UNIVERSITEIT VAN AMSTERDAM

June 2006

Master Thesis in Grid Computing:

Simulation for Scheduling Parameter Sweep Tasks on Global Grids

ALEXANDROS BAKAROS

Supervisor: Adam Belloum

Abstract

Parallel and distributed systems are now days moving towards to the concept of Grid computing. The globalization of dynamic and heterogeneous resources connected via Internet and shared by different users is now days a reality. The new power that rises gave the ability to scientists to turn into type of studies that was before for them a utopia. Parameter sweep studies are such a type of study. This thesis will try to investigate the impact that will have the consideration of communication cost within a parameter sweep task workflow. For that reason a scheduling algorithm will be analyzed and modified.

Contents

Chapter 1

Introduction

1.1 Workflow Paradigm

A simple scientific experiment can be easily described by a 3 step procedure. The first step is the data collection (e.g. from remote sensors). Second step is the data filtering (e.g. via data process centers) and the last step is the result visualization (e.g. via a visualization cluster). Someone may think that this operation is a repeated 3 task procedure with the one task need to follow the other in the given order. We can illustrate this simple procedure using a pipeline:

Data Collection(1) → Data Filtering(2) → Result Visualization(3)

The steps (tasks) 1, 2 and 3 are now considered as a single application. Each of these 3 tasks consists of various numbers of subtasks. The next figure presents the paradigm of this simple **workflow**.

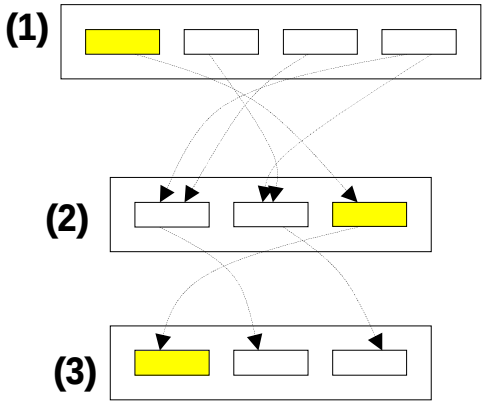


Figure 1.1: Scientific Workflow Example

The visualization of the result part (3) can start before all the subtasks that compose task (1) and (2) finish their execution. If a particular subtask in (1) finish executing then it is able to send its result to its successor subtask in (2) without having to wait the other

subtasks in (1) to finish their execution. The same can occur in (2) if a subtask obtains the results from its predecessor subtask(s) in (1). Considering this, it is easy to realize that there can be a more efficient way than the pipeline to define this workflow. The result will be a possible reduction of the total application execution length. This is the work for a scheduling algorithm. A scheduling algorithm assigns tasks to resources with a scope to minimize the total application execution length (makespan).

1.2 Parameter Studies – Parameter Sweep Tasks

Imagine a simulation that produces results for solving a scientific problem. By varying the initial conditions we can produce different results that correspond to dissimilar cases of the same problem. That is called "parameter space". In the past decades in order to solve partial differential equations as for example the study of fluid flow, scientists were able to use just one processor of a high speed computer that was probably in a super computer center placed locally. In terms of cost all the tasks were so expensive in computer cycles that these kinds of studies were ignored [1].

Nowadays the situation has changed and the parameter studies are performed by many scientists in various scientific fields. Parametric studies are now used in searching for extra-terrestrial intelligence [3] (SETI@HOME project), crash simulation, molecular modeling for drug design, human-genome sequence analysis, hoc network simulation brain activity analysis, high-energy physics events analysis, tomography, financial modeling, MCell simulations [4] and a lot of other scientific areas. For that reason, high-throughput parametric computing studies are nowadays considered as the killer application for the Grid, meaning that they are able to take a maximum advantage of the Grid capabilities. This was possible because of the fast development of Grid technology the last years. A scientist is now able to access supercomputers not just locally in his laboratory but is capable to do distributed computations using other supercomputers or clusters spread at different geographically areas of the planet and accessed via network links. The advantage of the global Grid technology makes it easier to integrate resources from distributed scientific computer centers with those of one's particular scientist environment, creating a technical basis and opening a field for complex parametric investigations [1][2].

Parameter sweep applications consist of a number of independent experiments "tasks" named parameter sweep tasks. Such an application is composed by a fixed number of layers (levels). From now on we will refer as T_i this type of task where (i) indicates the number of the level that the task belongs. Every task T_i consists from many smaller subtasks. There is no inner-task communication or any type of data dependencies between the same level subtasks but dependencies are likely to exist between different level subtasks. The only restriction is that the dependencies are allowed only between the subtasks that belong to i+1(children) or i-1(parent) layer, with subtasks that belong to i layer. Every subtask can take various files as an input and produces one or more files as an output which will be the input for the next level (T_{i+1}) successor subtask. Figures1.1

represents such a parameter sweep task T_i where the yellow circles symbolize the subtasks that this particular task includes. These subtasks by definition are independent each other.

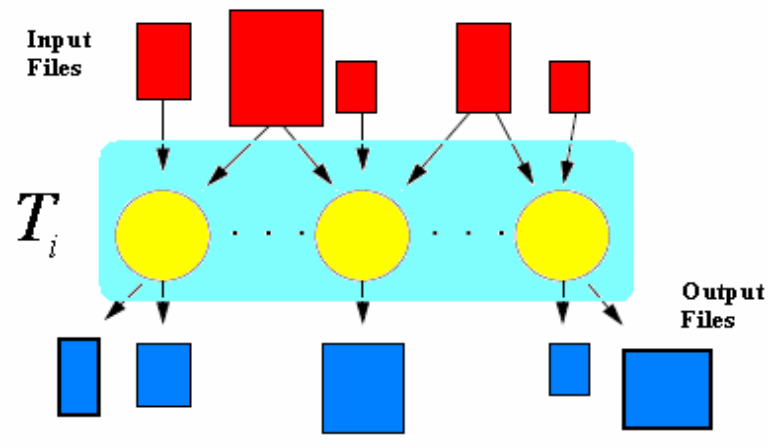


Figure 1.1: A paradigm of parameter sweep task.

1.3 Static – Dynamic Scheduling

A hierarchical taxonomy that represents algorithms for scheduling working units or else tasks in parallel and distributed computational systems is presented in [5]. Grid technology belongs to these kinds of systems and so we are able to say that Grid scheduling algorithms are part of this taxonomy. Local and Global algorithms is the distinction on the highest level of hierarchy. Local refers to algorithms that are responsible for scheduling tasks on one processor. Reversely Global refers to scheduling policies for task allocation on multi-processors in order to optimize the final system performance. The Grid is on the Global part of the hierarchy.

On the next level of the Global abstraction we find 2 categories of scheduling algorithms, **Static** and **Dynamic** respectively. The difference between these 2 categories is the time which the scheduling decisions are made. Static scheduling assumes that there is available priory information about the Grid resources, which is being used to take decisions about the final scheduling. On the other hand dynamic scheduling is using information that gathers during the running time to make the correct processor task allocation. Both methods are used regularly in Grid Computing. This thesis examines and modifies the *xDCP* [8] algorithm which belongs to the static category of the Grid scheduling algorithms. The *xDCP* algorithm has been designed to schedule parameter-sweep application workflow. There will be an effort to insert communication cost between the application's different level subtasks, something that the *xDCP* do not consider assuming that is zero.

1.4 Simulation Framework

Great emphasis is nowadays giving to the creation and evaluation of scheduling algorithms and a lot of researchers are working on this specialized field of Grid computing. Due to the dynamic Grid characteristic the evaluation of such algorithms need to cover a wide range of different scenarios. One could say that the best way to perform such an evaluation is to do experiments with different scheduling strategies using real resources and try to schedule real applications on the Grid. This is not so easy to achieve for the reason that, first it would be difficult to obtain a considerable number of experiments that need to have a final statistical meaning and second using real resources will limit the change of using a big range of dissimilar resource configurations. For that reason *Simulation* is the most effective way to obtain results with statistical meaning that could help the efficient evaluation of this category of algorithms.

The **Simgrid** [6] simulation framework is a toolkit designed specially for the study of scheduling algorithms. It is implemented in C and consists of about 10.000 lines of source code and uses simple optimization techniques to improve memory usage and speed. The **Simgrid** library offers functions that support the arrangement of the computing environment, the implementation of the algorithm itself and finally the simulation of the application and execution over a set of defined resources. **Simgrid** is an open-source software. The newer version is v3.0 but for the purposes of this thesis v2.18.5 will be used as it is the last one that considers Direct Acyclic Graphs (DAG's) abstractions. The reader can refer to chapter (4) for a more detail description.

1.5 Structure of thesis

This thesis is organized as follows. The following chapter(2), will discuss algorithms from the static category. The next chapter(3) will present the xDCP algorithm. This chapter also proposes a modification in the xDCP algorithm in order to be able to accept communication time cost within its structure. The purpose of this modification is to investigate the impact of adding communication time between the layers of a Parameter Sweep Task application workflow. Chapter(4) presents the toolkit (**Simgrid**) that is used for the simulation and implementation of the modified algorithm. The last chapter (5) examines if the new modified algorithm is still suitable or not for scheduling when this communication cost is considered. By applying simulation we measure the algorithm performance under a lot of configuration scenarios.

Chapter 2

Static Scheduling

2.1 Scheduling Open Problems

The most important problem in scheduling task graphs (**DAG's** see section 2.1.1) for distributed computing is to efficiently find a good enough schedule with respect to the application total execution time (makespan). The Directed Acyclic Graph can state information about file and task dependencies of an application. The work of a scheduling algorithm is to gather this information and combining it with information that gathers about the system (e.g. from Grid Information Service) to take decisions about how to sent working units to resources that are ready to accommodate them in such an order to minimize the application's makespan. It has been proved that such a problem is NP-complete [7]. It is possible for someone to create a Linear Programming model with the aim of solving this task allocation to resources problem, but the existing current methods do not allow this to happen within a rational time space and thus researchers proposed a plethora of different heuristics.

All these heuristics use simplifying assumptions in order to achieve a respectable performance that will lead to a better makespan. The optimal performance is not easy to achieve and if we consider the dynamic Grid characteristic, the design of scheduling algorithms for the Grid is rather a great challenge. In order to map efficiently tasks to resources we have to keep in mind beforehand how task relations are able to influence scheduling decisions. Besides that we have to examine how the heterogeneous nature of the resources can react to the performance of the schedule. Another very important aspect is how we will be able to measure the performance of the proposed algorithm under a realistic performance model. All these open scheduling problems have been the study field between many researchers world wide and a lot of algorithms have been proposed with an effort to overcome this problem.

Designing a scheduling algorithm in order to achieve an efficient processing of a parallel application includes four fundamental aspects [11]:

- **Scalability**

- **Performance**
- **Time-Complexity**
- **Applicability**

Scalability

A parallel scheduling algorithm is scalable when it is able to create proper solutions for several problem sizes. As the number of processors that is capable to use becomes higher the algorithm should be able to produce solutions with the same value in a lower time period.

Performance

The performance of a scheduling algorithm states the quality of solutions that produces. The algorithm should have the ability to accommodate a wide range of input graphs that may describe different applications.

T-Complexity

The time-complexity of an algorithm is a very important factor that can influence its design. Although the quality of the solution that might be able to produce is good, a high time complexity can easy lead to a low scalability and thus the efficiency of the algorithm may drop dramatically. There is a need to find an acceptable trade-off between the quality of solutions that the algorithm produces and time complexity. The time-complexity of a DAG is usually expressed as a factor of number of edges, number of nodes and the number of processors that uses. In most of cases an algorithm includes a traversal of the DAG and a deep search for the appropriate place to put a task within the processors slots. Backtracking is also possible but it results in a higher time complexity. Static priority in general terms has as consequence lower T-complexity than a dynamic priority assignment.

Applicability

When we speak about applicability of a scheduling algorithm we speak in terms of its practical usage. To achieve this, realistic assumptions have to be considered and real benchmarks have to be used for evaluation purposes. If an algorithm does not include realistic assumption, it may advance the theoretical field of science but it will never have any practical usage.

