to implementing any improvements to the target system, introduced in Section 4.1. The purpose with the simulator was achieved by allowing the initial work to be strictly concentrated on understanding scheduling principles and testing the initial approach. With an isolated environment the debugging became easier and faster since no other unexpected elements could impact the behavior of the scheduler, making the feedback loop short. This consequently provided a smooth transition to the target system since the setup was implemented similarly and most pitfalls were already solved in the simulator scheduler.

One negative aspect of using this approach was that it postponed the discovery of the unreliable task execution times in the target system. This provided a shorter time frame of implementing a solution to this problem. This consequently results in less time to perform a literature survey of possible solutions, testing different approaches and implementing the chosen solution. In conclusion, knowledge of not having reliable task execution estimation would have been beneficiary to have earlier in the thesis.

One can also argue that implementing a simulator required time that could have been spent on implementing the algorithms directly in the target system. Though this could potentially have the effect of lower code quality with more bugs which would have been harder to identify and the total required time would possible exceed that of the chosen approach. Overall the decision of implementing a scheduling simulator was positive.

Algorithm Choice

Next we discuss the algorithm choices made in this thesis. As mentioned in Section 4.1, the purpose with the initial algorithms was to use well established list scheduling algorithms in order to achieve early results. The initial algorithms were HLFET and ETF, described in Section 3.3 and 3.3 respectively. Combined with the approach of using a simulator was positive since these results provided some initial feedback on the implementation. These first algorithms operated of simple principles which was a good starting point for learning basic scheduling theory. Another motivation for using a simpler algorithm was because the current setup did not require any advanced features to solve the assignment. Potentially the performance gains from implementing more advanced features and algorithms would not be high enough to motivate such a decision.

Since the initial algorithms was working quite well, the decision was made to implement two additional algorithms which were based on the same principles but more advanced nature, see Section 4.3. This might have been a wrong decision since it did not provide any new approach and this time might have been better utilized to examine a different approach and perhaps a more modern one. It would potentially have been interesting to compare if more modern algorithm would perform better than older ones in the setup used in the thesis. A particularly interesting alternative to these algorithms might have been to examine the possibility to implement algorithm(s) more specialized towards distributed computing which was a secondary objective in the thesis. This was however dependent on certain hardware additions that was not available at the time. This approach would for instance require additional computers which were not available at the time and would also require effort on the communication elements needed in such algorithms. One could also argue that the size of the task graphs processed by the target system were to small to have benefited from distributed computation. This was probably the case, but it would still have been interesting to collect data to confirm this hypothesis.

The final two algorithms which were used in the performance comparison was the dynamic algorithms, see Section 4.3. The decision to implement these came relatively late in thesis and were somewhat improvised, not following any specific algorithm. This could be argued to be suboptimal as no previous solution could be compared with. However, these algorithms was tailor made for the target system and also provided the best results in practical scenarios which made the comparison interesting nevertheless. The second of the two dynamic algorithms was especially interesting since it combined statical attributes with dynamic ones which in this case also yielded good results.

Lastly, the usage of a optimal solver to find the best possible solution will be briefly discussed, introduced in Section 4.3. As mentioned, this approach was never intended to provide a potential practical solution but would rather provide data for the discussion. This approach was far more time consuming than first anticipated and implementation was started late in the thesis. Additionally there were some licensing issues when using solvers which further prolonged progress on this approach. Consequently, this solution was unpolished and may not have been functioned properly. Setting up the problem as mathematical optimal model was educational but overall provided a very small contribution to the thesis.

Execution Time Estimation

We continue and discuss the solution to the estimation of task execution time, described in Section 4.2. Now, providing the algorithms with estimations of the time required to process each task proved to have a greater impact than first anticipated. As described above, the discovery of this problem showed up later on the thesis due to the implementation of the scheduling simulator, where such an issue were not present.

The implemented solution was basic, using the median value of previous task execution times to provide an estimation. This approach is rather primitive but provided good results in comparison to using the last execution time as an estimate. More advanced approaches were assessed to not provide enough result in relation to the amount of effort required to implement them. This was probably the correct decision since the direct causes of the execution time anomalies was not entirely known and defined, making it difficult to apply the correct method to address it.

