Scheduling Methods for Divisible Workloads in Distributed Environments

by

Nguyen The Loc

submitted to
Japan Advanced Institute of Science and Technology
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

Supervisor: Professor Takuya Katayama

School of Information Science
Japan Advanced Institute of Science and Technology

March 4, 2007

Abstract

Divisible Load is a kind of workload that can be divided into arbitrary, independent chunks. By definition, Divisible Load is "a load may be arbitrarily divisible which has the characteristic that they can be arbitrarily partitioned into any number of load fractions." We can see this kind of workload in many domains of science and technology such as:

- Protein sequence analysis
- Simulator of Cellular Micro physiology
- Parallel and Distributed Image Processing, Video Processing, and Multimedia

Because of total workload may be as much as a single computer can't execute itself, so we have to share the total workload between many available workstations in distributed environments such as Grids. At this point, a problem, called scheduling problem, arises to pose the following question: how to divide the total workload, residing at a computer called master, into many parts and assign them to computers of the Grid, hereafter called workers, so that the execution time (makespan) is minimum.

The main issue of scheduling process is to find an optimal division of total workload into workers. Up to now many approaches have been proposed. A simple solution, called *Single Round*, divides the workload in as many part as workers, then sends each part to appropriate worker. Because each worker receives its load only one time, this method is named Single Round. Another approach, called *Multi Round*, splits the overall process into many consecutive rounds. In each round, the master delivers chunks to each of workers in turn.

One apparent shortcoming in many scheduling algorithms that exist in the literature is the abandon of designing a solid selection policy for generating the best subset of available workers. Part of the reason is that the main focus of these algorithms is confined to the LAN environment, which makes them not perfectly suitable for a WAN environment such as the Grid [1]. In the Grid, resource computing (workers) join and leave the computing platform dynamically. Unlike other algorithms, we cannot assume in the Grid that all available resources, which may be in thousands, must participate in the scheduling process. The more recent algorithms very tersely allude to this problem by proposing primitive intuitive solutions that are not back up by any analytical model.

In the first part of this dissertation, we propose a new scheduling algorithm, MRRS (inspired by existing algorithm called UMR), which is better and more realistic. MRRS is superior to UMR with respect to two aspects. First, unlike UMR that relies primarily in its computation on the CPU speed, MRRS factors in several other parameters, such as bandwidth capacity and all types of latencies (computation and communication) which renders the MRRS a more realistic model. Second, the MRRS is equipped with a worker selection policy that finds out the best workers. As a result, our experiments show that our MRRS algorithm outperforms previously proposed algorithms including the UMR.

However, all of above approaches assume that computational resources at workers are dedicated. This assumption renders these algorithms impractical in distributed environments such as Grids where computational resources are expected to serve local tasks in addition to the Grid tasks. The inevitable variation of workers' power in the Grid embodies a non-trivial challenge for scheduling (split and distribute) workloads to workers.

The second purpose of this dissertation is to develop an efficient multi-round scheduling method for non-dedicated environments such as Grids. In order to find the optimal division of workload in each round, we need to forecast, as accurate as possible, the available CPU power of each worker before the division happens. We develop a performance model to represent a worker's activity with respect to processing local and external tasks. This model help us to estimate the computing power of a worker under the fluctuation of number of local and Grid applications in the system. Based on this model, we propose a new strategy for predicting the computing power of processors. After that we design a dynamic scheduling algorithm incorporates the performance model and the prediction strategy into the static algorithm MRRS that mentioned above.

As an alternative method we apply an existing prediction algorithm, Mixed Tendency-Based Prediction, in developing a new dynamic scheduling algorithm. The Mixed Tendency-Based Prediction is integrated into the static algorithm MRRS in order to partition the workload in non-dedicated environments.

At last, we describe the experiments for comparing between proposed dynamic and static scheduling algorithms as well as for comparing the proposed algorithms with the existing scheduling algorithm.

Acknowledgments

First of all, I would like to express my sincere gratitude to my principal supervisor, Professor Takuya Katayama of Japan Advanced Institude of Science and Technology (JAIST), for his constant encouragement and kind guidance during this work. I would like to express my profuse thanks to my advisor, Professor Ho Tu Bao of School of Knowledge Science (JAIST), for his valuable suggestions during the period of my PhD. study.

I wish to continue my gratitude to Associate Professor Xavier Defago, Professor Yasushi Hibino, Professor Kunihiko Hiraishi, Associate Professor Ryuhei Uehara for gladly agreeing to serve as members of my dissertation committee and for providing helpful advice and constructive comments that have helped to significantly improve the presentation of my thesis.

I wish to express my gratitude to Assistant Professor Said Selim Elnaffar of College of IT, UAE University (UAE) who has had a great influence on my work, for his kindness and valuable suggestions during the period of my study.

Also, I would like to take this opportunity to thank all the teaching staffs at School of Information Science at JAIST for teaching me a worth of knowledge. I am greatly indebted to the Graduate Research Program (JAIST) for the financial support I have received for staying and studying in JAIST.

I would like to thank Dr. Nguyen Truong Thang and Vietnamese members in JAIST for their helps and supports not only in my research but also in my living life during last three years.

On this occasion, I wish to thank Dr. Nguyen Thi Tinh and Dr. Ho Cam Ha of Faculty on Information Technology, Hanoi University of Education (Vietnam) for their encouragements and helps.

I am grateful to all who have affected or suggested his areas of research. I devote my sincere thanks and appreciation to all of them, and my institute.

Statement of Originality

I, Nguyen The Loc, hereby certify that this PhD dissertation is original and all the ideas and inventions attributed to others have been properly referenced.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Scheduling Divisible Load in Distributed Platform

Applications in many scientific and engineering domains are structured in large numbers of tasks. Scheduling these tasks on a distributed computing platform efficiently is critical for achieving high performance. With a common distributed infrastructure, scheduling is to answer the question: given a set of applications (usually computing intensive), how to schedule them over multiple decentralized resources? In mapping tasks of applications to resources (CPU power, memory, storage devices capacities, etc.) we have several basic questions to deal with:

- How do the relations between tasks affect scheduling decisions?
- How does the heterogeneity of resources affect the performance of a schedule?
- What performance models should a scheduler use to determine the quality of a schedule?

Tasks can be dependent on each other, which makes scheduling more difficult. In fact, most task scheduling research efforts have been dealing with independent task/divisible workloads or loosely-dependent task scheduling. These approaches typically use either analytical methods to make deterministic schedules (usually an optimal schedule solution) or empirical data analysis and heuristic search methods to look for a good solution.

Consider the simpler local scheduling problem. Four major objectives of local schedulers have been:

- overcoming heterogeneous of computing resources.
- maximizing overall system performance, such as cycle harvesting, high throughput, and high resource utilization rate.
- supporting various computing intensive applications, such as batch jobs and parallel applications (MPI, PVM,...).
- providing robust remote job execution functionalities, such as reliable remote I/O and efficient job management involving job preemption, migration, and check pointing.

The result is a system-centric resource management system, which provides a reliable local distributed computing environment while maximizing resource utilization of a local site. Its resource scheduling is usually an opportunistic matchmaking process that requires some way to specify and express application requirements and resource advertisement.

Scheduling in distributed environments such as Grids [1] is more complicated than local resource scheduling because it must manipulate large-scale resources across management boundaries. In such a dynamic distributed computing environment, resource availability varies dramatically. So scheduling becomes quite challenging. There has been extensive research work on scheduling problems in distributed systems.

The scheduling problem has been studied for a variety of application models, such as the well-known DAG [2] (Directed Acyclic Graph task) model. Another popular application model is that of independent tasks with no task synchronization and no task communication. Applications conforming to this simple model arise in most fields of science and engineering. A possible model for independent tasks is one for which the number of tasks and the task sizes, i.e. their computational costs, are predefined. In this case a number of scheduling heuristics have been proposed in the literature [3].

Another kind of the independent tasks model is one in which the number of tasks and the task sizes can be chosen arbitrarily. This corresponds to the case when the application consists of an amount of computation, or load, which can be divided into any number of independent pieces. This corresponds to a perfectly parallel job: any sub-task can itself be processed in parallel, and on any number of workers. In practice, this model is an approximation of an application that consists of large numbers of identical, low-granularity computations. This divisible load model has been widely studied in the last several years, and *Divisible Load Theory* (DLT) has been popularized by the book written in 1996 by Bharadwaj, Ghose, Mani and Robertazzi [4]. There exists a vast literature on DLT. In addition to the landmark book [4], two introductory surveys have been published recently [5, 6]. Furthermore, a special issue of the Cluster Computing journal is entirely devoted to divisible load scheduling [5], and a Web page collecting DLT-related papers is maintained [7].