Most of the proposed scheduling algorithms belong to the static part. This chapter will focus on this part presenting a collection of this kind of algorithms. First the DAG model is discussed as it fits the needs of our work and is appropriate to describe a

parameter sweep application workflow. Afterwards some of the most well performed known algorithms that belong to this category are briefly discussed. Moreover the main task is the analysis and modification of the xDCP [8] algorithm that is designed specially for scheduling Parameter Sweep Tasks on Global Grids and thus match with the case of our study.

2.1.1 The DAG Model

A graph is the basic part of study in the field of graph theory. It is a set of points or vertices connecting by links that are called lines or edges. Furthermore a **directed graph** or a digraph **G = (V, E)** is a graph that is made from a set of vertices or nodes **v** that are connected with directed edges **e**. A **directed acyclic graph** or else **DAG** is a directed graph with no directed cycles [9]. Saying no directed cycles we mean that there is no directed path in the task graph that can start and ends to a particular node n. It is called acyclic because there is no cycle in any path of the graph structure. Every node has a weight that symbolizes the computation cost and is denoted by $\mathbf{W(n_i)}$. The edges of a DAG represent the communication messages that have to be passed from a node to its children nodes to start their execution. The weight of an edge is called communication cost and is denoted by $\mathbf{C(n_i, n_j)}$. In the case of the task graph shown in Figure 2.1 between the A and B nodes there is an edge with communication cost **C(A, B)**.

Every edge has a source and a sink node that is connect to. The source is called the parent node and the sink the child node respectively. A parent node can have a number of children nodes and in the same way a children node may have more than one parent nodes. Both cases are easy to observe in Figure 2.1 were the set of tasks **{B, C, D}** are children of task A and the set of tasks **{B, C}** are father of task E in that order. In the same figure we can observe as well that task A has no father and task G has no children. These tasks are called entry and exit nodes of the graph respectively. The structure of a DAG do not allow a node to start executing before it collects all the communication messages that its parent nodes send. The **Path** of the graph is all the way from an entry node to an exit one. In the same figure we are able to detect a various number of paths. For example (A→B→E→G) is one of these paths. As the reader can easily observe there is one path with red arrows and that is the (A→C→E→G) path.

We represent on purpose this particular path with red arrows because it represents the so called Critical path of the graph. The Critical path of a DAG is the path with the longest computation and communication cost among all the task graph paths. It is characterized as Critical because it follows a restriction that is if the nodes composing it are not scheduled on time, the total application execution length will be higher. The nodes (A, C, E, and G) are additionally called **Critical Path** nodes CPN's and together create a linear cluster within the graph structure. A DAG can have more than one CP's. It can happen when we schedule a CP node, after this a new CP may appear in the graph arrangement.

According to [10] a DAG can be described by different models, some of which are following. The first is the 'accurate model'. This model considers that the weight of a node consists of the computation time and the time to receive and send the messages before and after the computation. Being a function of source and destination nodes, it depends both on node distribution and on the network arrangement. The same model assumes that communication messages between nodes within the same processor are zero. The second model is the 'first approximation model'. It assumes that the edge weight is independent of the message passing and is approximated by a constant number. The third one is the 'second approximation' model which fully ignores message passing. These approximation models are useful when the granularity ranges from medium to large and thus the communication is low and the network is not heavily loaded. The granularity is the ratio of the task execution time versus the communication overhead resulting from the messages passing among the computational tasks.

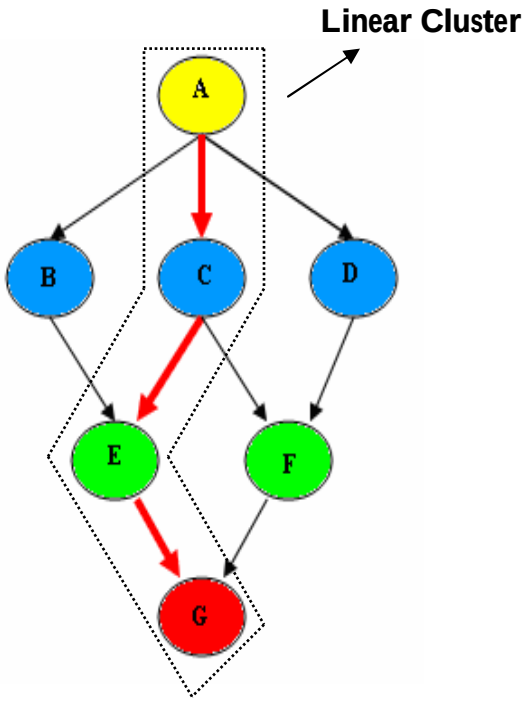


Figure 2.1: Representation of a DAG graph structure.

There are 2 ways to schedule a DAG to a processor network. In the "direct mapping" way, using the accurate model we described above, a DAG is mapped into a given network processor topology. The other way around is the indirect mapping. In this model, a DAG is scheduled without considering the processor network topology using one of the two approximate models. The most common Grid workflow can be easily modeled as simple DAG where the tasks represented by nodes have to keep a predefined order of execution. This is determined by the dependencies being modeled as directed arcs between the nodes. These data dependencies are files of various sizes sending from one application component to another application component to start its execution.

2.2 Compile Time Scheduling

Problem Statement: Given a set of n resources R_i $\{i = 1...n\}$ and a set of m tasks T_i with $\{i = 1...m\}$ try to map all *m* tasks to all *n* resources with the target to *minimize the finish time* of the last executing task.

In static scheduling the scheduler calculates the total execution schedule **S** (makespan) of an application before the run time at the compile time - this kind of scheduling is also called "compile time scheduling". In this type of scheduling, resource information like CPU's, link's speed and performance parameters are assumed to be known in advance. This model is very popular because it is easier to program from the point of view of a scheduler. A plethora of heuristics are proposed mainly based on the before known information about the application and its resource requirements with a scope of solving this NP-complete problem. Decisions can take place about if it is worth to keep tasks on the same cluster or if it is better to send them to a different one to achieve a lower computation cost. Doing this we need to consider the communication overhead that this action will have as a consequence. The solutions all these heuristics give are sub-optimal and mostly suitable just for a specific platform configuration (e.g. homogenous clusters) or a particular application (e.g. independent task graphs.). All these algorithms can not be adapted to situations where a resource fails to complete the task assigned to it because of a hardware or network problem. Some efforts have been done to address this problem by introducing rescheduling mechanisms [12] that help to bridge the gap between static and dynamic scheduling. The results published in [13] show that any approximate algorithm which can guarantee a standard performance still does not exist for the static part of the Grid scheduling algorithms. Figure 2.2 represents, using a tree structure the hierarchy of the static scheduling algorithms.

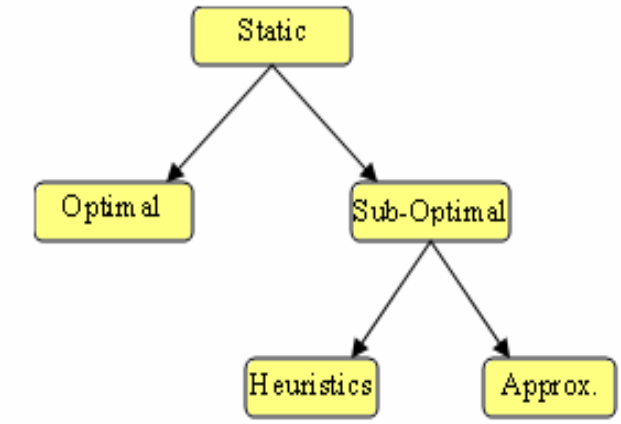


Figure 2.2: Static scheduling algorithm hierarchy

A very important factor that may influence the design of an algorithm is the task dependency of the application. Speaking about task dependency we refer to the precedence relations between tasks in a workflow graph. Restrictions exist about the right order in which the tasks have to be executed and might appear within a workflow. From

now on we will use $A \rightarrow B$ to indicate that task B must succeed task A. If there are no such a type of relations between tasks then we can say that the tasks are independent each other. At this level of abstraction we can distinguish two new categories. Independent and Dependent task graph scheduling algorithms. In the next sections we present some of the most well performed known algorithms of both categories.

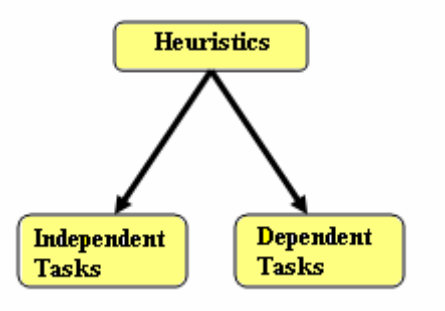


Figure 2.3: Heuristics hierarchy

2.3 Scheduling Independent Tasks

Using the terminology defined in [14], a set of independent tasks can be characterized as a Metatask. For example a Metatask is a collection of individual tasks that are submitted by different users to a supercomputer center. With the purpose of scheduling a Metatask a number of heuristics have been proposed in the literature. The common characteristic of all these heuristics is the execution time of each task. Every computing machine is assumed to execute one task each time. Multitasking operations are not allowed meaning is not possible for tasks to execute in parallel on the same resource. We will refer to the most used independent task scheduling algorithms based on related surveys and comparisons starting from the most naïve and finishing by the most efficient [13] [14] [15] [16] [17] [18].

- **Opportunistic Load Balancing (OLB)**

Among all mapping strategies the simplest is to map each task to the next available host (the first idle processor) without considering the execution time of the task in this particular host. This strategy tries to minimize the idle time of the processors by assign jobs as soon as a processor is free of tasks and ready. The time complexity is low O(v) but the method is not efficient at all, as it is very easy to assign heavy tasks to slow nodes. It is a resource centric method because it does not make effort to minimize the makespan of the application but tries to keep the processors busy.

- **Minimum Execution Time (MET)**

Every task is assigned to the first available processor that gives the lower execution time. MET does not consider CPU's ready times but assigns tasks to processors according the impact that it has in reducing the execution time. This may create imbalance situations in the load across the processors if we rely on the assumption that multitasking operations are not allowed.

- **Minimum Completion Time (MCT)**

What this method does is straightforward. The main idea is to assign every task to the processor that is able to give the earliest competition time. In this approach, tasks may be assigned to processors that are not able to give the minimum execution time (MET). Thus the efficiency of the algorithm is low but it is easy to implement and with low T-complexity.

- **Switching Algorithm**

The motivation behind this heuristic is the efficient combination of MCT and MET. Using both in a cyclic way there is an effort to defeat the load imbalance that MET creates, when assign a bigger number of tasks to some processors, by the ability of MCT to balance the load. Dividing the min-ready time over all the machines by the max-ready time they introduce a new parameter (π) that is being valued within a uniform distribution. Afterwards they set a low and a high tolerance π value and starting from 0 they assign tasks using the MCT heuristic until π reaches its high tolerance value. When this occurs they switch to MET until the load balance reaches the low tolerated value. This procedure continues in a cyclic way until all tasks are scheduled.

- **Min-min**

The Min-min heuristic was first introduced in [19]. It uses the **minimum** MCT as its basic metric to assign priorities to tasks. The inspiration behind this heuristic is the possibility to reduce the application makespan if we assign tasks to processors that are able to give the minimum completion time. The procedure starts by mapping tasks that could change the processors ready times as least as possible. For example, given two tasks t_i and t_j, which destination is to allocate to a processor m_k, the algorithm assigns first the task that will allow the processor to have a faster ready time. This strategy can raise the probability for the other task to still have the earliest completion time into this processor. They expect that assigning more tasks to processors that finish them earlier but also complete them faster can lower the application makespan. The Min-Min algorithm follows:

Let $R_{m,j}$ symbolize the **m** processor inside the **j** cluster. $C(T_i, R_{m,j})$ denote the estimated completion time of a task T_i within an $R_{m,j}$ processor.

Min-min

```
while (T ≠ ∅)
    foreach (T_i ∈ T)
        foreach Cluster J
            foreach Processor m
                Compute MCT of a task T_i
            endfor
        endfor
    endfor
  s = min_i (MCT)
  Map T_s to R_{m,j} that gives min C(T_s, R)
  T = T − {T_s}
endwhile
```

- **Max-min**

Working the same way like the Min-min heuristic, the Max-min uses the **maximum** MCT as it basic metric. The difference between Min-min and Max-min is that when a processor which gives the earliest execution time is idle, the task that has the maximum MCT is then assigned to that processor. Max-min performs better from Min-min in some cases. Imagine that we have to schedule a metatask that consists of a high number of small tasks and one large task. Max-min tries to synchronize the execution of the short tasks with the long task. On the other hand Min-min will map the short tasks first and will leave the long task for the end. This influences negatively the application's makespan compared with Max-min. Both algorithms have the same time complexity something very rational as the only different in the implementation of the two algorithms is the metric that is used, the minimum and maximum MCT respectively.