Evaluation

Now, we will discuss the way that the algorithm comparison was conducted, see Section 4.3. One of the major remarks that can be discussed are that the algorithms were only evaluated on two, three and four processors. Evaluating with more processing elements would clearly have been advantageous since this would have provided more robust data on the scalability of the algorithms. Unfortunately the required hardware could not be provided in time. Additionally, it would have been interesting to evaluate the algorithms in an environment where communication costs were present. This would add additional aspects which could affect performance and provide more data to analyze.

The judgment of the set of scenarios which were used to collect data was overall good, providing performance data on the both a general case and a average builds from the target system. This made it possible to make a more accurate estimation of the suitability of the algorithms. Both the scenarios extracted from the target system represented two average cases well and was to provide interesting data. Evaluating the general performance by utilizing standard tasks graphs from Kasahara's website was advantageous, since it provided legitimacy to the results and made it easy to determine if the results were reasonable [33]. Evaluating on both standardized task graphs and system specific scenarios were a good decision since these illustrated the different qualities of both the static and dynamic algorithms.

Source Criticism

The number of available sources of this research field was of an abundant quantity and is also a field which have been analyzed for a long time. Since the basic principles surrounding scheduling has remained fairly similar directed the research focus towards robust and well established sources. Consequently, most of the sources are fairly old. The literature study has been greatly influenced by the work of *Kwok* and *Ishfaq*, especially the extensive comparative studies [38, 40]. These works provided a complete introduction to scheduling theory and is cited in most studies regarding scheduling in multiprocessing systems.

Perhaps a minor critic of the sources used are that most of main sources are dated from before the year 2000. This might suggest that the methods used are out of date. However, most studies in this field build upon similar basic principles and rather adds new methodologies to existing. Other alternatives which are based on different principles, like genetic algorithms and branch-and-bound algorithms, were as explained earlier in Chapter 3, too complicated for the defined problem and would most likely not have been worth the effort to implement it. Another minor issue concerning the sources was the absence of works which targeted actual application task graphs. Such task graphs proved to be a bit different from task graphs used in most studies which disregarded many aspects of the problem. Consequently it would have been useful to have a more solid literature background regarding such cases and possible complications. The general impression was that overall, the sources used were of good quality and trustworthy.

6 Conclusion

This thesis presents a study of improving the performance of a CI build system. In this chapter we will discuss to which extent the research questions could be answered and if the overall goal was achieved.

6.1 Answering Research Questions

Research Question 1:

The overall goal of this work was to at least double the original build speed and we can with certainty conclude that this goal was achieved as the speedup in scenario 2 would reach up to 3.43 with the DMISF algorithm and four cores enabled. This scenario was designed to emulate a typical build on the build machine and required approximately 40 minutes to execute. If we make the assumption that the dedicated build machine was running constantly during work hours (8 hours), the total amount of builds that would be performed each day is approximately 12. The improvements provided by the work in this thesis will increase this figure to an estimated 40 builds each day. This will save significant time for developers waiting for build results on their latest code submission and allow the software code base to be tested more often.

Research Question 2:

According to most of the studied material in this thesis, the static algorithms should have the advantage over dynamic algorithms in general. These conclusions could be confirmed by comparing the results from the chosen algorithms using the scheduling simulator in Section 5.1 and 5.1. These studies were however based upon certain assumptions that neglected the effect of the individual task build time varying between builds. This would prove to effect the implemented algorithms differently where the dynamic algorithms was less affected by this aspects. The dynamic algorithms would even surpass the static algorithms in performance on the target build system.