DLT provides a practical framework for the mapping on independent tasks onto heterogeneous platforms, and has been applied to a large field of scientific problems, including:

- Kalman filtering [8],
- image processing [9],
- video and multimedia broadcasting [10, 11],
- database searching [12, 13],
- processing of large distributed files [14],
- ...

These applications are amenable to the simple master-worker [4] programming model and can thus be easily implemented and deployed on computing platforms ranging from LAN to computational Grid.

1.2 Motivation and Objectives

Per the Divisible Load Theory [4], the scheduling problem is identified as: *"Given an arbitrary divisible workload, in what proportion should the workload be partitioned and distributed among the workers so that the entire workload is processed in the shortest possible time"*. There are 2 kinds of scheduling for divisible loads: single-round algorithm and multi-round one. Single-round algorithms [12, 15] are an early and simple way. As shown in [4], for a large workload, the single-round approach is not efficient due to the large idle time attributed to the last worker as it waits for receiving its workload chunk. Because of we aim at wide distributed systems, thus our study is motivated to follow the multi-round scheduling direction.

Multi-round algorithms, first introduced by Bharadwaj [4], further utilize the overlapping between communication and computation processes at workers. Beaumont [16] proposes a multi round scheduling algorithm that spends a fixed execution time for each round. This enabled the author to give analytical proof of the algorithm's asymptotic optimality. Yang et al. [17, 18], through their UMR (Uniform Multi-Round) algorithm, designed a better algorithm that extends the MI by considering latencies.

However, most of above scheduling approaches are static approaches, i.e. they assume that computational resources at workers are dedicated. This assumption renders these algorithms impractical in distributed environments such as Grids where computational resources are expected to serve local tasks in addition to the Grid tasks. Hence, our **goal** is developing an efficient scheduling method for non-dedicated distributed environments such as Grids.

The variation of worker's power in the Grid embodies a non-trivial challenge for scheduling (split and distribute) workloads to workers. During the execution of a Grid task on a certain worker, some local tasks may arrive causing to interrupt the execution of the lower priority Grid tasks. Our **first objective** is using proposed queuing model to represent the Grid task processes occurs at workers under the effect of local tasks. Based on this theoretical model, we estimate the portion of original CPU power that the workers can donate to Grid applications, here is our **second objective**.

Per the Scheduling Theory [3], any scheduling algorithm should address the following issues:

- Workload Partitioning Problem. This problem is concerned with the method by which the algorithm should divide the workload in order to dispatch to workers.
- Resource Selection Problem. This problem is concerned with how to select the best set of workers that can process the workload partitions such that the execution time (hereafter referred to as *makespan*) is minimal.

In the Grid, resource computations (workers) join and leave the computing platform dynamically. Unlike the case of a dedicated environment, we cannot assume in the Grid that all available resources, which may be in thousands, must participate in the scheduling process. The scheduler obtain the best performance is the one with the ability to find out the best group of workers among whole of available workers. Thus, our **last objective** is the selection of the best subset of workers that is suitable for the current configuration of platform.

1.3 Related Works

There are many scheduling approaches as following:

- Single round algorithm [12, 15] is the early and most simple way for the scheduling problem. The strategy is to utilize the overlapping between communication and computation processes at workers. As showed in [15], for a large workload, the single-round approach is not efficient due to a large idle timing suffered by the last worker to receive its chunk.

- Multi-round algorithm, firstly introduced by Bharadwaj [4], is the way to further utilize the overlapping between communication and computation processes at workers. The general policy of multi-round algorithms let as many workers start to execute as quickly as possible, then keep them and communication channel busy in the following rounds. The initial chunk of loads in the first round often be small to get all workers started. The chunks of loads in the next rounds will be larger, and decrease in the last round. In a special case, with some additional constraints, some multi-round algorithm can reach to near-optimal close forms such as in [16, 17].

- While the above algorithm are static, i.e. the performance of workers are assumed to be stable during their processing, RUMR [19] is designed to tolerate performance prediction errors by using Factoring method, however all of its parameters are fixed before RUMR starts, which makes RUMR a non-adaptive scheduling algorithm. In [2], the authors use M/M/1 queue to model the tasks processing, however, [2] lacks an efficient prediction strategy because it is merely based on probability parameters. In addition, the work in [2] does not address the needs of divisible workloads scheduling.

1.4 Contributions

As mentioned above, our goal is to develop scheduling methods for divisible load in non-dedicated distributed environments. We restrict our scope to star-shaped logical network topologies, because they often represent the solution of choice to implement master-worker computations. Furthermore, star network encompasses the case of a bus, which is a homogeneous star network.

The following points constitute the contributions of this dissertation :

- The **first contribution** is building a queuing model to represent worker's activities with respect to processing local and external Grid tasks. Unlike the work done in [4, 16, 17], this model helps us to estimate the computing power of a worker under the fluctuation of number of local and Grid applications in the system. Based on the estimated power of each worker, the proposed scheduling algorithm will decide how to distribute workload chunks to them.

- One apparent shortcoming in the many scheduling algorithms [12, 16, 18] that exist in the literature is the abandon of designing a solid selection policy for generating the best subset of available workers. Part of the reason is that the main focus of these algorithms is confined to the LAN environment, which makes them not

perfectly suitable for a WAN environment such as the Grid [1]. In order to find out the best group of workers, we develop a worker selection policy that finds out the best workers correspond with proposed scheduling algorithm. This is the **second contribution** of the dissertation.

- In order to get the optimal division of workload in each round, we need to forecast, as accurate as possible, the available CPU power of each worker before the division happens. Our **last contribution** is developing a new strategy for predicting the computing power of processors, i.e. the portion of original CPU power that the owner can donate to Grid applications. To the best of our knowledge, this is the first dynamic scheduling algorithm for divisible load in which a prediction strategy is applied. Another important point in comparison with the previous studies is, in proposed queuing model, all kind of latency such as CPU, bandwidth are considered.

By combining above works, we develop a new dynamic scheduling method for divisible load. Unlike [17, 16] where the load partitioning relies on the CPU power only, our scheduling considers all of system parameters such as bandwidth and latency.

1.5 Roadmap

The rest of the dissertation is organized as follows:

- Before studying the new scheduling algorithms, it is important to have an overview about classification of workload in real parallel and distributed applications. **Chapter 2** first introduces main kinds of loads and their applications. After discussion about some alternative kinds of load, we focus to Divisible load, which is the main topic of the dissertation. The definition of Divisible Load is defined, then the statement of the Divisible Load Scheduling problem is presented. We present the Divisible Load Theory (DLT) and describe its wide applications in Distributed computing and Grid computing. We sketch the system models regarding network shape and divisible load communication and computation. Finally, we summarize previous results about the complexity of the Divisible Load Scheduling (DLS) problem in StarLinear network, Homogeneous/Heterogeneous system and Affine cost model.

- Chapter 3 proposes a new scheduling algorithm, MRRS (inspired by an existing algorithm called UMR), which is better and more realistic. MRRS is superior to the previous algorithms with respect to two aspects. First, MRRS considers all platform parameters such as bandwidth capacity and all types of latencies (computation and communication) which renders the MRRS a more realistic model. Second, the MRRS is equipped with a worker selection policy that finds out the best workers. MRRS, to the best of our knowledge, is the first divisible load scheduling algorithm that addresses the resource selection problem. Having such policy is indispensable in large commuting platform such as the Grid, where thousands of workers are accessible but the best subset must be chosen. We, theoretically and experimentally, show that MRRS is superior to previous algorithms such as UMR and LP, specifically in a WAN computing platform such as the Grid.

- Chapter 4 address the issue of divisible load scheduling in Non-dedicate distributed environment. We develop a computation model to represent worker's activities with

respect to processing local and external Grid tasks. The proposed model helps us to estimate the computing power of workers under the fluctuation of a number of local and Grid applications in the system. Based on this computation model we propose a new strategy for predicting the computing power of processors. Proposed dynamic scheduling algorithm incorporates the performance model and the prediction method into the static algorithm MRRS that mentioned in the last Chapter, which is originally a static scheduling algorithm. An alternative way is given by applying an existing prediction algorithm, Mixed Tendency-Based Prediction, in developing a dynamic scheduling algorithm DSA. At last, we describe the experiments for comparing between proposed dynamic scheduling algorithms as well as for comparing the proposed algorithms with the existing static scheduling algorithm.