- **Sufferage – XSufferage**

Another well known heuristic that also uses the MCT metric is the Sufferage. The logic behind this heuristic is that a task must be assigned to the processor that will make it to suffer less. They define the Sufferage [18] value as the difference between the best

and second best MCT. When scheduling, the algorithm gives higher priority to the tasks with the higher Sufferage value. These kinds of algorithms are likely to have problems when resources are clustered. Within a cluster composed of processors with identical performance the best and second best MCT value is almost the same and so the Sufferage will stay near zero having as consequence to give low priority to tasks that were supposed to take high priority. To tackle this problem they propose a modification to the Sufferage heuristic, which they name XSufferage because it is an extension to the Sufferage heuristic. What they actually do is that instead of calculating the two MCT's for all the processors they compute MCT only for processors within the same cluster (cluster-level MCT). In this way tasks with higher cluster-level value obtain higher priority solving a big percentage of the problem that might be created when initially applying Sufferage.

Sufferage

while $(T \neq \varnothing)$
  foreach $(T_i \in T)$
    foreach Cluster J
      foreach Processor m
        Compute first and second best MCT of a task T_i
        $Sufferage = MCT_{[2]} - MCT_{[1]}$
      endfor
    endfor
  endfor
  $s = \max(Sufferage_i)$
  Map T_s to $R_{m,j}$ that gives the best MCT
  $T = T - \{T_s\}$
endwhile

The above heuristics that uses MCT as their critical metric (Min-min, Max-min and "X"Sufferage) are based on a particular model. For every task a pair of best processor-task **(R,T)** is selected and then T is mapped to R as soon as possible. The difference between them is the definition of the best pair and the way it is selected. For example the Min-min heuristic accepts as best task the one that makes the **(R,T)** relation minimum. A benchmarking performed in [20] between the previous mentioned heuristics shows that Sufferage and additionally XSufferage perform better in the most cases. The XSufferage heuristic is used by the AppLeS [21] workflow engine. This engine is a Grid Middleware designed specially for scheduling parameter sweep tasks.

2.4 Scheduling Dependent Tasks

There are cases where tasks composing a task graph must keep a precedence order within a workflow. One appropriate model that suits and can describe that case is the DAG model (see Section 2.1.1). The edges between the nodes represent the communication that occurs between the connected tasks. Taking into consideration this message passing it is easy to realize the reaction in the application makespan. It will create a delay. This is natural as this action needs time depending on the size of the file that passes between the two adjacent processors (nodes). This extra action introduces a new challenge when scheduling decisions are made. The parallelism that an algorithm will try to achieve must not increase the communication cost in such a way that will influence negatively the final application execution length.

For example if we consider a task that when executed produces a large file that must be input to the task that depends on it in order to start its execution. If the two tasks are not scheduled to the same resource array (e.g. Cluster) the file transfer that will take place on the link connecting the two adjacent nodes will create a significant delay. This will have a dramatic effect on the application's makespan. In this section we will discuss scheduling algorithms that consider the task dependency and try to tackle the DAG scheduling problem. These algorithms use different techniques to achieve the same goal. Depending on the method they use, they are categorized as clustering, replication and list scheduling algorithms respectively (Figure 2.4).

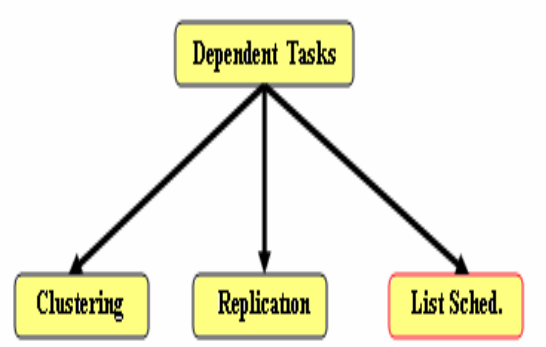


Figure 2.4: Scheduling Methods

2.4.1 Clustering methods

The main idea behind clustering methods is to group tasks with heavy communication to the same cluster and assign each task of this group to the same processor within this cluster. This process of grouping different tasks is called clustering. If we consider the communication between tasks within the same processor is zero they the task grouping and afterwards mapping on the same processor can lead to a lower makespan. Usually clustering algorithms are working by following a two step procedure. The first step is to modify the given task graph creating different clusters of tasks. The

second step is called post-clustering and is responsible for mapping the clustered tasks to the appropriate resources.

There are two different types of clusters that can be created from a given task graph, linear and nonlinear clusters respectively. Linear clustering creates groups of tasks that are sequential in order. A cluster of these tasks can for example be a simple directed path in a graph structure. Nonlinear clustering groups sequential parallel tasks. This action can improve the application makespan if the amount of communication needed is low. There is a tradeoff between the linear clustering parallelization and the nonlinear clustering sequentialization. The following figures illustrate the two different types of clustering methods for a simple DAG with computation and communication costs. In Figure 2.5 we can observe a linear clustering with three clusters and the tasks that follow a directed path within each cluster. In Figure 2.6 a nonlinear clustering is represented where dependent tasks are grouped together.

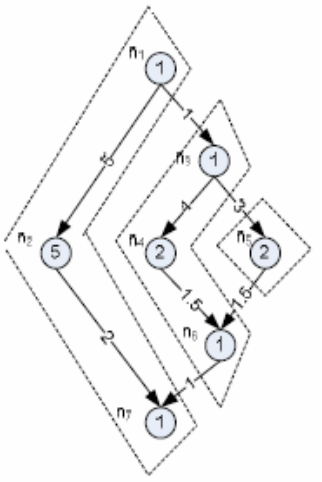


Figure 2.5: Linear Clustering

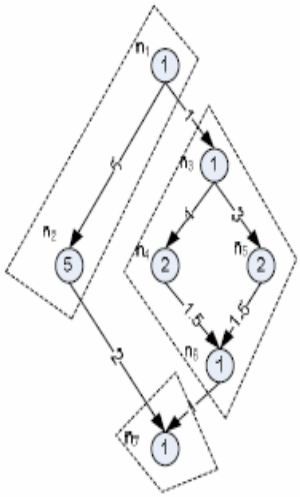


Figure 2.6: Nonlinear Clustering

- **Dominant Sequence Clustering (DSC)**

The Dominant Sequence Clustering (DSC) heuristic was proposed in [22]. As we described in section 2.1.1 the critical path (CP) is the longest path within a DAG and provides an upper bound that influences the schedule length. In order to track the CP during the scheduling procedure a new metric the Dominant Sequence (DS) is proposed. The **DS** is simple the CP of the **scheduled** DAG. The motivation behind the DSC algorithm is to perform a series of edge zeroing with goal to reduce the DS.

Initially every task is scheduled to its suitable cluster. Then the algorithm tries to merge these clusters. This action can take place by zeroing the edges of the DAG that connect them. The bottom-level is computed for each node and the top-level for each free node (node with no predecessors or with already mapped ones). The top and bottom level

are the sum of the computation and communication cost along the longest path of a DAG from a node to an entry and an exit node respectively. The first is computed during the scheduling process and the second at the beginning of the scheduling. The algorithm starts assign tasks according to their priority. This priority is defined as follows:

For a given node n $priority(n) = t-level(n) + b-level(n)$

The algorithm at each step keeps tracks of the partially scheduled DAG's CP using this priority attribute. Nodes with highest priority are not selected unless they are ready. The algorithm considers two cases. In the first case the highest priority CP node is a ready node. The second case occurs when the highest priority is not a ready node.

A ready task is a task that can start its execution because there are no pending data dependencies that prevent it. On the other hand a task is considered as not ready if is still waiting for data to start executing. In the first case (1) the algorithm searches for the cluster that allows the minimum start time and then schedules there the node. This is possible by scheduling predecessor nodes to the same processor in order to minimize the communication edge.

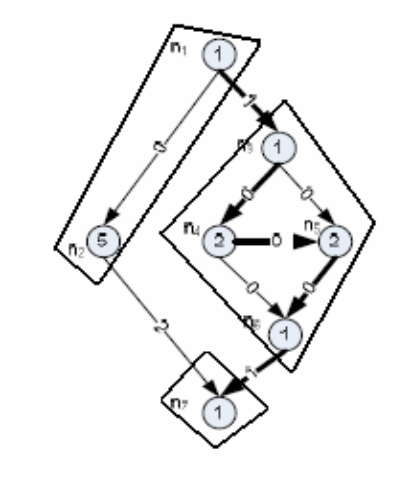


Figure 2.7: DSC applied

When the highest priority CP node is not a ready node (2) the algorithm finds the node that lies on the CP path and has the highest priority. Then the node is scheduled to a processor that minimizes its starting time without delaying the starting time of a CP node that is not scheduled yet. In Figure 2.7 the reader can observe a schedule that is created when the DSC algorithm is applied to a simple DAG. The clustering method creates three different sets of nodes. These are {n1, n2}, {n3, n4, n5, n6} and {n7}. Every set will schedule to one separate processor. The bold arrows show the DS of the scheduled graph.

2.4.2 Replication methods

This method is based on the idea of creating replicas of a task in different resources. For that reason the idle time of a processor is used to create replicas of predecessor tasks when a task is executed. During this replication mechanism the first that finishes being either the task itself or the replica is considered valid and all the others are cancelled. Changes that were made in the global state by other replicas are not considered as well. Only the finished replica can influence the global state. Replication does not need information about the execution time of a task in a resource because many replicas of the same task are distributed over multiple resources. It only considers which replica finishes first its execution. Using replication we can prevent task failures by executing, a task that fails in a resource, in a different resource using one of its replicas. This will have a negative influence to the makespan of the application but it can prevent a total failure situation.

- **Task Duplication Scheduling Algorithm (TDS)**

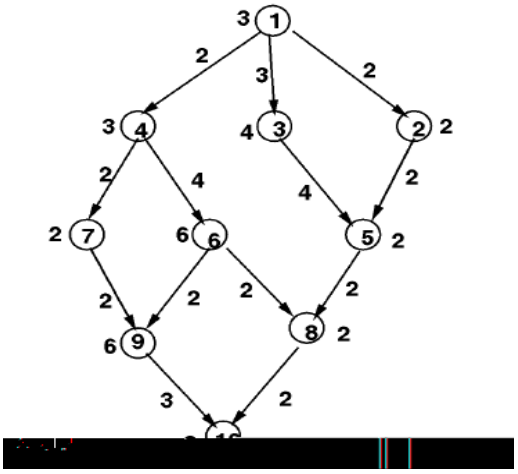
The TDS heuristic that is presented in [23] schedules tasks based on certain metrics. These are:

- Earliest Start Time **(est)**
- Earliest Completion Time **(ect)**
- Latest Allowable Start Time **(last)**
- Latest Allowable Completion Time **(lact)**
- Favorite Predecessor **(frpred)**
- Level **(level)**

They assume that every DAG has an entry and an exit node. If this is not the case they reckon a rational solution, the use of a dummy node with zero computation and communication cost that can be adjusted to the graph one at the top and the other at the bottom. The (est) and (ect) of a node are computed traversing the graph top-down starting from the entry node and finishing at an exit node of the graph. The (last) and (lact) are computed in an opposite way, traversing the graph bottom-up starting from the exit node and finishing at the entry node. The (level) of a node is the longest path from the node to the exit node. Moreover the (level) of the entry node is an upper bound to the final schedule length, something similar to the CP. The (fpred) of a node n is assigned using the following equation:

$$fpred(n) = j \,\Big|\, \Big(ect(j) + C(j,n)\Big) \geq \Big(ect(k)\Big) + C(k,n)$$
$$\forall j,k \in P(n) \wedge k \neq j$$

The equation assigns as favorite predecessor the one that finishes earlier than all the others. Having some of the metrics (est, ect, fpred, level) computed, the algorithm creates a priority queue based on the level value. Then starting from the lower level value node makes clusters from this node to the entry node following its favorite predecessors. Each separate cluster is assigned to one particular processor. In this procedure the (last) and (lact) metrics are used to decide where there is a need for a node replication. If the communication cost between a node and its favorite unsigned predecessor is higher than the difference between (lact) and (last), then the predecessor will be assigned to the same cluster with the current node. If it is in a different cluster a replica will be created. Figure 2.8 illustrates a DAG with valued computation and communication weights and a matrix with the calculation of the above mentioned metrics.