6.2 Future Work

As discussed in Chapter 5, the applied algorithms were not particularly modern. This could be a further step of improvement, especially if Configura wants to examine the possibility developing a distributed build system. A distributed system would demand work to handle the increased synchronization requirements and also introduces elements such as communication delays which would have to be considered by the algorithms [2]. The current implemented algorithms can be modified if needed to take into account the communication delays. However, these algorithms might not be the best suitable for such a system due to their age and there would most exist more modern and specialized algorithms. Since a distributed system has a higher efficiency when applied to larger work graphs and that the work required to implement such a system would be extensive, it is concluded that this solution currently is not worth the effort. Although an alternative would be to have a distributed system where each machine handles an individual build and each source code commit initiates a new build which is inserted to a job queue which the next available machine undertakes.

An area that could potentially increase system performance is the unreliable task execution time, covered in Section 4.2. Improving this aspect of the system might make the static algorithms produce more reliable schedules and possibly change the current result to coincide with the results achieved in the literature. A possible aspect to investigate is Configura's CM compiler and try to establish what causes these unstable build times. An alternative is to improve the current solution to a more sophisticated time estimation. Making improvements to task execution time estimation is also deemed to not be worth the effort since it would probably require a lot of time to investigate and the potential performance gains would not be large enough to motivate this endeavor.

Another alternative would be to attempt to improve the implemented algorithms, DETF and DMISF. Configura could investigate the possibility to develop better dynamic algorithms or modify the current to be further customized for their specific system. The fact that these algorithms are currently having the best performance for Configura's system speaks for exploring this approach.

The main target for improving the current system was identified to be the load balancing capabilities of the system, especially when Configura will add more processing cores to their current build machine. To achieve better core load balancing, the main aspect to address is the uneven size of the build tasks since there are currently a couple of tasks that acts as a bottlenecks in the schedule. These tasks often appears early in the dependency graph and many tasks often have dependencies to these tasks which causes idle time, see Figure 4.1 in Section 4.2 for an illustration. This is particularly the case when performing smaller builds, see Section 5.1. Several of these tasks should be fairly easy to break into smaller packages, since they are often tools containing core functionality grouped together. Further, breaking such bottleneck tasks into smaller pieces would, although create more dependencies to take into consideration, have additional benefits in reducing build time due to packages being grouped together which might result in unnecessary packages being built. Putting work into these improvements should be considered since it most likely be valuable, especially if the number of tasks continues to scale up.

Bibliography

[1] Thomas L Adam, K. Mani Chandy, and JR Dickson. "A comparison of list schedules for parallel processing systems". In: *Communications of the ACM* 17.12 (1974), pp. 685–690.

[2] Ishfaq Ahmad and Arif Ghafoor. "Semi-distributed load balancing for massively parallel multicomputer systems". In: *IEEE Transactions on Software Engineering* 17.10 (1991), pp. 987–1004.

[3] Ashfaque Ahmed and Bhanu Prasad. *Foundations of Software Engineering*. CRC Press, 2016.

[4] Shameen Akhter and Jason Roberts. *Multi-core programming*. Vol. 33. Intel press Hillsboro, 2006.

[5] Gene M Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities". In: *Proceedings of the April 18-20, 1967, spring joint computer conference*. ACM. 1967, pp. 483–485.

[6] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, et al. *The landscape of parallel computing research: A view from berkeley*. Tech. rep. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[7] Erik H. Baalbergen. "Design and implementation of parallel make". In: *Computing Systems* 1.2 (1988), pp. 135–158.

[8] Blaise Barney. "What is parallel computing". In: *Introduction to Parallel Computing* (2012).

[9] Peter Brucker and P Brucker. *Scheduling algorithms*. Vol. 3. Springer, 2007.

[10] Artem M Chirkin, Adam SZ Belloum, Sergey V Kovalchuk, Marc X Makkes, Mikhail A Melnik, Alexander A Visheratin, and Denis A Nasonov. "Execution time estimation for workflow scheduling". In: *Future Generation Computer Systems* (2017).

[11] Jens Clausen. "Branch and bound algorithms-principles and examples". In: *Department of Computer Science, University of Copenhagen* (1999), pp. 1–30.

[12] Edward G Coffman and Ronald L Graham. "Optimal scheduling for two-processor systems". In: *Acta informatica* 1.3 (1972), pp. 200–213.