- Chapter 5 concludes the dissertation with a sketch of the main contributions as well as the drawbacks. We outline some directions for future work, considering areas that might be worthy of further research.

Chapter 2

Scheduling Problem in Distributed Environments

2.1 The Load Scheduling Problem

A scheduling problem addresses the following question: *What is the best possible way to organize a given workload so that it can be completed in the shortest possible time?* Load scheduling problems can be classified in many ways. One of the possible classifications is

- Static scheduling
- Dynamic scheduling

In *static scheduling* approach, the objective is to find an optimal schedule of a given number of tasks to a set of processors or machines. For example, the scheduling of n tasks to a set of m machines so that the time required to process all the tasks is minimum is one such problem. No dynamics of the system are taken into consideration. If the dynamics of the platform are considered, then the scheduling is said to be *dynamic*. In the above example, if tasks arrive at arbitrary time instants, or computation power of machine are not stable, and the scheduling strategy depends on the current state of the platform, then the scheduling is dynamic.

In general, the formulation of a scheduling problem consists mainly of fours steps.

- Modelling the system
- Defining the type of processing loads
- Formulating the objective function (or cost function)
- Specifying the constraints

The model describes the type of computation system, the type of connection, the number of processors, the topology of the network and so on. The type of processing load totally determines the scheduling algorithm. Per [4], the objective of the scheduling problem can be defined as follows *"Given a set of loads and a system, what is the best possible mapping of these loads onto the processors such that the desired cost function is optimized"*. The cost function can be expressed as:

Total cost = Communication cost + Computation cost

Here the total cost refers to the processing time of a load. It can be seen that from the above equation that, the total cost takes into account both of communication and computation cost. The constraints for this problem may be the limitation on the availability of processor and transfer connections.

2.2 Classification of Loads

Scheduling of loads has been categorized as either *job scheduling* or *task scheduling*. A job is defined as a composed of a number of tasks. If a job is assigned to a processor, it is called job scheduling, as introduced in [20]. If different tasks are assigned to different processors, it is called task scheduling, as mentioned in the next chapters. Thus, the kind of scheduling depends primarily on the type of load being processed. In Figure 2.1 we show a classification of loads based upon their divisibility property.

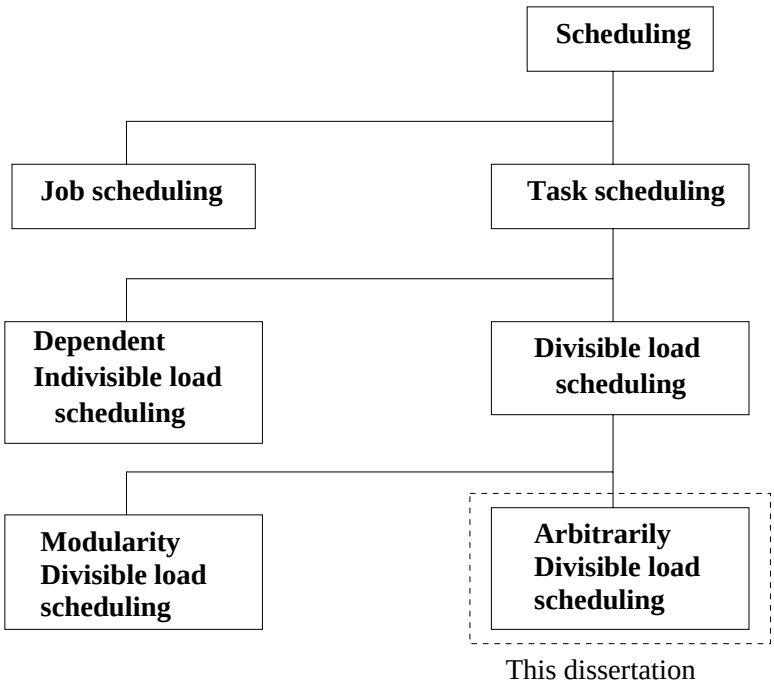


Figure 2.1: Classification of loads

2.3 Job Scheduling

Suppose that m machine $M_j(j = 1, ..., m)$ have to process n jobs $J_i(i = 1, ..., n)$. A schedule assigns for each job an allocation of one or more time intervals to one or more machines. Schedules may be represented by *Gantt charts* as shown in Figure 2.2. Gantt charts may be machine-oriented (Figure 2.2 (a)) or job-oriented (Figure 2.2 (b)).

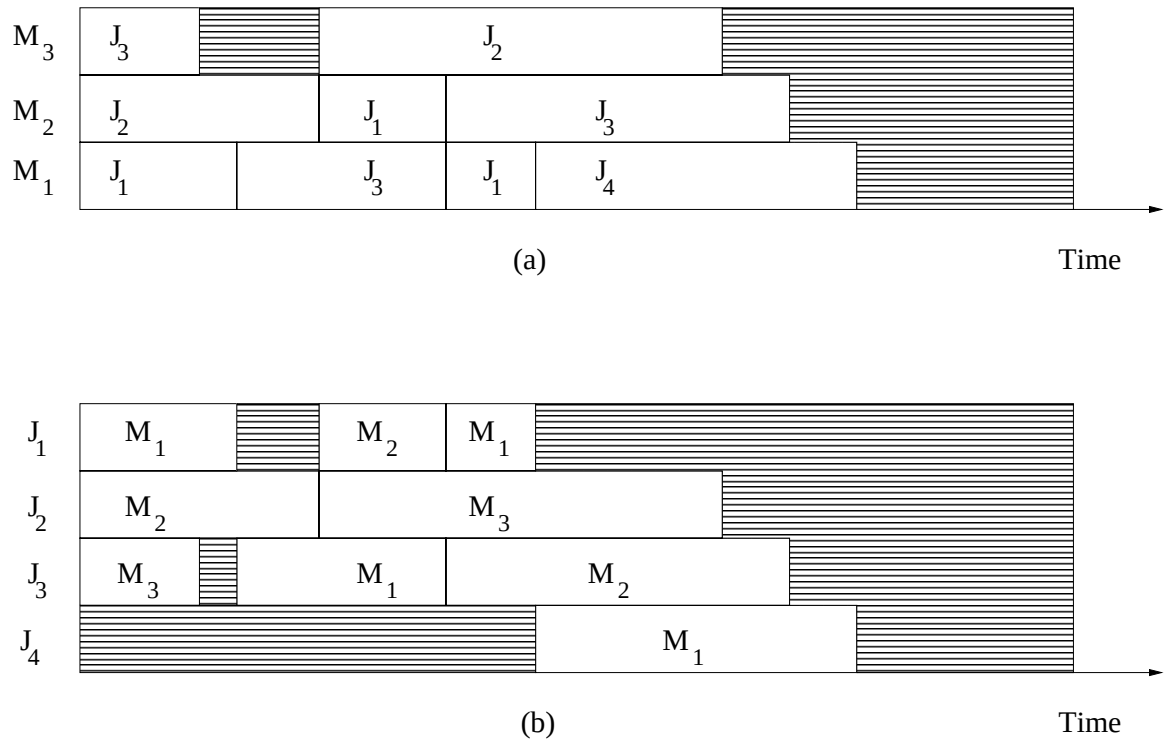


Figure 2.2: An job schedule for 3 machines and 4 jobs

2.4 Task Scheduling

2.4.1 Independent-Indivisible Loads

These loads can not be further divide and has to be processed in its entirety in a single processor. They have not any precedence relation. Independent load scheduling problems are known to be NP-complete and hence only heuristic algorithms can be processed to obtain suboptimal solutions in reasonable time.

Given a set of independent tasks and a set of available resources, independent task scheduling attempts to minimize the total execution time of the task set by finding an optimal mapping of tasks to machines. The metric used to find such a mapping is the estimate of turnaround time or completion time (machine available time + expected time to compute). Finding the best mapping is actually a combinatorial optimization problem, which in this case is NP-hard. The common solution is to use some heuristic search procedure to find a near-optimal schedule quickly. Most of the existing research in this area is quite theoretical, because even heuristic search is often too expensive for large search spaces. In a Grid, however, due to the existence of large number of tasks and resources, a practical and efficient heuristic search algorithm can be a desirable solution for Grid application scheduling.