START AND COMPLETION TIMES FOR THE NODES

Node	*level*	*est*	*ect*	*last*	*lact*	fpred
1	21	0	3	0	3	–
2	9	3	5	6	8	1
3	11	3	7	6	10	1
4	18	3	6	3	6	1
5	7	7	9	10	12	3
6	15	6	12	6	12	4
7	11	6	8	8	10	4
8	5	12	14	14	16	6
9	9	12	18	12	18	6
10	3	18	21	18	21	9

Figure 2.8: Calculation of scheduling metrics for TDS

2.4.3 List Scheduling Methods

A heuristic is able to produce a solution in less time but there is no guarantee that the solution that produces is optimal. Among all the related literature approaches the most popular and most efficient is the **list scheduling** technique. Using list scheduling priorities are assigned to tasks and after scheduling decisions are taking according to the priority list. In the same way that DAG identifies the precedent relations between tasks, a priority list states the order in which tasks have to assign to resources. In the static list scheduling the priority list of nodes is statically created before the task allocation. The sequence of priorities in the list can not change during the scheduling operation. The priority list technique is mainly based on a three step procedure:

1. When all processors are busy wait.
2. If there is a free processor then assign the first **ready** task of the priority list to this processor.
3. Repeat until all tasks are scheduled.

Scheduling algorithms that employ this three-step approach can potentially generate better schedules [24]. Only ready tasks can be assigned to resources. A ready task is the one which execution respects the graph precedence relations. Precedence relations can override the priority list. If for example $A \rightarrow B$ and the given priority list is constructed in such an order that B has higher priority than A, task B cannot be assigned to a processor until task A is completed. This can easily lead to a deadlock situation of the application during the schedule and needs to be considered when creating a priority list.

For an application composed from N tasks there is fixed number of possible priority lists allowed. This number is $N!$. For an application that consists from 10 tasks it will be 10! = 36288800 different cases. It is easy for the reader to realize that for more tasks the number of possible lists grows exponentially. Since it is not practical to analyze all the list possibilities and some times it is impossible within a reasonable time, (e.g. an application with 1000 tasks) a plethora of heurists are proposed in the related literature [8][25][26][27][28][29] [30].

All these various scheduling algorithms differ in the methods used to assign priorities in order to create the ready list. They also differ in the way they select processors to accommodate tasks. When a task is selected from the priority list, it is then send to a processor to execute. In general, in order to create an efficient schedule the scheduler must allocate the task to the resource that allows the minimum completion time and minimize the data file transfer time. The most frequently used priority attributes are the b-level and t-level (see Section 2.4.1). The t-level can give information about the earliest start time of a task after it is assigned to a processor. This attribute has a dynamic characteristic. This is clear to realize is if we consider the case of two communicating tasks that are mapped to the same processor. Then the connecting edge and so the communication become zero and does not count any more on the calculation of the attribute.

- **Modified Critical Path (MCP)**

The MCP algorithm was introduced in [25] [30]. It uses the Absolute Latest Starting Time (ALST) as it priority metric. The ALST is computed, subtracting the b-level from the CP weight. Considering this the ALST of a node that lies on the CP is just its t-level. Afterwards a priority list is constructed in an ascending order of ALST times. Ties are broken based on the priorities of the descendant nodes. In this step ties are broken by all the descendants of the node. Experiments in [26] show that is not necessary to use all the descendants to break ties. Instead if we distinguish one level of descendants we can produce the same results. This can improve the time complexity of the algorithm. During the scheduling procedure there may appear empty spaces within the processors, between the nodes due to dependencies. MCP schedules a ready node in the first available empty space. This approach is called **insertion**. If a node is scheduled after its last predecessor node without consider the empty spaces then the approach is called **non-insertion**.

Revised MCP

1. Compute $ALST \forall (t \in T_i) \in T$
2. Create priority list of nodes. Ties are broken by the successor that minimizes the $ALST$ time.
3. Assign the highest priority node to the processor that allows the earliest execution time based on the **insertion** approach.
4 Repeat step (3) until the node list is empty.

- **Fast Critical Path (FCP)**

The motivation behind FCP [27] is based on an observation about the time complexity of each individual step that is involved in most of list scheduling methods. The first step is the computation of the nodes priorities, needed to traverse (visiting all the graph nodes and edges) the DAG at least once. Consequently the time complexity for this computation is at least **O(E+V)**. The second step is the specification of the priority list by sorting the tasks according to their predefined priorities. It takes **O(VlogV)** time. The third and last step assigns tasks to their "favorite" processor. In order to take the decision about the "favorite" processor usually the AEST metric is used. Considering that we have a set of **P** resources to map the tasks, the time complexity for the calculation of the AEST will be $O((E+V)P)$.

Among these three steps the third one is the most time consuming in many cases. To reduce this complexity the FCP algorithm uses two queues. One shorted priority queue but with a constant number of tasks and one unsorted FIFO queue with just O(1) time access. The first queue accommodates only the ready tasks and all the others are put in the FIFO queue. When a task is ready and there is a free slot in the sorted queue it moves there or else moves in the FIFO queue. Keeping a good analogy between the sizes of the two queues could prevent the assignment of a ready task in the FIFO queue. The idea of shorting every time only ready tasks can lower the time complexity.

Using a shorted queue with size K **<** V the complexity drops to $O(V\log(K))$. They also prove that the complexity can drop if there is a restriction between the possible destinations (assignment to a processor) of a given task. The observation that they did was that there are just two processors that can minimize the starting time of a node. These are the task's enabling processor and the first idle processor. As a consequence the complexity drops to just $O(E+V\log(K))$.

FCP

```
foreach Task
    Compute priority value
        foreach ready Task
            AddReadyTask()
        endfor
endfor
while unscheduled Task
    SelectReadyTask()
    SelectProcessor()
    Schedule()
        foreach new ready Task
            AddReadyTask()
        endfor
endwhile
```

The algorithm uses three basic functions (AddReadyTask(), SelectReadyTask(), SelectProcessor()) to schedule. Firstly the AddReadyTask function is used to stage the tasks into the two queues with a static priority assignment. Secondly the SelectReadyTask function selects the highest priority task from the priority queue having in mind to keep this list full after the selection. This can be done by moving a task from the FIFO queue (if there is one) to the priority queue. Finally the SelectProcessor function finds the best processor to accommodate the selected task using the lower complexity strategy they proposed. A comparison in [28] between FCP and other static scheduling heuristics shows that FCP outperforms in most of the cases even higher efficiency and complexity heuristics (e.g. MCP).

- **Dynamic Critical Path (DCP)**

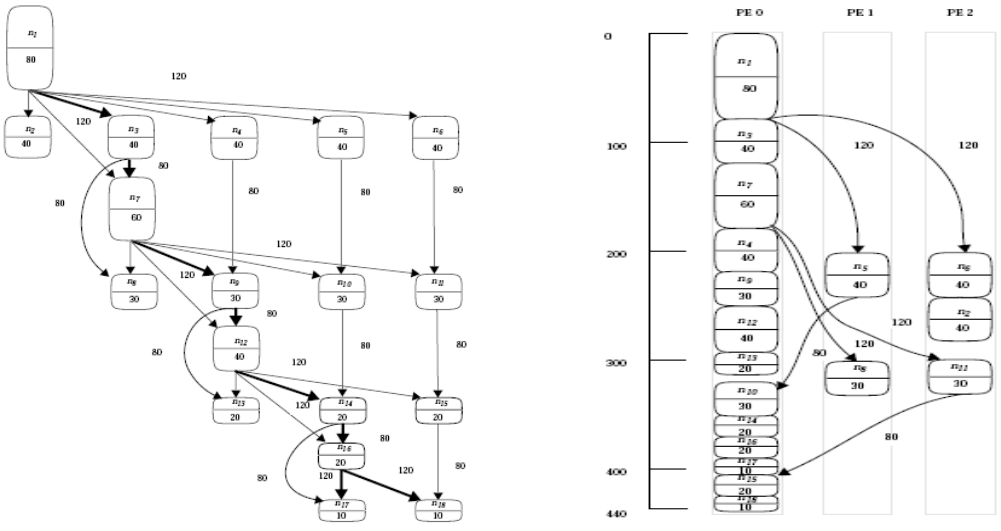
The DCP heuristic [29] is based on the observation of a mobility attribute. This attribute is the difference between AEFT and AEST. This can be defined as:

$$DS - \left(b - level + t - level \right)$$

They consider that the CP of a graph can change after a CP node is scheduled. For that reason they introduce a new attribute denoted by DCPL (Dynamic Critical Path Length) . This is simply the CP of the partially scheduled graph. The nodes on the CP are

the ones where the difference between the ALST and AEST is zero. The algorithm uses a "looking ahead" strategy to select processors for tasks. The aim of this strategy is to prevent scheduling a task to a processor that has no space for a heavy communicating successor of the task. If this occurs, the possibility of zeroing the communication edge will be lost. The reaction on the application makespan will be obvious. They define a restriction on the examination of the processors in order to minimize the starting time of a node. This set can be reduced to the processors that accommodate its parent nodes and those processors that hold the nodes earliest scheduled children. The algorithm schedules the CP nodes first and updates its AEFT and AEST after each scheduling step to determine the next CP node. The DCP algorithm is described by a six step procedure as follows:

1. Compute $AEST$ and $ALST$ for all nodes.
2. Select the node n_i that minimizes the $ALST - AEST$ value and find its unscheduled child n_c with the largest communication cost.
3. Select a processor P such that gives the smallest $AEST(n_i) + AEST(n_c)$ value among all the processors that hold the n_c's parents or children. Doing this first try to find an idle time slot. If not possible try to create an idle time slot by moving some already scheduled nodes downward always considering the mobility attributes. In case of failure select a new processor.
4. Schedule n_c to P.
5. Update $AEST$ and $ALST$ for all nodes.
6. Repeat 2-5 steps until all nodes are scheduled.



(a) (b)

Figure 2.9: A Gaussian Elimination graph scheduled by DCP (a) and the result Gant-Chart (b).

In Figure 2.9 (a) a type of DAG (Gaussian Elimination) is scheduled using the above six step procedure. The result schedule is represented by a Gant-Chart in Figure

2.9 (b). The thick arrows show the CP of the graph. This creates an upper bound to the final application's execution length. As we can see from the result schedule, the algorithm arranges all the CP nodes in one processor in order to minimize the communication overhead between them.

The DCP algorithm was designed specially for a Multiprocessor system (homogenous set of processors) with the same performance. They also assume the availability of an unbounded number of processors. This does not match to the dynamic Grid characteristic (heterogeneous and limited number of resources). The reason we referred to this particular algorithm is the correlation with our case study algorithm the Extend Dynamic Critical Path (**xDCP**) described in the next chapter.

Chapter 3

Extended Dynamic Critical Path (xDCP)

3.1 Parameter-Sweep Task Graph/Workflow Semantics

The xDCP algorithm is proposed in [8]. The aim of this algorithm is to schedule workflows with parameter sweep-tasks (**PST's**). These types of task graphs belong to the DAG taxonomy. Compared to other applications, the graph structure and the workflow of parameter-sweep applications are distinguished by some special features:

1. Subtasks and resources are arranged as layers (sets of tasks and resources).
2. Given a set of n resources $\bigcup_{i=1}^{n} R_i$ and a set of n tasks $\bigcup_{i=1}^{n} T_i$. A subtask $t_{ij} \in T_i$ is allowed to be scheduled **only** to R_i.
3. There is no dependency between the same layer subtasks.
4. Dependency may exist between subtasks but only between an ancestor and a descendant layer.