[13] Richard Walter Conway, William L Maxwell, and Louis W Miller. *Theory of scheduling*. Courier Corporation, 2003.

[14] *Cplex*. URL: `https://www.ibm.com/analytics/data-science/prescriptive-analytics/cplex-optimizer` (visited on 02/30/2018).

[15] Tatjana Davidović and Teodor Gabriel Crainic. "Benchmark-problem instances for static scheduling of task graphs with communication delays on homogeneous multi-processor systems". In: *Computers & operations research* 33.8 (2006), pp. 2155–2177.

[16] Muhammad K Dhodhi and Imtiaz Ahmad. "A multiprocessor scheduling scheme using problem-space genetic algorithms". In: *Evolutionary Computation, 1995., IEEE International Conference on*. Vol. 1. IEEE. 1995, p. 214.

[17] Paul M Duvall. *Continuous integration*. Pearson Education India, 2007.

[18] Stuart I Feldman. "Make—A program for maintaining computer programs". In: *Software: Practice and experience* 9.4 (1979), pp. 255–265.

[19] Michael J Flynn. "Some computer organizations and their effectiveness". In: *IEEE transactions on computers* 100.9 (1972), pp. 948–960.

[20] Martin Fowler and Matthew Foemmel. "Continuous integration". In: *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf* (2006), p. 122.

[21] Richard F Freund. "Optimal selection theory for superconcurrency". In: *Supercomputing, 1989. Supercomputing'89. Proceedings of the 1989 ACM/IEEE Conference on*. IEEE. 1989, pp. 699–703.

[22] Richard Gibbons. "A historical application profiler for use by parallel schedulers". In: *Job scheduling strategies for parallel processing*. Springer. 1997, pp. 58–77.

[23] *GLPK*. URL: `https://www.gnu.org/software/glpk/` (visited on 02/30/2018).

[24] Ronald L Graham. "Bounds for certain multiprocessing anomalies". In: *Bell Labs Technical Journal* 45.9 (1966), pp. 1563–1581.

[25] Thomas Gross, Angelika Sobel, and Markus Zolg. "Parallel compilation for a parallel machine". In: *ACM SIGPLAN Notices*. Vol. 24. 7. ACM. 1989, pp. 91–100.

[26] *Gurobi*. URL: `http://www.gurobi.com/` (visited on 02/30/2018).

[27] Tarek Hagras and J Janeček. "Static vs. dynamic list-scheduling performance comparison". In: *Acta Polytechnica* 43.6 (2003).

[28] Mark D Hill and Michael R Marty. "Amdahl's law in the multicore era". In: *Computer* 41.7 (2008).

[29] Edwin SH Hou, Nirwan Ansari, and Hong Ren. "A genetic algorithm for multiprocessor scheduling". In: *IEEE Transactions on parallel and distributed systems* 5.2 (1994), pp. 113–120.

[30] Jing-Jang Hwang, Yuan-Chieh Chow, Frank D Anger, and Chung-Yee Lee. "Scheduling precedence graphs in systems with interprocessor communication times". In: *SIAM Journal on Computing* 18.2 (1989), pp. 244–257.

[31] Michael A Iverson, Fusun Ozguner, and Lee C Potter. "Statistical prediction of task execution times through analytic benchmarking for scheduling in a heterogeneous environment". In: *Heterogeneous Computing Workshop, 1999.(HCW'99) Proceedings. Eighth*. IEEE. 1999, pp. 99–111.

[32] *Jenkins*. 2017. URL: `https://jenkins.io/`.

[33] *Kasahara*. URL: `http://www.kasahara.cs.waseda.ac.jp/schedule/index.html` (visited on 02/30/2018).

[34] Hironori Kasahara and Seinosuke Narita. "Practical multiprocessor scheduling algorithms for efficient parallel processing". In: *IEEE Transactions on Computers* 33.11 (1984), pp. 1023–1029.

[35] James E Kelley Jr. "Critical-path planning and scheduling: Mathematical basis". In: *Operations research* 9.3 (1961), pp. 296–320.