For a general search procedure, given a task, the estimate of a certain metric is calculated for each machine on the resource list. The task is assigned greedily to the machine with the "best" metric. Then the task is removed from task list and a new task scheduling search starts until all tasks are mapped.

Braun and Siegel [21] investigate different heuristic strategies for independent task scheduling in heterogeneous distributed computing systems using simulation. They in-

clude task heterogeneity and machine heterogeneity as performance evaluation measurements when generating an N*M ETC (expected time to compute) matrix, given N tasks and M machines. Each element ETC_{ij} on the matrix indicates the expected time to complete task i on machine j. This matrix serves as metric (heuristic function) table used in resource selection, combined with machine available time, which changes every time a new task is mapped to the machine. The matrix is generated randomly with bounds indicating different task/machine heterogeneity.

Eleven heuristics are studied in [21]: OLB, MET, MCT, Min-min, Max-min, Duplex, GA, SA, GSA, Tabu, A*. Among them, the following heuristics are very interesting:

- **OLB**: *Opportunistic Load Balancing* is the simplest strategy to assign each task to the next available machine without considering the expected execution time on that machine.

- **MET**: *Minimum Execution Time*, in contrast, only considers the expected execution time of each task on a machine and selects the machine with minimum execution time.

- **MCT**: *Minimum Completion Time* assigns each task to the machine with minimum completion time (machine available time + ETC). This is the most common metric in use.

- **Min-min**: The *Min-min heuristic* is a two step task scheduler. First, select a "best" (with minimum completion time) machine for each task. Second, from all tasks, send the one with minimum completion time for execution. The idea behind Min-min is to send a task to the machine which is available earliest and executes the task fastest.

- **Max-min**: The *Max-min heuristic* takes the same first step as Min-min but send the task with maximum completion time for execution. This strategy is useful in a situation where completion time for tasks varies significantly. Using this heuristic, the tasks with long completion time are scheduled first on the best available machines and executed in parallel with other tasks. This leads to better load-balancing and better total execution time.

- **GA**: A *Genetic Algorithm* is an evolutionary technique for large space search. The general procedure of GA search is as following: 1) *Population generation.* A population is a set of chromosomes. Each chromosome represents a possible solution, which is a mapping sequence between tasks and machines. [21] randomly generate 200 chromosomes; 2) *Chromosome evaluation.* Each chromosome is associated with a *fitness value*, which is the total completion time of the task-machine mapping this chromosome represents. The goal of GA search is to find the chromosome with optimal fitness value; 3) *Crossover* and *mutate* the chromosomes selected based on *selection rules.* The selection rules are analogous to evolutionary selection rules. But this paper randomly selects chromosomes. Crossover is the process of swapping certain subsequences in the selected chromosomes. Mutate is the process of replacing certain subsequences with some task-mapping choices new to the current population. Both crossover and mutation are done randomly in [21]. After crossover and mutation, a new population is generated. Then it will be evaluated, and the process

starts over until some *stopping criteria* are met. The stopping criteria can be, for example, 1) no improvement in recent evaluations; 2) all chromosomes converge to the same mapping; 3) cost bound is met.

- **SA**: *Simulated Annealing* is a search technique based on physical process of annealing, which is the thermal process of obtaining low-energy crystalline states of a solid. The *temperature* is increased to melt solid. If the temperature is slowly decreased, particles of the melted solid arrange themselves locally, in a stable "round"state of a solid. SA theory states that if temperature is slowed sufficiently slowly, the solid will reach thermal equilibrium, which is an optimal state. By analog, the thermal equilibrium is an optimal task-machine mapping (optimization goal), the temperature is the total completion time of a mapping (cost function), and the change of temperature is the process of mapping change. If the next temperature is higher, which means a worse mapping, the next state is accepted with certain exponential probability. The acceptance of "worse" state provides a way to escape local optimality which occurs often in local search.

[21] gives a comprehensive investigation of various heuristic search algorithms on independent task scheduling. The simulation results show that GA heuristic has the overall best performance but with most expensive search time cost. SA is not as efficient as its application in other domain science problems. More research on choosing efficient fitness values and selection rules are needed. For example, all "evolutionary changes" are done randomly without making use of any learned knowledge. Also, more practical experiments should be done to verify the results under a broad range of machine conditions.

Casanova [22] develops the *Xsufferage* heuristic for an application-level scheduler system APPLeS. The conventional Sufferage heuristic uses MCT as metric for a mapping. The rationale behind Sufferage is that a host should be assigned to the task that would "suffer" the most if not assigned to that host. The sufferage value of each task is the difference between the MCT and the second MCT. Casanova found that the Sufferage heuristic does not work well for cluster resources because the MCT on the machines belonging to the same cluster are quite close, which makes the sufferage value approaches zero and eliminates cluster machines from selection. So Xsufferage computes a cluster-MCT to enable the Sufferage heuristic to work in a cluster environment.

Generally speaking, the research on use of heuristic search algorithms on independent task scheduling is preliminary. Deeper understanding of Grid application execution behavior will help develop more practical and efficient heuristic algorithms. In the Grid world, batch jobs are the largest portion of Grid applications, so there is a strong need for further research on heuristic search algorithms in the large search spaces typical of the Grid world.

2.4.2 Modularly Divisible Loads

Per [4], these loads are subdivided into smaller loads or tasks based on some characteristics of the load or the system. These smaller loads are also called tasks, subtasks or modules. The processing of a load is said to be completed when all its modules are processed. Usually these loads are represented as graphs whose vertices correspond to the modules, and whose edges represent interaction between these modules. This modular representation of a load is known as *Task Interaction Graph* (TIG) in the literature. If

these modules are subject to precedence relations, then a directed graph is used. On the other hand, if the graph is not directed, though the modules may exchange information, then it is assumed that they can be executed in any order. They can also be totally independent, in which case they may be modelled as indivisible loads.

Consider a classical case of Modularly divisible loads: an application to be executed on distributed platform consists of many precedence constrained tasks, and the task graph presented by a *Directed Acyclic Graph* (DAG)[23] . In the Directed Acyclic Graph, every node represents a task of the application, and each directed edge represents a communication link between two tasks.

Example: consider a task graph represented by a directed acyclic graph $G = (V, E)$ with every node weight L=20, where V is a set of nodes, E is a set of directed edges, and L is a positive integer. We write a directed edge from node u to node v as (u, v). Figure 2.3 shows an example of a task graph. The numerical value close to node v in Figure 2.3 denotes the weight of v. We assume that every node weight is the same as L. An application program is modelled as a task graph. A node in a task graph represents a task in the application. An edge (u, v) means that the computation of v needs the result of the computation of u. If $(u, v) \in E$, task u is called an *immediate predecessor* of task v. A task that has no immediate predecessor is called an entry task. The length of a path in a task graph is the number of tasks on the path. The level *level(v)* is the length of the longest path from v to an entry task. The longest path in a task graph is called the critical path in the task graph.

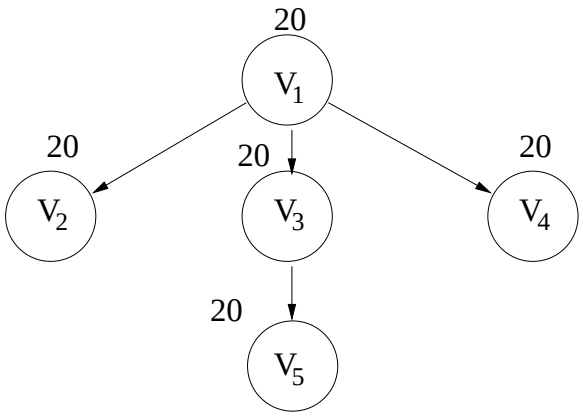


Figure 2.3: Task graph G=(V, E) with every task length 20

Let m be a number of processors in a computational grid. Let $G = (V, E)$ be a task graph with every task length L. We define a schedule of G as follows. A schedule S of G onto a grid with m processors is a finite set of triples $< v, p, t >$ which satisfies the following rules R1, R2, and R3, where $v \in V$, $p(1 \leq p \leq m)$ is the index of a processor, and t is the starting time of task v. A triple $< v, p, t >$ means that the processor p computes the task v between time t and time $t + d$ where d is defined so that the number of instructions computed by the processor p during the time interval $[t, t + d)$ is exactly L. We call $t + d$ the completion time of the task v. Note that starting time and completion time of a task are not necessarily integral.