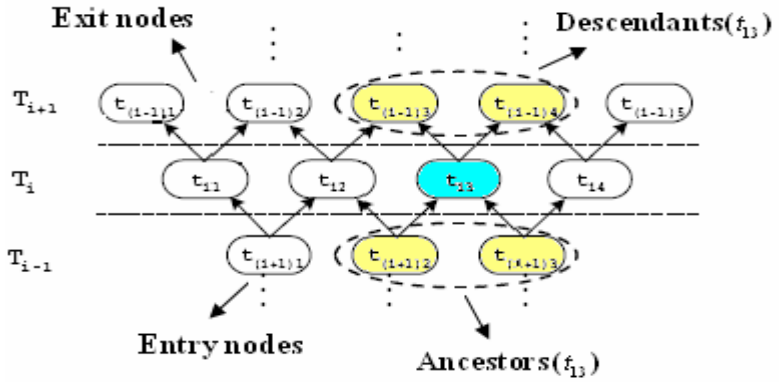


Figure 3.1: An example of a parameter sweep task graph workflow.

Figure 2.9 illustrates a small part of a workflow graph that belongs to a parameter sweep application which consists of T_i {i = 1…n} tasks. Every separate task parts from a predefined number of subtasks. It is easy to notice that there are no dependencies between the same level subtasks. Dependencies appear only between subtasks from a current layer with subtasks from a previous and a next layer respectively. For example on the given graph there are dependencies between ($T_{i-1} \rightarrow T_i \rightarrow T_{i+1}$) layer. The subtask t_{13} depends directly from the subtasks in the Ancestors circle (Figure 3.1). Correspondingly the subtasks in the Descendants circle depend from it. These dependency relations are denoted as follows:

$$Ancestor(t_{ij}) = \begin{cases} \varnothing \text{ if } t_{ij} \in \textbf{Entry} \\ \\ t'' \text{ if } t_{ij} \in T_{i-1} \wedge (t_{ij} \prec t'') \end{cases}$$

The Ancestor of a subtask t_{ij} is the predecessor subtask the current subtask directly depends on. It is not allowed any subtask to have an intermediate execution between the two dependent subtasks. The subtask t_{ij} can not start executing before its Ancestor t'' finishes, because **depends directly** from it.

$$Descendant(t_{ij}) = \begin{cases} \varnothing \text{ if } t_{ij} \in \textbf{Exit} \\ \\ t'' \text{ if } t_{ij} \in T_{i+1} \wedge (t'' \prec t_{ij}) \end{cases}$$

The Descendant is the successor subtask that directly depends on the current subtask. The subtask t'' can not start executing before the subtask t_{ij} finishes. Again the direct dependency defines this restriction. In general a subtask t_{ij} can not be executed before the time where all its ancestors finish and after the time where all its descendants start.

Having a collection of **K** subtasks that construct a PST graph we can define using the above terms the **Entry** and the **Exit** of the graph respectively:

$$Exit(K) = k \,|\, (k \in K \vee Descedant(k) = \varnothing)$$
$$Entry(K) = k \,|\, (k \in K \vee Ancestor(k) = \varnothing)$$

Although the above mentioned terms define the direct dependencies between subtasks, in a given PST task graph this is not only the case. A subtask depends not only on its Ancestor layer subtask(s), but also on other subtasks that belong to lower level layers. This new type of dependency that emerges in a PST graph is the **indirect**

dependency. Direct and indirect dependencies form a path from a current node to an entry node. This path creates a tree. Recursively this can be defined as:

$$AncestorS\left(t_{ij}\right)=\bigcup\left(AncestorS\left(Ancestor\left(t_{ij}\right)\right)\cup Ancestor\left(t_{ij}\right)\right)$$

The execution time is the time cost for the longest path from an entry node to an exit node in any given PST graph. Having a set of resources R we can map the collection of subtasks K within the resources and in this way we can compute the execution time as it is defined below.

$$f\left(t\right):K\rightarrow R$$

$$ExTime\left(K,R\right)=\begin{cases}0 \text{ if } K=\varnothing \\ \max_{t_{ij}\in Exit(K)}\left(ExTime\left(AncestorS\left(t_{ij}\right),R\right)\right)\end{cases}$$

3.2 Algorithm extensions

The DCP [29] algorithm was designed to work efficiently in a homogenous set of processors. To remove this restriction they proposed some extensions in order to be able to find use in irregular heterogeneous resources. These extensions can be separated in two parts, one focusing on the structure and the other on the performance of the algorithm. The structure extensions simplify parts of the algorithm that have no more significance if we consider the attributes described in section 3.1. The other extensions target to improve the performance based on some observations and experiments performed on workflows with PST's.

- **Structure Definitions**

1. Instead of AEST and ALST two new metrics are introduced, the AEFT(Absolute Earliest Finish Time) and ALFT(Absolute Latest Finish Time). If we consider the case of a subtask that has better AEST on a particular resource than from any other resources available, then the assignment of the task to this resource will create a better schedule. In a heterogeneous resources environment the execution time of the same task is different on every resource and thus the AEST and ALST metric has no more meaning. In Figure 2.2 we can observe the variation of the range of these metrics when we assign the same task to different resources.

2. When implementing the subtask dependencies the DCP algorithms considers the communication overhead between the subtasks when a message (file) passes from one depended subtask to the other. The xDCP algorithm claims that this overhead does not exist. They prove this considering the special characteristic of a PST workflow graph. In this type of graph there is no inner-task communication. Therefore the terms related with the communication cost from the DCP algorithms when implementing the xDCP are removed.

3. Another distinct attribute of a PST workflow is that every separate layer of the application can only be scheduled to a pre-specified set of resources (e.g. heterogeneous cluster). The DCP algorithm tries to prevent deadlock situations by looking if a descendant is assigned to execute before of an ancestor in the same resource queue. On the other hand in the xDCP this check is not needed because the subtasks that form one layer are independent. Thus subtasks with dependencies among them will never be scheduled to the same set of resources.

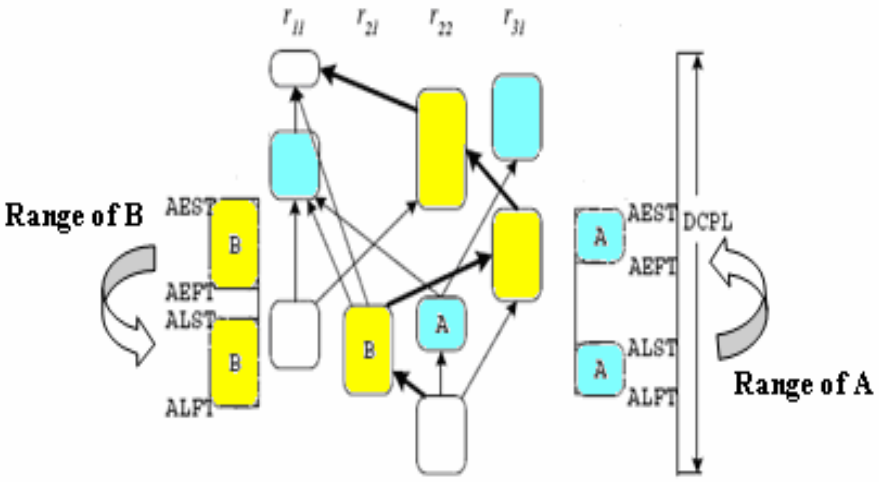


Figure 3.2: Variation of AEST/AEFT
and ALST/ALFT values

- **Performance Improvement**

1. The DCP algorithm initializes the tasks in one resource and then arranges them in sequential order in one queue. The other resources are initially empty. Reversely in xDCP the tasks are initialized in a round-robin way. This is done for every separate layer which targets the corresponding resource set (Ti scheduled to Ri). This naïve algorithm is called Shuffle. The description of the Shuffle algorithm follows:

Given a set of n resources $\bigcup_{i=1}^{n} R_i$ and a set of m tasks $\bigcup_{i=1}^{m} T_i$ with $r_{ij} \in R_i$ and $t_{ij} \in T_i$ if

N(Ri) states the number of the individual resources in a particular resource array then initialize by mapping t_{ij} to r_{ik} {k = j%N(Ri)}. Based on experiments, this can improve the effectiveness of the DCP by 30% for PST workflows.

2. The xDCP algorithms does not terminate when all tasks are scheduled. Despite that it keeps looping with respect to a predefined tolerance value. This tolerance value is set to 5%. This value is the rate of the current schedule length and the new schedule length after the rescheduling. This can give an extra 10% to 20% effectiveness to the algorithm.

3.3 Algorithm Description

The new basic metrics behind the *xDCP* algorithm are the AEFT and ALFT. In order to define these two new metrics we have first to provide the definition of the resource queue. A processor queue is assumed to be a FIFO (First In First Out). This queue is composed of an unbounded number of slots. Every slot is able to accommodate only one subtask and do not allow any multitasking operations. Subtasks enter from the tail of the queue and leave from the head of queue to start their execution. We can see an example of such a type of queue in the Figure 3.3. In the same

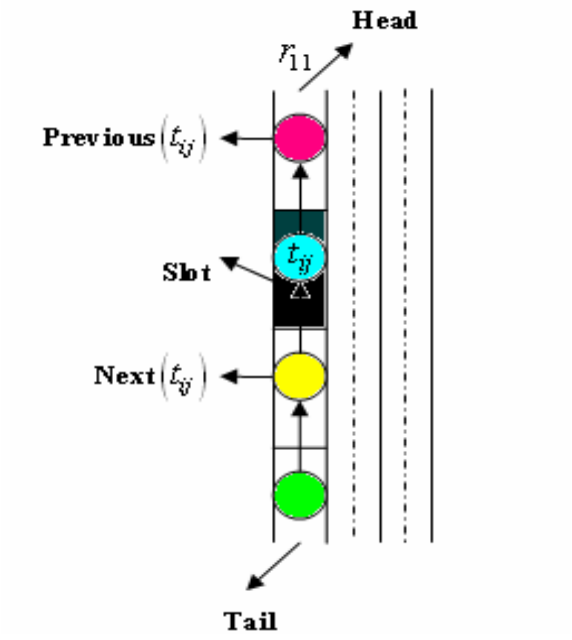


Figure 3.3: *xDCP* queue definition

figure we can observe the appearance of two new terms, the next and the previous of a current subtask in the same resource queue. These can be expressed as:

$$P(t_{ij}) = \begin{cases} \varnothing \text{ if } \forall t' \in T, t \,\&\, t' \text{ are not in the same queue} \\ t' \text{ if } t \,\&\, t' \text{ are in the same queue } \wedge (t' \prec t) \end{cases}$$

$$N(t_{ij}) = \begin{cases} \varnothing \text{ if } \forall t' \in T, t \,\&\, t' \text{ are not in the same queue} \\ t' \text{ if } t \,\&\, t' \text{ are in the same queue } \wedge (t \prec t') \end{cases}$$

Imagine N(t) → t → P(t) as a pipeline with strict ties. There can not be other subtask between them. In order a subtask to start its execution it needs to satisfy some conditions. It can not start executing before all its ancestors finish their execution and before the resource that is assigned to is idle. This means that all the previous subtasks on the same resource queue have to finish their execution before the current subtask start. The Next and Previous terms refer to the execution order and not the queue order. Previous P(t) is the subtask being executed before the execution of a current examined subtask in the same processor. Next N(t) is the subtask that will execute right after the execution of the current examined subtask in the same processor queue.

The AEFT value can be calculated by adding the computation cost, in time units (denoted by ω(t)) in the current resource, to the AEST with respect to the resource queue restrictions. Starting from the Entry and using an up-down graph traversal strategy we can calculate the AEFT value. This can recursively defined as :

$$AEFT(t) = \begin{cases} 0 \text{ if } Ancestor(t) = \varnothing \wedge P(t) = \varnothing \\ \max\left\{ \max_{\forall t' \in Ancestor(t)} \left\{ AEFT(t') + \omega(t) \right\}, AEFT\left(P(t)\right) + \omega(t) \right\} \end{cases}$$

- $\max_{\forall t' \in Ancestor(t)} \left\{ AEFT(t') + \omega(t) \right\}$

The above term computes the absolute earliest time that the last Ancestor of a subtask (t) finishes its execution plus the computation time of (t) on its current resource. This value is not the real AEFT(t) value if we consider the queue restrictions.