[36] Boontee Kruatrachue. "Static task scheduling and grain packing in parallel processing systems". PhD thesis. 1987.

[37] Gary Kumfert and Tom Epperly. *Software in the DOE: The Hidden Overhead of"The Build"*. Tech. rep. Lawrence Livermore National Lab., CA (US), 2002.

[38] 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.

[39] Yu-Kwong Kwok and Ishfaq Ahmad. "Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors". In: *IEEE transactions on parallel and distributed systems* 7.5 (1996), pp. 506–521.

[40] Yu-Kwong Kwok and Ishfaq Ahmad. "Static scheduling algorithms for allocating directed task graphs to multiprocessors". In: *ACM Computing Surveys (CSUR)* 31.4 (1999), pp. 406–471.

[41] Ailsa H Land and Alison G Doig. "An automatic method of solving discrete programming problems". In: *Econometrica: Journal of the Econometric Society* (1960), pp. 497–520.

[42] Jan Karel Lenstra, AHG Rinnooy Kan, and Peter Brucker. "Complexity of machine scheduling problems". In: *Annals of discrete mathematics* 1 (1977), pp. 343–362.

[43] Jan Karel Lenstra and AHG Rinnooy Kan. "Complexity of scheduling under precedence constraints". In: *Operations Research* 26.1 (1978), pp. 22–35.

[44] Joseph YT Leung. *Handbook of scheduling: algorithms, models, and performance analysis*. CRC Press, 2004.

[45] John L Manferdelli, Naga K Govindaraju, and Chris Crall. "Challenges and opportunities in many-core computing". In: *Proceedings of the IEEE* 96.5 (2008), pp. 808–815.

[46] *Microsoft Solver Foundation*. URL: `https://msdn.microsoft.com/en-us/library/ff524509(v=vs.93).aspx` (visited on 02/30/2018).

[47] Melanie Mitchell. *An introduction to genetic algorithms*. MIT press, 1998.

[48] Gordon E Moore et al. "Cramming more components onto integrated circuits". In: *Proceedings of the IEEE* 86.1 (1998), pp. 82–85.

[49] Lionel M. Ni and Kai Hwang. "Optimal load balancing in a multiple processor system with many job classes". In: *IEEE Transactions on Software Engineering* 5 (1985), pp. 491–496.

[50] Brian Reistad and David K Gifford. *Static dependent costs for estimating execution time*. Vol. 7. 3. ACM, 1994.

[51] Gilbert C Sih and Edward A Lee. "A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures". In: *IEEE transactions on Parallel and Distributed systems* 4.2 (1993), pp. 175–187.

[52] Steven S Skiena. *The algorithm design manual: Text*. Vol. 1. Springer Science & Business Media, 1998.

[53] Peter Smith. *Software build systems: principles and experience*. Addison-Wesley Professional, 2011.

[54] Warren Smith, Ian Foster, and Valerie Taylor. “Predicting application run times using historical information”. In: *Job Scheduling Strategies for Parallel Processing*. Springer. 1998, pp. 122–142.

[55] Daniel Ståhl and Jan Bosch. “Experienced benefits of continuous integration in industry software product development: A case study”. In: *The 12th iasted international conference on software engineering,(innsbruck, austria, 2013)*. 2013, pp. 736–743.

[56] Herb Sutter and James Larus. “Software and the concurrency revolution”. In: *Queue* 3.7 (2005), pp. 54–62.

[57] Krishnaiyan Thulasiraman and Madisetti NS Swamy. *Graphs: theory and algorithms*. John Wiley & Sons, 2011.

[58] Takao Tobita and Hironori Kasahara. “A standard task graph set for fair evaluation of multiprocessor scheduling algorithms”. In: *Journal of Scheduling* 5.5 (2002), pp. 379–394.

[59] Annie S Wu, Han Yu, Shiyuan Jin, K-C Lin, and Guy Schiavone. “An incremental genetic algorithm approach to multiprocessor scheduling”. In: *IEEE Transactions on parallel and distributed systems* 15.9 (2004), pp. 824–834.