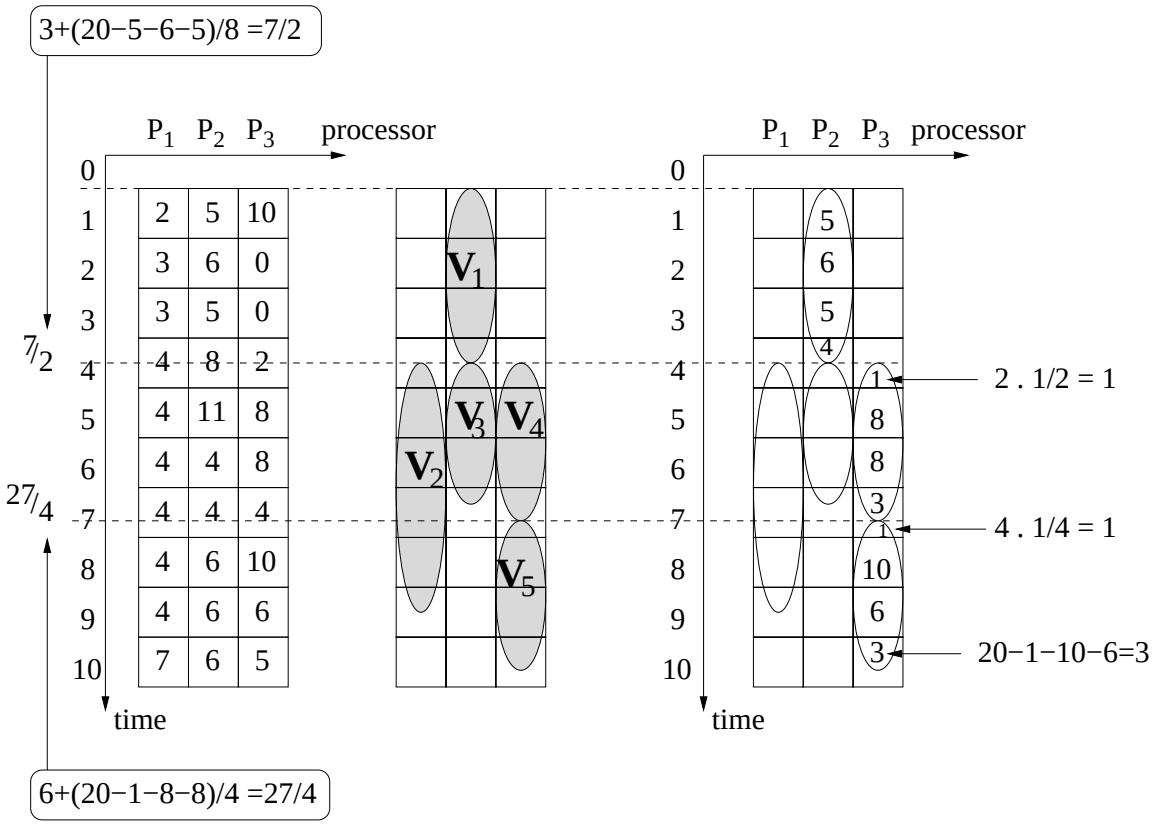
- R1: For each $v \in V$, there is at least one triple $< v, p, t > \in S$.

- R2: There are no two triples $< v, p, t >\in S, < v', p, t' >\in S$ with $t \leq t' \leq t + d$ is the completion time of v.
- R3: If $(u, v) \in E$ and $< v, p, t >\in S$, then there exists a triple $< u, p', t' >\in S$ with $t' + d'$ is the completion time of u.

Informally, the above rules can be stated as follows. The rule $R1$ enforces each task v to be executed at least once. The rule $R2$ says that a processor can execute at most one task at any given time. The rule $R3$ states that any task must receive the required data (if it exists) before its starting time. A triple $< v, p, t >\in S$ is called the task instance of v. Note that $R1$ permits a task to be assigned onto more than one processor. Such a task has more than one task instances. To assign a task onto more than one processor is called task replication. The makespan of S is the maximum completion time of all the task instances in S. For example, Figure 2.4(b) shows a schedule of G, i.e.

$\{< v_1, P_2, 0 >, < v_2, P_1, 7/2 >, < v_3, P_2, 7/2 >, < v_4, P_3, 7/2 >, < v_5, P_3, 27/4 >\}$

The makespan of the schedule is 48/5.



(a) processor speed (b) a schedule of G

Figure 2.4: A schedule for task graph G=(V, E) with makespan = 48/5

2.5 Divisible Loads Scheduling

2.5.1 Introduction

The interest in network-based computing has grown considerably in recent times. In this environment, a number of workstations or computers are linked through a communication network to form a large loosely coupled distributed computing system. One of the major attributes of such a distributed system, apart from its role in storing information in a distributed manner and allowing the use of shared resources, is the capability that it offers to a user at any single node to exploit the considerable power of the complete network or a subset of it by partitioning and transferring its own processing load to the other processors in the network.

This paradigm of load distribution is basically concerned with a single large load which originates or arrives at one of the nodes in the network. The load is massive and requires an enormous amount of time to process given the computing capability of the node. The processor partitions the load into many fractions, keeps one of the fractions for itself to process and sends the rest to its neighbors (or other nodes in the network) for processing. An important problem here is to decide how to achieve a balance in the load distribution between processors so that the computation is completed in the shortest possible time. This balancing can be done at the beginning or dynamically as the computation progresses and the computational requirements become clearer. This framework of computing is suitable for applications that permit the partitioning of the processing load into smaller fractions to be processed independently so that the partial solutions can be consolidated to construct the complete solution to the problem. Obviously not all processing loads satisfy this requirement. But there is a large class of applications that not only permit this kind of processing, but for which it is essential to do so in order to complete the task in time.

In general, scheduling problems discussed in the literature do not attempt to formulate scheduling policies based on the type of loads submitted by an user, except where resource constraints are involved. Usually, the stress has been on designing efficient parallel algorithms in place of conventional sequential algorithms, which requires exploitation of function parallelism in the algorithm. However, there is another kind of parallelism that occurs in the data and is called data parallelism. Such loads can be split and assigned to many processors. But, the manner in which this partitioning (or load division) can be done depends on its divisibility property, that is, the property which determines whether a load can be decomposed into a set of smaller loads or not (Figure 2.1).

Accordingly, loads may be indivisible in which case they are independent, of different sizes, and cannot be further sub-divided. Thus, they have to be processed in their entirety in a single processor. These loads do not have any precedence relations and, in the context of static/deterministic scheduling, they give rise to bin-packing problems that are known to be NP-complete and hence amenable only to heuristic algorithms that yield sub-optimal solutions. In the context of dynamic/stochastic scheduling, these loads arrive at random time instants and have to be assigned to processing nodes based on the state of the system.

Alternatively, a load may be modularly divisible in which case it is a priori subdivided into smaller modules based on some characteristics of the load or the system. The processing of a load is complete when all its modules are processed. Further, the processing of these modules may be subject to precedence relations. Usually such loads

are represented as task interaction graphs whose vertices correspond to the modules, and whose edges represent interaction between these modules and perhaps also the precedence relationships.

Finally, a load may be arbitrarily divisible [5] which has the property that all elements in the load demand an identical type of processing. These loads have the characteristic that they can be arbitrarily partitioned into any number of load fractions. These load fractions may or may not have precedence relations. For example, in the case of Kalman filtering applications, the data is arbitrarily divisible but there may exist precedence relation among these data segments or load fractions. On the other hand, if the load fractions do not have precedence relations, then each load fraction can be independently processed. This latter type of loads is the ones which are of interest to us. Applications which satisfy this divisibility property include processing of massive experimental data, image processing applications like feature extraction and edge detection, and signal processing applications like extraction of signals buried in noise from multidimensional data collected over large spans of time, computation of Hough transforms, and matrix computations.