- $\max\left\{ \max_{\forall t' \in Ancestor(t)} \left\{ AEFT(t') + \omega(t) \right\}, AEFT\left(P(t)\right) + \omega(t) \right\}$

The AEFT(t) is the maximum between the before mentioned value and the AEFT of its previous subtask on the same resource queue. For example if we refer to the first time as time1 and for the second as time2 and if time1 > time2 then AEFT(t) = time1. Although its previous subtask P(t) on the same resource queue has already finished, t can not be executed before his last Ancestor finishes executed in some other resource queue. Having the AEFT computed, another metric the DCPL can then defined:

$$DCPL = \max_{t \in T} \left\{ AEFT(t) \right\}$$

The Dynamic Critical Path Length (DCPL) is the maximum time between all subtask's AEFT's. It is a dynamic metric and its value changes every new scheduling step. The algorithm recalculates the AEFT of all the subtasks in every scheduling step to define the new DCPL.

The ALFT value can be calculated traversing the graph in an opposite direction than the AEFT way (bottom-up) starting from the Exit and moving to the Entry. The DCPL is used as initial value for the calculation of the first scheduling step. This is defined as follows:

$$ALFT(t) = \begin{cases} DCPL \text{ if } Descendant(t) = \varnothing \wedge N(t) = \varnothing \\ \max\left\{ \max_{\forall t' \in Descendant(t)} \left\{ ALFT(t') - \omega(t') \right\}, ALFT(N(t)) - \omega(N(t)) \right\} \end{cases}$$

- $\max_{\forall t' \in Descendant(t)} \left\{ ALFT(t') - \omega(t') \right\}$

The above term computes the maximum latest starting time of a subtask's Descendant. Subtracting the computation time on the current resource from the latest finish time we can calculate the latest starting time (ALST) of the subtask. This is logical if we consider that the earliest finish time is the sum of the earliest starting time and the computation on the current resource. A subtask can not finish its execution after the latest starting time of its last starting Descendant.

- $\max\left\{ \max_{\forall t' \in Descendant(t)} \left\{ ALFT(t') - \omega(t') \right\}, ALFT(N(t)) - \omega(N(t)) \right\}$

Consequently the Absolute Latest Finish (ALFT) time of subtask should not be later than the maximum latest starting time of its Descendant (the one that starts last its execution) and the latest starting time of the next (N(t)) subtask on the same resource queue. We can combine these new metrics with the previous we discussed (AEST and ALST) in Section 2.4.3 to create some rules needed to prevent deadlock situations during the scheduling process.

Rules: If C denotes the communication cost between two directly dependent subtasks and $Ancestor(t) = t'$, $Descendant(t) = t''$ then:

I. $AEST\left(Descedant(t)\right) + C_{tt''} \geq AEFT\left(Ansestor(t)\right) + \omega(t) + C_{t't}$

II. $ALST\left(Descedant(t)\right) + C_{tt''} \geq ALFT\left(Ancestor(t)\right) + \omega(t) + C_{t't}$

Having stated all the algorithm semantics, we can now present the final form of the **xDCP** as it was originally proposed in [8]:

1) Use the Shuffle algorithm to stage the all the subtasks in the resources
2) $\forall t_{ij} \in \bigcup_{i=1}^{n} T_i$ set flag $FALSE$
3) $\forall t_{ij} \in \bigcup_{i=1}^{n} T_i$ compute $AEFT(t)$ and $ALFT(t)$
4) Select $t_{ij} \in \bigcup_{i=1}^{n} T_i$ that orderly follows the next 3 criterions:
   a) Minimize the $ALFT\left(t_{ij}\right) - AEFT\left(t_{ij}\right)$
   b) Minimize the value of i
   c) Minimize the value of $AEFT\left(t_{ij}\right)$
5) $\forall r_{ij} \in \bigcup_{i=1}^{n} R_i$ select r and the slot in $r's$ queue that assuming t is allocating into that slot orderly satisfying:
   a) $ALFT(t) - AEFT\left(P(t)\right) \geq \omega(t)$
   b) Minimize the value of $AEFT(t)$

   If there exist such a slot move t or else do not move anything.
6) Set t flag $TRUE$. If $\exists t$ $FALSE$ goto 3)
7) If $\left(DCPL_{init} / DCPL\right) * 100\% < 95\%$ goto 2) or else the algorithm terminates.

3.4 Algorithm Analysis

In the xDCP analysis we can distinguish four different functions used to create the final schedule result:

- Initialization (1, 2, 3 steps)
- Select Appropriate Task (4 step)

- Find Resource and Schedule Task (5,6 steps)
- Reschedule (7 step)

The first function is the Initialization, where the Shuffle algorithm (see section 3.2) is being used to stage the subtasks within the resources. On every subtask a flag is then attached to provide its state (if is scheduled or not) and uses the two basic Boolean operators (TRUE or FALSE). Afterwards the new proposed attributes AEFT and ALFT are being computed respectively using the two traversal strategies (up-down and bottom-up respectively). At this time the initial priority list is being constructed using the new calculated attributes.

The second function is the appropriate subtask selection. At this point three conditions have to follow in the proposed order and are used to select the highest priority subtask. The algorithm first checks which subtask minimizes the ALFT-AEFT value. This action takes place in order to determine which subtasks are on the CP path. If the difference of ALFT and AEFT for a given subtask is zero then it is on the CP. The algorithm collects all these subtasks in a set. Afterwards reduces the set to these subtasks that minimize the (**i)** layer value. This creates a restriction in the scheduling order of the PST graph layers. One layer has to follow the other going from an (i) to an (i+1) layer. Only when all the subtasks on a current layer are scheduled the algorithm starts examining the subtasks from the next layer. Finally from this set selects the subtask that minimize the AEFT value and this is the one that is considered ready and starts first its execution.

Third in order is the Find Resource and Schedule function. The selected subtask is assumed to be allocated on every available resource queue. Following the queue restrictions mentioned in Section 3.3 the ready subtask can only be allocated at the tail of each queue. Then two new conditions have to be maintained to reduce the processors set. First the algorithm checks which processors follow the next rule:

$$ALFT(t) - AEFT(P(t)) \geq \omega(t)$$

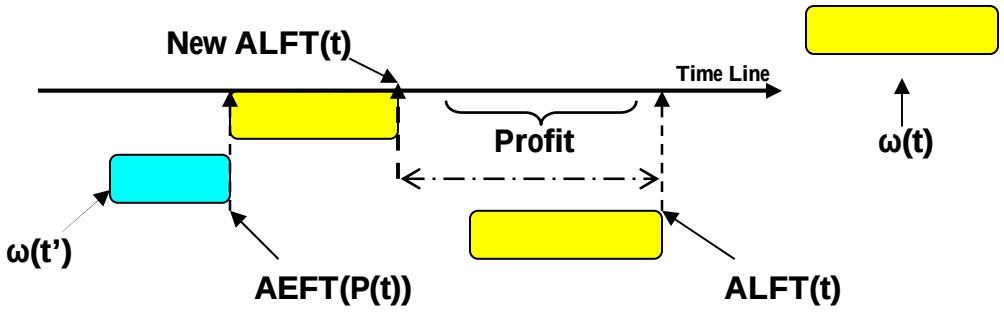


Figure 3.4: Best Processor Search

Figure 3.4 illustrates a visualization of the above rule. It represents one processor queue that follows this rule. This particular processor is the one that can accommodate the selected subtask decreasing its ALFT value to the maximum possible. The ALFT of the

current subtask varies in the same way varies its computation cost due to the heterogeneous processor environment. The selected subtask is assigned to the processor that gives the biggest profit in time units and is now considered as a scheduled one (the subtask's flag value becomes TRUE).

The last is the Reschedule function. The algorithm after having assigned all the subtasks to the resources does not terminate but loop back and the scheduling procedure starts from the beginning. Although the subtasks are initially arranged now according to the first schedule. After the rescheduling a new DCPL value emerges. Then the rate between the new DCPL and the pre-rescheduling DCPL is computed. If this rate is less than a certain threshold value the scheduling is repeated again or else the algorithm terminates. This tolerance value is set to 5% for the xDCP case.

- **Algorithm Limitations**

During the analysis procedure we faced limitations in the structure of the algorithm and in the definition of the semantics. We enumerate these limitations-drawbacks:

1. When in [8] the AEFT value is defined, they claim that AEFT(t) = 0 if (t) is an entry node and there is no previous subtask (P(t)) in the same resource queue. This is not accurate. The AEFT of this subtask can not be zero. It has to be the execution time cost (ω(t)) on the current resource. This value is needed as the initial value for the graph traversal in order to determine the next subtasks AEFT. Thus in our implementation of the algorithm we use the ω(t) value and not zero that is defined in [8].

2. In the Select Subtask function we reckon that it is needed to reverse the order of the (a), (b) step (see section 3.3). The algorithm first checks which subtask(s) minimize the ALFT-AEFT value and collect it/them into a set. After reduces the set according to which subtask(s) minimize the layer (i) value. We know that if ALFT(t)-AEFT(t) = 0 then the subtask lays on the CP. The subtasks that lay on the CP are not belonging only in the current examined layer but also in other layers. Thus if we make first the layer restriction we will have already the set reduced. This can give more speed to the algorithm in cases where the application consists of a large number of subtasks.

3. Another drawback appears in the Find Resource function. The selected for scheduling subtask is assumed to be assigned to $\forall r_{ij} \in R_i$ for {i = 1…n} and after two conditions have to be followed in the given order. In Section 3.1 we described some special characteristics of a PST workflow. One of these is the fact that a subtask t_{ij} is only allowed to be scheduled to R_i, this means it can only be scheduled to one particular resource array and not in every available ones. The (i) value is predefined before, in the Select Subtask function. The algorithm assigns subtasks to

resources that are by definition not allowed to execute them. Thus in our implementation we assume that a subtask t_{ij} is being mapped only to $\forall r_{ij} \in R_i$ for a constant (i) value. Moreover the second condition in the Find Resource function states that the selection of the appropriate processor from the set that was created before by the first condition is based on which one minimizes the AEFT(t) value. This value is later used to make the final processor choice. We reckon that instead of AEFT(t), the AEFT(P(t)) value has to be used and that is what we used in our implementation.

We can prove this based on the Figure 3.4. The processor r_{ij} that minimizes the AEFT(P(t)) will give the biggest time units profit. Instead if we use the AEFT(t) we will have, every time that we assume a subtask is put in a different resource queue, to recalculate the AEFT and ALFT. For example if we have to map a PST layer with 100 subtasks to a given resource array with 40 resources then we will have to recalculate the AEFT and ALFT values 100 x 39 = 3900! times. Of course this can not be the case.

4. The Reschedule function is executed at least once in order to obtain the new DCPL value and to create the proposed tolerance rate. This automatically raises the time complexity of the algorithm because it is initially executed twice. We find that the 10%-20% more efficiency it gives is not a sufficient number that can overcome the time complexity if we consider an application with a high number of subtasks.

3.4 Modified xDCP with Communication

The subtasks which compose one layer of a PST application are independent (there is no inter process communication) by definition. Each subtask typically evaluates a multi-dimensional objective function at a point in a multi-dimensional parameter space. The work in [21] considers that subtasks in a PST may share file dependencies but despite that they target to schedule just one layer of a PST workflow and they do not treat it as a multilayer application. Thus they use independent task heuristics (e.g. XSufferage) to schedule. But layers of PST's should communicate, not communicating horizontally among subtasks of a PST but at least vertically between layers of a PST. In the implementation of the modified algorithm we will consider this communication.

In order to insert communication cost within the xDCP algorithm structure we have to revise the two basic attributes, AEFT and ALFT. These are being used to assign priorities to subtasks with a scope to define the ready for scheduling subtask in every step. We can calculate the AEFT value recursively starting from an entry node and traversing the graph till an exit node. The AEFT of the first ready subtask is the computation cost (in time units) in its current resource, if there is no Previous (P(t)) subtask on the same resource queue. In the case of the first ready subtask it will always have no Previous. Even if the resource array that is assigned to this subtask layer consists of one processor, it will be always first in order. Having this value calculated we can now

calculate the AEFT's of the other subtasks that belong to the same layer. The first layer has no vertical dependencies thus no communication is considered. If the number of processors ($N(R_i)$) in the resource array is greater or equal to the number of subtasks then all of them AEFT's will be the computation time on their current resource respectively. If this is not the case and the $N(R_i)$ is less than the number of subtasks on layer, then the AEFT of a subtask will be the AEFT of its Previous on the same resource queue plus the computation time on this resource.