Traditionally, the parallelism inherent in these problems was exploited through parallel algorithms. Now with the availability of distributed computing systems it is realized that these parallel algorithms can be mapped on to network based computing. However, this is not a straightforward mapping since many of the special properties and limitations of distributed systems affect the performance of the load distribution algorithms. One such factor is the communication delay which is considerably higher in a distributed network environment (that has distributed memory machines with message-passing architecture) than in a parallel processing environment (which has a shared memory architecture). This is especially true for the processing of divisible loads since there is very little interprocessor communication during the actual computation process.

2.5.2 Divisible Load Theory

Studies in *Divisible Load Theory* (DLT) area address the following question: *Given an arbitrary divisible load without precedence relations and a multi processor/ multicomputer system subject to communications delays, in what proportion should the processing load be partitioned and distributed among the processors so that the entire load is processed in the shortest possible time?* One the major issue is that of computation-communication trade-off relationships. The answer to the above question depends on this issue and we will devote the remains of this chapter to discuss about it.

The increasing prevalence of multiprocessor systems and data-intensive computing has created a need for efficient scheduling of computing loads, especially parallel loads that are divisible among processors. During the past decade, divisible load theory has emerged as a powerful tool for modelling data-intensive computational problems. DLT originated from a desire to create intelligent sensor networks, but most recent applications involve parallel and distributed computing.

Simple Divisible Load Theory Example

Consider a five-processor star network [6] with root processor P_0 processing some load itself while simultaneously distributing the rest of the load to processors P_1 through P_4. Let α_i be the fraction of load processed by each processor. The system parameters are

w_i, the inverse computing speed of the ith processor; z_i, the inverse transmission speed of the ith link; T_{cp}, the computation intensity; and T_{cm}, the communication intensity. Thus, $\alpha_i w_i T_{cp}$ is the time to process the ith load fragment on the ith processor and $\alpha_i w_i T_{cm}$ is the time to transmit the ith load fragment on the ith link.

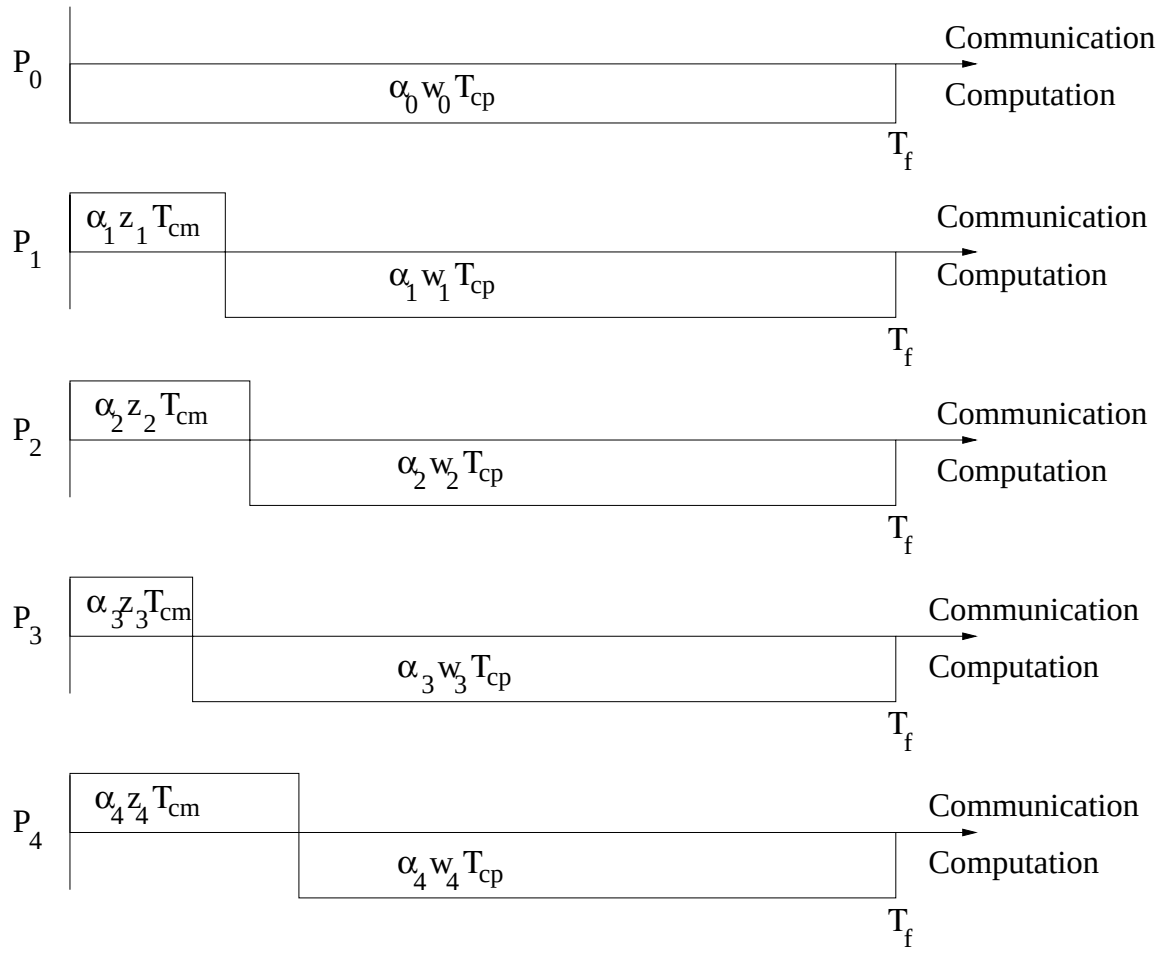


Figure 2.5: Gantt-chart-like timing diagram for star topology. Transmission commences simultaneously on all links, and computation follows load reception on each processor.

As Figure 2.5 shows, a Gantt-chart-like timing diagram can represent a schedule in which transmission commences simultaneously on all links and computation follows load reception on each processor. For a minimum time solution, all processors must stop computing at the same instant; otherwise load could be transferred from busy to idle processors.

To determine the optimal fragment size for processors 1 through 4, set the equation for the solution time of processor P_0 equal to that for processor P_1, the solution time equation for P_1 equal to that for P_2, and so on. Chaining together the load fragment size solutions results in a complete solution, where m is the number of satellite processors:

$$\alpha_i = \left(\frac{z_{i-1} T_{cm} + w_{i-1} T_{cp}}{z_i T_{cm} + w_i T_{cp}} \right) \alpha_{i-1} = f_{i-1} \alpha_{i-1} = \left(\prod_{j=1}^{i-1} f_j \right) \alpha_i \quad (i = 2, 3...m) \tag{2.1}$$

For processor P_0:

$$\alpha_0 = \left(\frac{z_1 T_{cm} + w_1 T_{cp}}{w_0 T_{cp}} \right) \alpha_1 = \left(\frac{1}{k_0} \right) \alpha_1 \tag{2.2}$$

We can then use normalization ($\alpha_0 + \alpha_1 + ... + \alpha_m = 1$) to solve for all the optimal load fractions. If all processor and link speeds are the same, the time to complete a solution is:

$$T_{finish} = \alpha_0 w_0 T_{cp} = \left(\frac{1}{m + 1/k_0} \right) (z T_{cm} + w T_{cp}) \tag{2.3}$$

Finally, the linear speedup is:

$$Speedup = 1 + k_0 m \tag{2.4}$$

The same methodology can handle sequential load distribution, sequential load distribution with installments, simultaneous load distribution in which the root does no processing, and simultaneous load distribution in which computation and communication commence at the same time. We summarize the most important features of DLT as following.

A tractable model

The optimality principle provides the key to divisible load scheduling. Setting up a continuous-variable model and assuming that all processors stop computing at the same instant lets you determine the optimal amount of total load to assign to each processor or link using a set of linear equations or, as in queuing theory and many other cases, recursive equations. DLT can thus account for heterogeneous computer and link speeds, interconnection topology, and scheduling policy. It can also include fixed delays, such as propagation delay in links. Moreover, as the above example shows (Figure 2.5), the model can use Gantt-chart-like schematics to easily portray loads with different computation and communication intensities.

DLT's tractable nature contrasts with the traditional indivisible load problem. That is, when you assign atomic jobs or tasks that each must run on a single processor, combinatorial optimization is often NP complete. Precedence relations provide an additional complication. Although not applicable to all computer-scheduling problems, DLT does apply to an important class of such problems in grid computing; signal, sensor, and experimental data processing; and data-intensive and data-parallel computing.

Interconnection topologies

Over the years, researchers have successfully applied divisible load modelling to a wide variety of interconnection topologies, including linear daisy chains, trees, buses, hypercubes, and two and three-dimensional meshes. Figure 2.6, for example, illustrates a possible load distribution flow originating from a single processor in a 2D mesh network. In addition, asymptotic results developed for infinite-sized networks are useful in sequential load distribution as speedup saturates with the addition of more processors.

The ability to guarantee performance close to that of an infinite-sized network with a small to moderate number of processors therefore provides useful design information. It is worth noting, however, that finding an optimal schedule occurs in the context of a specific interconnection network and scheduling strategy.

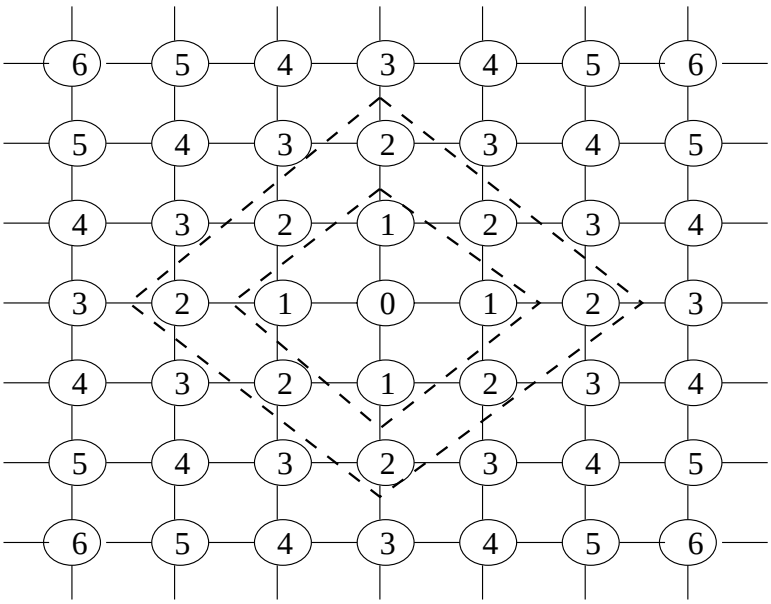


Figure 2.6: Load distribution flow in a 2D network. The load originates at node 0 and propagates throughout the mesh in a diamond-shape pattern.

Equivalent networks

Like other linear theories, including Markovian queuing theory and resistive electric circuit theory, DLT represents a complex network with an exactly equivalent network element. For some network topologies such as trees, aggregation can be recursive, one subtree at a time.

For example, consider either a pair of adjacent processors and their connecting link in a linear daisy-chain network or a single-level subtree in a multilevel-tree network. You first set the computing speed of a single equivalent processor equal to this subnetworks speed and then continue aggregating subnetworks of processors, including intermediate equivalent processors, until one processor is left with a computing speed equivalent to the original network. Final expressions for equivalent processor computing speed can be either closed form or iterative.

Installments and sequencing

A number of applied optimization problems arise in divisible load scheduling. For example, instead of a node in a tree sequentially distributing load to its children, improved performance results if load is distributed in installmentssome to child 1, child 2, child M; more to child 1, child 2, child M; and so on. Performance under sequential multi-installment load distribution strategies tends to saturate as the number of installments increases.

Some sequencing results are surprising. For example, consider a linear daisy-chain network in which all processors and links have the same speed. Under one basic sequential scheduling strategy, if load originates at any interior processor, the same solution time results whether load is first distributed to the left or right parts of the network. Other results are more intuitive. For example, distributing load over a slow link to a relatively fast processor can degrade overall network solution time.

Scalability

Early DLT studies determined that if load is distributed from one node to its children sequentially,as in a tree network, speedup saturates as more nodes are added. If link speed is of the same order as processor speed, optimal sequential load distribution offers a 20 to 40 percent improvement in overall solution time compared to equally dividing the load among processors.

Although simply increasing the number of installments also saturates performance, recent studies indicate that speedup is scalable if a node transmits load simultaneously to all its children, that is, speedup grows linearly in the number of children. As long as a node CPU can load output buffers to all links, performance scales. While there is qualitative support for this scalability concept in parallel processing, DLT allows a quantitative solution.

Metacomputing accounting

A devilish metacomputing problemdistributed computing with payment to computer owners challenges developers to factor problem size and system parameters into monetary accounting. DLT can incorporate an intuitive linear model for computing and communication costs. Simple to moderately complex heuristic rules can be developed to efficiently assign load in terms of both cost and performance. DLT allows using similar rules for a related problem in parallel processor configuration design, namely how to optimally arrange links and processors with certain characteristicsfor example, speed and costin a given topology.

Time-varying modelling

The actual effort a computer can devote to a divisible job depends on the status of other background jobs. Ongoing transmissions likewise reduce a links capacity to transmit part of the job. Developers can use integral calculus to apply solution time optimization to divisible loads if they know the start and end times and effort of such background jobs and messaging. With less than perfect knowledge of background processes, stochastic modelling can be combined with deterministic DLT.

Unknown system parameters

It can be difficult to obtain accurate estimates of available processor effort and link capacity, which are key inputs to divisible load scheduling models. Several recently proposed probing strategies send some small fraction of a load to processors across a network of links to estimate currently available processing capacity at nodes and bandwidth on links. Actual implementations must account for the time-varying nature of available processor effort and link capacity as well as processors release timesthe times at which processors become free to accept additional load. Further, load must be distributed on the fastest processors and links. Nevertheless, these probing strategies offer a promising approach to robust divisible load scheduling.

Extending realism

In recent years, researchers have attempted to generalize divisible load scheduling by considering systems with finite buffers, finite job granularity, scheduling with processor release times, and scheduling multiple divisible loads. Other efforts have sought to synthesize deterministic divisible load modelling and stochastic modelling. Specialized applications of divisible load scheduling include databases and multimedia systems.

Experimental results

Experiments with actual distributed computer systems demonstrate that DLT can be a useful prediction tool, as the Divisible Load Theory Experimental Work sidebar illustrates. With investigators scrambling to initiate DLT research in various sub areas, there is a need to integrate work to datefor example, to assess time varying load sharing on hypercubes. In addition, analytical proofs of divisible load optimality exist for only a limited subset of topologies and scheduling policies. While there is no reason to believe that the principle doesnt hold in other environments, rigorous proofs are yet to follow. Beyond these refinements, several potential breakthroughs are on the horizon.

2.5.3 Divisible Load Applications

Feature extraction and edge detection in image processing

A typical divisible load scheduling application might involve a credit card company that must process 30 million accounts each month. The company could conceivably send 300,000 records to each of 100 processors, but simply splitting the load equally among processors does not take into account different computer and communication link speeds, the scheduling policy, or the interconnection network. Divisible load theory provides the mathematical machinery to do time-optimal processing.

Similarly, banks, insurance companies, and online services often must process large numbers of customer records for billing, data mining, or targeted direct mail advertising, or to evaluate the profitability of new policies. A midsize cap fund would likewise have to process many complex financial records to make the best investment decisions or evaluate new investment strategies.

In computer vision systems, Image Feature Extraction [4, 24] is an extremely important function. This basically consists of two levels of processing, namely, a local computation followed by a nonlocal interprocessor communication and computation. The first level of computation partitions the given image into many segments. Each of these segments is processed locally and independently on different processors. This is done to extract local features of the image from different processors. In the second level of computation, these features from different processors are exchanged and processed to extract the desired feature.

Similarly, edge detection is a very well-known problem in image processing. Here the objective is to detect the edge or boundary of an image. As before, the given image can be arbitrarily partitioned into several subframes of varying sizes and each of these subframes can be processed independently.

A practical situation in which processing of such data may frequently be necessary involves the space shuttle orbiter, which collects massive volume of image data that has

to be communicated to the earth station for processing (by a parallel or distributed system). This kind of data also has the potential of arbitrary divisibility. The data can be partitioned and sent to a number of processors situated at various geographical points on the surface of the earth, in which case they incur considerable communication delay. Depending on the location of the processing units the communication delays will be different.