Since we computed all the subtasks AEFT of the first layer we move to the second layer. In the original xDCP algorithm they continue traversing the graph and computing AEFT's without considering any type of dependencies between the previous and the successor layer. They compute AEFT based on which of a particular subtask's Ancestor finishes last its execution. Again if there is no Previous in the same resource queue the current subtask AEFT will be the maximum AEFT between its Ancestor(t) plus the computation time in the resource. If there is a Previous (P(t)) then the AEFT of this subtask will be the AEFT of P(t) plus its computation cost on the current resource.

We reckon that there is the point where we can insert communication time cost denoted by C_{ab} where (a) is the subtask that sends the file and (b) the one that receives it. If (a) sends the data on a given time t to the other subtask (b), it can not start executing before the time step $t + C_{ab}$. The new AEFT value denoted by AEFTC follows:

$$AEFT_C(t) = \begin{cases} \omega(t) & \text{if } Ancestor(t) = \varnothing \wedge P(t) = \varnothing \\ \max\left\{ \max_{\forall t' \in Ancestor(t)} \left\{ AEFT_C(t') + \omega(t) + C_{t't} \right\}, AEFT_C\left(P(t)\right) + \omega(t) \right\} & \end{cases}$$

(1) (2)

We have inserted communication cost only in the first term and not in the second. The reason is that the subtasks t and P(t) belong to the same layer and so there is no dependency between them.

Having the AEFT values computed, we are now able to compute the DCPL value. This is the maximum value between all the subtasks AEFT's. We will work with the ALFT in the same way we work with the AEFT. In order to compute the ALFT the original algorithm uses a traversal strategy starting from the exit nodes and going to the entry nodes. If an exit node has no next N(t) in the same resource queue then the ALFT of this particular node will be the DCPL value. Thus the DCPL creates a lower bound at each scheduling step. If has next then the ALFT will be the ALFT of the next minus the computation cost of the next $w(N(t))$ in the current resource queue.

All the exit nodes have no descendants and so there is no dependency that influences the ALFT value in this layer. When the searching of all the exit nodes finishes, having all the subtasks ALFT values computed, the previous layer has to be examined. The exit layer that was examining first depends from this layer. Then the ALFT of a subtask will be the maximum time between its last starting descendant and the starting time of its next N(t) in the same resource queue. That is the point where we can insert communication time in the ALFT value. The new ALFT value denoted by $ALFT_C$ follows:

$$ALFT_c(t) = \begin{cases} DCPL_C \text{ if } Descendant(t) = \varnothing \wedge N(t) = \varnothing \\ \max\left\{ \underbrace{\max_{\forall t' \in Descendant(t)} \left\{ ALFT_c(t') - \omega(t') - C_{t't} \right\}}_{(1)}, \underbrace{ALFT_C(N(t)) - \omega(N(t))}_{(2)} \right\} \end{cases}$$

Again we inserted communication cost only in the first term because t and N(t) belong to the same layer and thus do not communicate.

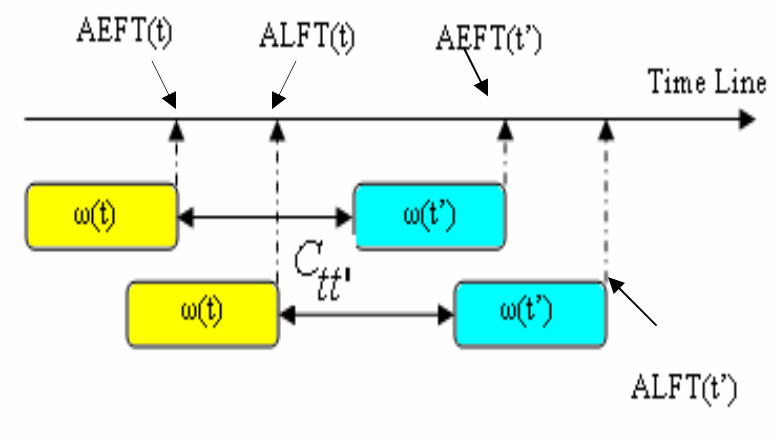


Figure 3.5: Variation of AEFT,ALFT after the communication insertion

In this way we have inserted communication cost in the xDCP structure. We can now use the new metrics together with the remarks from Section 3.3 to propose a new modified version of the xDCP algorithm denoted by $xDCP_C$. The new version of the algorithm follows and the modified parts are highlighted:

$xDCP_C$

1) Use the Shuffle algorithm to stage the all the subtasks in the resources

2) $\forall t_{ij} \in \bigcup_{i=1}^{n} T_i$ set flag $FALSE$

3) $\forall t_{ij} \in \bigcup_{i=1}^{n} T_i$ compute $AEFT_C(t)$, $ALFT_C(t)$ and $DCPL_C$

4) Select $t_{ij} \in \bigcup_{i=1}^{n} T_i$ that orderly follows the next 3 criterions:

a) Minimize the value of i

b) Minimize the $ALFT_C(t_{ij}) - AEFT_C(t_{ij})$

c) Minimize the value of $AEFT_C(t_{ij})$

5) $\forall r_{ij} \in R_i$ select r and the slot in $r's$ queue that assuming t is allocating into that slot orderly satisfying:

a) $ALFT_C(t) - AEFT_C(P(t)) \geq \omega(t)$

b) Minimize the value of $AEFT_C(P(t))$

If there exist such a slot move t or else do not move anything.

6) Set t flag $TRUE$. If $\exists t$ $FALSE$ goto 3)

7) If $(DCPL_{init} / DCPL) * 100\% < 95\%$ goto 2) or else the algorithm terminates.

Chapter 4

Simulating the Grid

4.1 Simulation tools

The research in distributed systems and more precisely in Grid is mainly based on a collection of methodologies and tools. A researcher first observes the system considering all the involved parameters before being able to model it. In large distributed systems numerous parameters must be considered. The interaction between the resources is complex and this makes impractical the effort to create an analytical model. This established the need for high level observation tools. The Simulators belong to these tools. They are focusing in the analysis of a particular system behavior abstracting the rest of it. Simulators are very useful to observe, with high accuracy, local or global characteristics of a distributed system. The advantage they give is the independency from the execution platform. Using Simulators we can easily build a model of a real system in a single PC. Then we are able to experiment with it as it is the real platform, producing results with a high accuracy.

For the evaluation of the $xDCP_C$ algorithm we did not try to create a Simulator from scratch but we looked in the correspond literature to find a Simulator that could fit the needs of our work. Grid Simulators we found in [6][33](SimGrid), [31](GridSim) and [32](OptoSim). All these Simulators try to investigate the dynamic Grid behavior focusing each one on different scheduling strategies. Examining their capabilities we decided to use SimGrid because it is more precise to our needs and allows us to define a DAG abstraction that is fundamental for our work. Thus we will present it analytically in the next Section.

- **GridSim**

The GridSim is a discrete event-driven Simulator implementing with Java on the top of SimJava, a discrete event simulation package written by the Department of Computer Science at the University of Edinburgh. The work in GridSim focuses on the Grid economy where scheduling includes producers and consumers that are the resource owners and the users respectively. Distributed brokers (agents) that each one uses its own

scheduling strategy are trying to find acceptable trade-offs for all the users. The tasks for scheduling are treated as independent. It is a high-level simulator and is mainly used to study cost-time optimization applications for scheduling PST's on heterogeneous Grids dealing with task execution deadline and budget constraints.

- **OptoSim**

The OptoSim is a Simulator especially for the study of replication based scheduling algorithms. The design of OptoSim is based on the EU DataGrid project. It tries to address the lack of generic simulators for Data Grids. The data replication involves the creation of data replicas in different resources in order to optimize the communication cost of the application. The simulator allows the description of the network topology by enumerating the links between resources and the available bandwidth. OptoSim is proper for the investigation of the stability and the behavior of replication based algorithms.

4.2 The SimGrid Simulation Framework

The SimGrid [6][33] is a toolkit that provides core functionalities for the evaluation of scheduling algorithms that target distributed applications in heterogeneous computational Grid environments. It aims to provide the proper level and model abstraction for studying Grid scheduling algorithms and is able to generate correct and accurate simulation results. Simgrid is widely used by many researchers [34][35][36] as a tool to evaluate their work. Simgrid performs event-driven simulation. It assumes that the resources have two performance characteristics:

1) **Latency**
2) **Service rate**

The latency is defined as the time in seconds to access a resource and the service rate is the number of working units performed per time unit. Both can be expressed as constant or using vectors of time stamped values. These vectors are also called traces. An example of such a trace vector follows:

$$\begin{pmatrix} 0.0 & 1.0 \\ 7.0 & 0.8 \\ 11.0 & 0.5 \\ 20.0 & 0.9 \end{pmatrix}$$

The first column represents the time period and the second the rate of performance. The performance changes as it is defined by the time stamps. The user's responsibility is just to define this vector in a text file and after the simulator parses the file during the running time. Traces allow the simulation to be more realistic if we consider that in a real Grid

environment the performance of a resource is possible to vary over time. Traces can only be used for dynamic and not for static scheduling where the priorities are calculated statically at the beginning of scheduling and the resource performance is assumed that is constant.

In SimGrid is not possible to have an interconnection topology between two communicating processors. Both processors and network links are treated as separate tasks. There is no correlation between them. It is the responsibility of the user to define his topology requirements. The user can create links between hosts or between clusters of hosts defining like this his required topology. The next figure illustrates the task model used by SimGrid. The communication edges on the left graph (T1-T4) become tasks creating a new equivalent graph on the right.

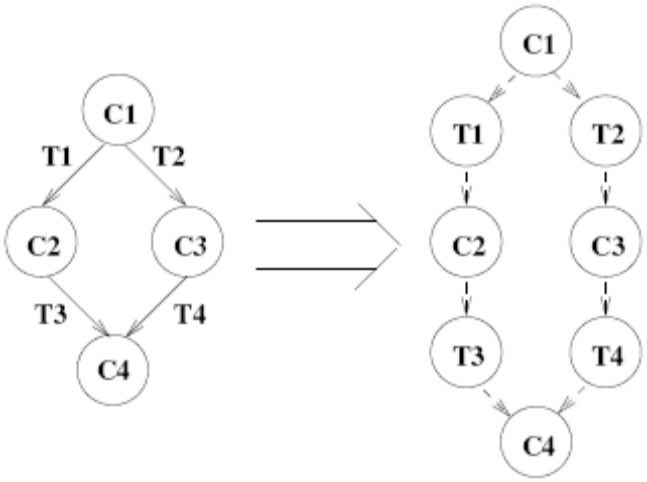


Figure 4.1: SimGrid Task Model

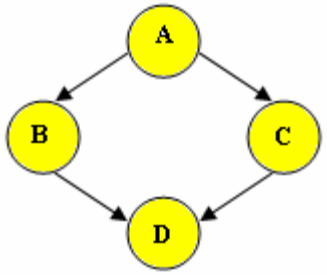
4.3 SimGrid API

The fundamental objects in SimGrid are Tasks and Resources. The SimGrid API is a library implemented in C that can be used to build simulators. In SimGrid tasks are either data transfers or computations. A task is being created using a call to the function:

SG_newTask (SG_task_t *type*, const char * *name*, long double *cost*, void * *metadata*)

The user can define the type (SG_Computation or SG_Tranfer), the state, the name and the cost of a task. It is also possible to insert Metadata to the task using a void* pointer. Metadata can be used to describe a special task characteristic that the API does not allow to define. In the case of data transfer the cost is the data size in bytes and for the computation is the required processing time on the reference processor. The life cycle of a task is defined by its state. A task can be scheduled, not scheduled, ready, running and complete. Using both computations and transfers it is easy to create a workflow graph.

For that reason the function SG_addDependency (SG_Task child, SG_Task parent) is being used to define the dependencies that construct the graph. The way to create a simple task graph is presented in the next Figure.



SG_addDependency(B, A)
SG_addDependency(C, A)
SG_addDependency(D, B)
SG_addDependency(D, C)

Figure 4.2: Simple Graph Definition

Dependencies can be removed by a call to the function:

SG_removeDependency (SG_Task child, SG_Task parent) or
SG_clearAllDependencies ( )

The API offers a wide range of useful task related functions that can give flexibility to the user. Some of them are:

SG_setTaskPriority (SG_Task task, long double priority)

Sets the priority value for a given task. Tasks with higher priority are executed first. The function takes two arguments. The task that we want to assign priority and the priority value we want to set.

SG_getTaskParents (SG_Task child)
SG_getTaskChildren (SG_Task parent)

These functions return a list of a task's children and parents respectively.

Low-level resource objects in SimGrid are hosts and links. A host is described by its computational speed relative to that of a reference host, and by its CPU availability (a value between 0 and 100%). We can create a host by a call to the function:

SG_newHost (const char *name, long double rel_speed, SG_resourcesharing_t policy, const char *cpu_avail, long double cpu_offset, long double fixed_cpu, const char *failure_trace, long double(*failure_function)(long double), long double fixed_failure, void *metadata)

A host can have constant or time varying performance using traces. Links are described by latency and a bandwidth. To create a link we just need a call to the function:

SG_newLink (const char *name, SG_resourcesharing_t policy, const char *latency, long double latency_offset, long double fixed_latency, const char *bandwidth, long double bandwidth_offset, long double fixed_bandwidth, void *metadata)

Again is possible to attach Metadata both in links and hosts to define user-level useful information. The flexibility that SimGrid offers is that it is easy to move from resources with constant performance to dynamic ones by just modifying some parameters of the resource creation function.

The resources in SimGrid can be implemented using three different sharing strategies:

1) First in First Out (FIFO)
2) First Ready First Out (FRFO)
3) Shared

In the FIFO mode tasks are executed in the same order they are assigned to a resource. In the FRFO mode only ready tasks are executed first. The last mode allows all ready tasks to execute concurrently on a resource and the user is able to implement a fair sharing strategy.

Tasks can be assigned to resources with a call to the function:

SG_scheduleTaskOnResource (SG_Task task, SG_Resource resource)

The user has simply to define the task and the target resource. A task can be removed from a resource using a call to the function:

SG_unScheduleTask (SG_Task task)

The user is capable to choose which task is needed to remove without defining the target resource. The function finds the resource automatically and removes the scheduled task.

Having defined tasks, resources and task dependencies a call to SG_simulate() is enough to start the simulation procedure. This function executes tasks through their life time and simulates the resource usage. We are able to run the simulation until all or some tasks complete but also it is possible to simulate the application for a given number of virtual seconds. The SG_simulate returns a list of completed tasks since the last time it was called. This list includes the starting and the finishing time of every completed task as well with the resource that the task was assigned. The models that SimGrid uses for the task execution time(1) and the file transfer(2) time are:

$$(1)\ \frac{ComputationalCost}{CPUSpeed} \qquad (2)\ \frac{DataSize}{LinkBandwidth} - Latency$$

4.4 Using SimGrid to Build a Simulator

The SimGrid API offers a wide range of functions that can help the user to define his system requirements and to simulate a PST application. Putting all together we will demonstrate an example to show how SimGrid can be used to simulate a simple DAG scheduling in a three processors network. In this example we will not implement any scheduling algorithm but we will assign tasks to the processors using a round-robin way.

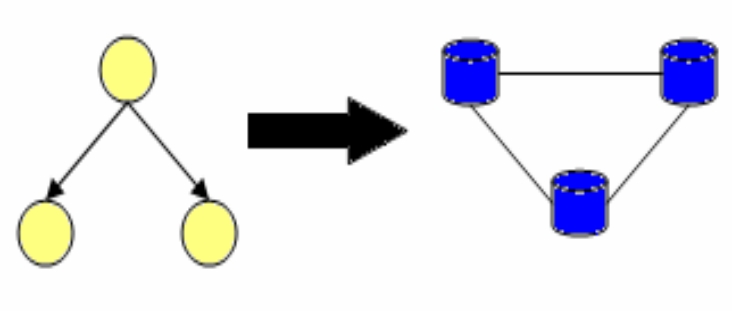


Figure 4.3: Mapping a Simple DAG

```
#include "simgrid.h"

int main(){

//Objects and needed Variables Definition
SG_Resource   Processor1, Processor2, Processor3, Link1, Link2, Link3;
SG_Task   Computation1, Computation2, Computation3;
SG_Task   Tranfer1, Tranfer2;
SG_Task   *Completed_List;
double clock;

//SimGrid Initialization
SG_init();

//Processors Creation
Processor1 = SG_newHost ( "P1", 1.0, SG_SEQUENTIAL_IN_ORDER, NULL, 0.0, 100.0,  NULL);
Processor2 = SG_newHost ( "P2", 1.0, SG_SEQUENTIAL_IN_ORDER, NULL, 0.0, 200.0,  NULL);
Processor3 = SG_newHost ( "P3", 1.0, SG_SEQUENTIAL_IN_ORDER, NULL, 0.0, 150.0,  NULL);

//Links Creation
Link1 = SG_newLink( "L1", SG_FAT_PIPE, NULL, 0.0, 0.0, NULL, 0.0, 10.0, NULL);
Link2 = SG_newLink( "L2", SG_FAT_PIPE, NULL, 0.0, 0.0, NULL, 0.0, 5.0, NULL);
Link3 = SG_newLink( "L3", SG_FAT_PIPE, NULL, 0.0, 0.0, NULL, 0.0, 8.0, NULL);

//Tasks Creation
Computation1 = SG_newTask(SG_COMPUTATION, "Comp1", 50.0, NULL);
Computation2 = SG_newTask(SG_ COMPUTATION, "Comp2", 200.0, NULL);
Computation3 = SG_newTask(SG_ COMPUTATION, "Comp3", 100.0, NULL);
Tranfer1 =  SG_newTask(SG_TRANFER, "Com1", 10.0, NULL);
Tranfer2 =  SG_newTask(SG_TRANFER, "Com2", 5.0, NULL);

//DAG Dependencies Definition
SG_addDependency( Com1, Comp1);
```

```
SG_addDependency( Com2, Comp1);
SG_addDependency( Comp2, Com1);
SG_addDependency( Comp3, Com2);

//Scheduling Tasks one on each Processor
SG_scheduleTaskOnResource( Comp1,P1);
SG_scheduleTaskOnResource( Comp2,P2);
SG_scheduleTaskOnResource( Comp3,P3);

//Schedule File Transfers
SG_scheduleTaskOnResource( Com1,L1);
SG_scheduleTaskOnResource( Com2,L2);

//Simulate until All Tasks Complete and Print the Total Execution Time
Completed_List = SG_simulate(-1.0, SG_ALL_TASKS, SG_VERBOSE);
clock = SG_getClock();
fprintf(stderr, "** Virtual clock = %f\n", clock);

//Free Used Memory and Exit
SG_clear();
free(Completed_List);
exit(0);
}
```

Instead a scheduling algorithm can be implemented here

This code can easily be used as a template to build a simulator using the SimGrid library. It describes all the fundamental aspects of this procedure. First tasks and resources are defined. Afterwards the task/file dependencies are created. Then it is up to the scheduling strategy that is used to take decisions about how to arrange tasks(computations or file transfers) to resources(CPU's or links). The simulation starts with a call to SG_simulate and the total simulation time is printed on screen with the list of all completed tasks.

4.5 Simulation System Specification

We used the SimGrid functionalities to model our system and it is mainly based on the work discussed in [18]. The Computational Grid is defined that is a set of n unrelated heterogeneous computation clusters $\{G_i\}$, $\{i = 1...n\}$. Our heterogeneous machine model is based on Moore's law. This law claims that computers double their speed every 18 months. If we consider that a computer lifetime is about 5 years, in this period of time the faster computer will be 8 times faster than his oldest ancestor speed. Thus we will use a uniform distribution for the machines speed ranged between a current number and a number 8 times bigger (e.g. U(200, 1600)).

We also assume without any loss of generality that each of these clusters is connected with a local storage facility. We have to make this assumption in order to reduce the communication overhead that will create the multiple transfer of the same file on a link. We can realize the need of this storage facility If we think about the case of a

subtask that has three successors in the next PST layer. This subtask will send the same file to all its successors to start executing. Knowing that the subtasks on any PST layer are independent, the file has to be send to each successor separately. Sending the file three times on the link will create match more delay than send it once and store it in a local storage facility. The time the successors subtasks will need to access this file from the local storage facility, is negligible for a normal file size.

The user is able to access these cluster with n links. The independency of the subtasks on the same layer will allow us not to model any inner-cluster connection between resources. We will not try to model precisely the network contention by implementing routers within the links. The current SimGrid API does not give that flexibility and the user has to define everything manually. This could be done automatically using MSG (Meta-SimGrid), a simulator build at he top of SG but it targets to schedule independent tasks that share files and does not allow us to define a DAG structure. The new SimGrid API that is under development will include this flexibility and a more precisely network description can be scheduled as a future work. The user is assumed that can have access to a GIS (Grid Information Service) that can give him information about the availability and the performance of the resources. The next figure illustrates the Grid system that is used for the evaluation of the proposed modified algorithm.

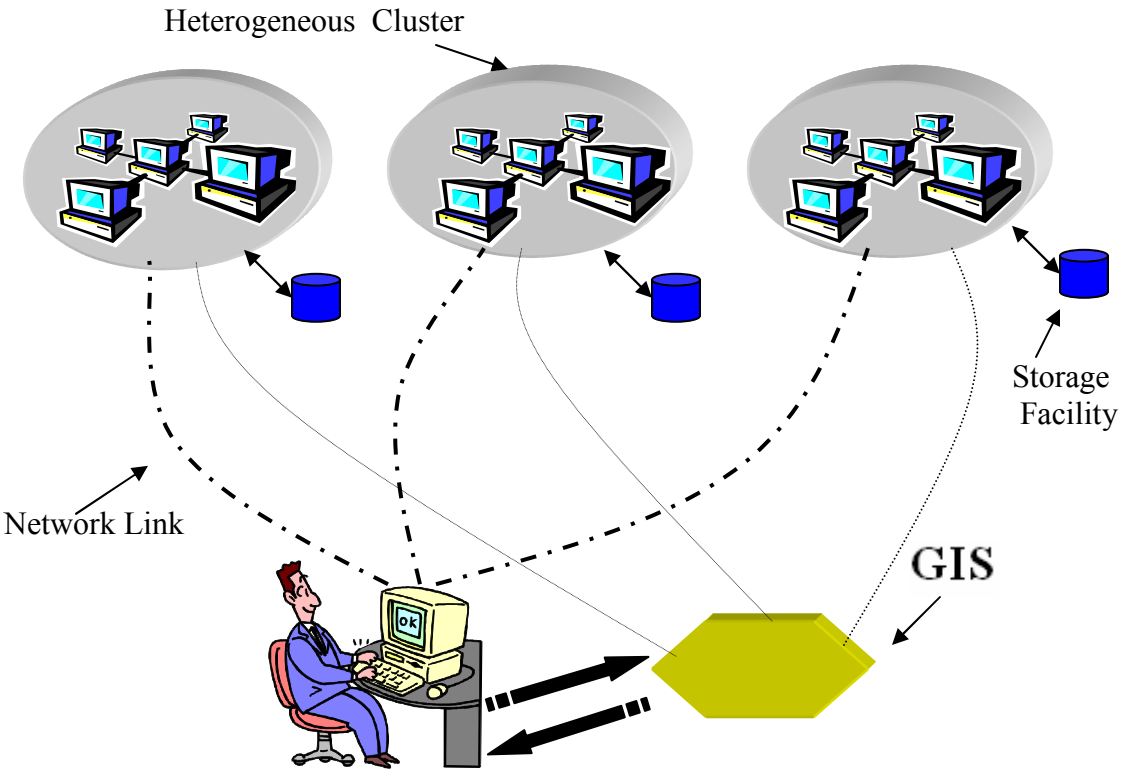


Figure 4.4: Grid System Specification