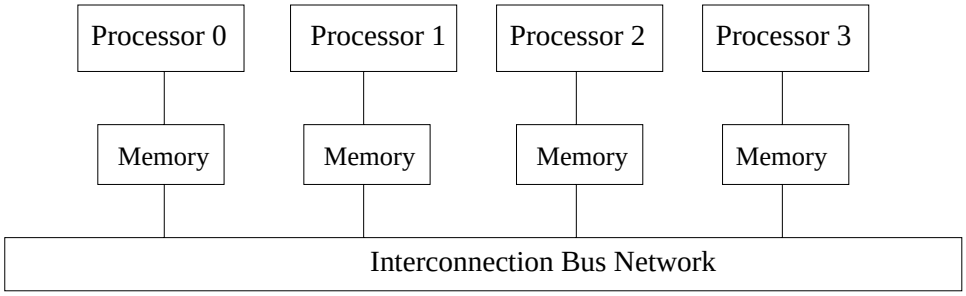


Figure 2.7: Bus network with 4 Processor

Signal processing

Here we briefly describe a Feature Extraction problem [4] in which the arbitrary divisibility property of the image data is exploited to processing. Consider an image in the form of a cluster of pixel that may be a subset of the original image array. The primary task of image feature extraction is to process this data to generate a representation that facilitates higher level symbolic manipulations. It is possible to exploit data parallelism at this stage of processing by assigning different portions of the image array to each of the processors in a parallel or distributed processing system.

To illustrate the above point, consider the Hough transform of the straight lines in an image. It is given as an array $B(\rho, \theta)$, each element of which represents the number of pixels whose spatial coordinates (x, y) in the given image array satisfy the equation

$$\rho = x.cos\theta + y.sin\theta \tag{2.5}$$

For each pair (x, y), the value of ρ is computed for a set of discrete values of θ. Thus, each point in the (x, y) plane generates a curve in the (ρ, θ) plane. Based on the nature of these curves and their relative position one can identify $B(\rho, \theta)$ and obtain information about the features in the given image array. Note that the computation of Hough transform for each point is done independent of any of the other point. This aspect makes the data (image array) arbitrarily divisible.

As an illustrate example, let us assume that the data to be processed in the above manner is stored in a (512×512) image array. Let the computation done on a single pixel take 1 unit of time in any of the processors in a network consisting of 4 identical processors p_0, p_1, p_2, p_3 connected through a bus and having separate local memories (Figure 2.7) The data to be processed is resident in p_0 which can communicates segments of the data, one at a time, to the other processors. If the communication delay in sending data is negligible then it is wise to distribute the data in 4 equal parts.

For example, each processor can be assigned 128 rows (Figure 2.8(a)), thus incurring a processing time of $(1 \times 128 \times 512)$ time units. However, when the communication delay

is not negligible, as when the processors are well separated, then this strategy is no longer optimal. Suppose the time delay for communicating one pixel from one processor to another is 10 percent of the computation time per pixel. Then the times taken by each processor to complete its computation is

- $(1 \times 128 \times 512)$ time units for p_0
- $(1.1 \times 128 \times 512)$ time units for p_1
- $(1.2 \times 128 \times 512)$ time units for p_2
- $(1.3 \times 128 \times 512)$ time units for p_3

Thus, the processing of the complete data is over only after processor p_3 complete its computation. Hence, the presence of communication delay has increased the processing time by 30 percent. But it is obvious that we can exploit the arbitrary divisibility property of the data to improve performance.

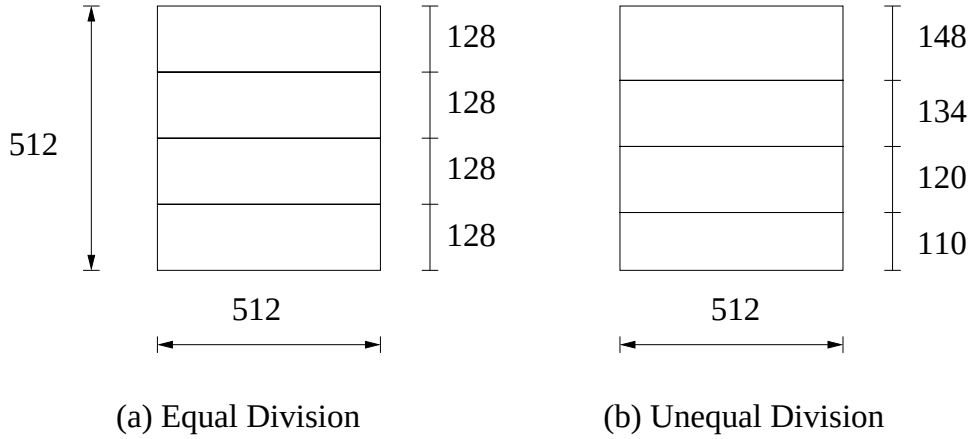


Figure 2.8: Partitioning of data for Image Feature Extraction

For example, let us allocate 147 rows to p_0, 134 rows to p_1, 121 rows to p_2, and 110 rows to p_3 (Figure 2.8(b). The the processing time for each processor is

- $(1 \times 147 \times 512)$ time units for p_0
- $(1.1 \times 134 \times 512)$ time units for p_1
- $(0.1 \times 134 \times 512)$ time units for p_2
- $(0.1 \times 134 \times 512 + 0.1 \times 121 \times 512 + 1.1 \times 110 \times 512)$ time units for p_3

From the above, we find that p_1 takes the maximum time to complete its computation. For comparison, we can rewrite this time as $(1.152 \times 128 \times 512)$ time units and note that this strategy has produced a 15 percent reduction over the naive equal division strategy.

The above example demonstrates how the arbitrary divisibility property of the data (image and signal data) can be exploited to enhance the performance of a real-world image feature extraction algorithm. However, note that since we have allocated data in terms of rows, the data is not arbitrarily divisible in the true sense, but may be considered to be so for large volumes of data.

2.5.4 Divisible Load System

The basic idea underlying the process of scheduling divisible loads to minimize the processing time in distributed networks is in devising efficient load distribution strategies. While a data partitioning algorithm is simple to implement, the non triviality of scheduling divisible loads lies in designing strategies that efficiently utilize the available network resources in terms of computational power and communication channel bandwidth.

The load distribution model

Divisible load distribution, in general, goes through the following process. The load to be processed arrives at a specified node, called the *root* node (in the case of tree networks) or the *master* (in the case of star networks), depending upon the architecture under consideration. Also, the architecture can be such that the processors can be equipped *with front-ends* or *without front-ends*. In front-end case, with a network involving m processors, the originator partitions the load into m fractions, starts the computation on its own load fraction and simultaneously starts distributing the other load fractions to other processors one at a time in a predetermined order. Note that the computation and communication events occur concurrently at the originator, if it is equipped with a front-end (also known as a *communication co-processor*). On the other hand, in the without front-end case, the originator first distributes the load fractions to the rest of the processors and then it computes its own load fraction.

Obviously, when we consider a linear topology for the network, the originator pumps all the data in a pipelined fashion, and every processor that receives the data from its predecessor keeps the portion intended for it and passes the rest to its successor. The problem is then to choose the size of these load fractions in such a way that our objective of minimum processing time is met. It is important to note that we are addressing the problem of load partitioning in a heterogeneous system of processors and links, and hence, dividing the load into equal sized fractions will naturally result in a poor performance.

Notations

Below we describe the standard notations used in this section:

- $\alpha = (\alpha_1, ..., \alpha_m)$: load distribution vector;
- α_i : load fraction allocated to processor p_i ;
- $T(\alpha)$: finish time of load distribution α;
- S_i : ratio of the time taken by processor p_i , to compute a given load, to the time taken by a standard processor, to compute the same load;
- T_{cp}: time taken to process a unit load by the standard processor;
- B_i : ratio of the time taken by link L_i , to communicate a given load, to the time taken by a standard link, to communicate the same load;
- T_{cm}: time taken to communicate a unit load on a standard link

Then $\alpha_i T_{cp}/S_i$ is the time to process the fraction α_i of the entire load on the ith processor. Note that the units of $\alpha_i T_{cp}/S_i$ are

$$[\text{load}] \times [\text{sec/load}] \times [\text{dimensionless quantity}] = [\text{seconds}]$$

Likewise, $\alpha_i T_{cm}/B_i$ is the time to transmit the fraction α_i of the entire load over the ith link. Note that the units of $\alpha_i T_{cm}/B_i$ are

$$[\text{load}] \times [\text{sec/load}] \times [\text{dimensionless quantity}] = [\text{seconds}]$$

The standard processor or link referred to above is any processor or link which is used as a reference. It could be any processor or link in the network or a conveniently defined virtual processor or link.

Linear networks

The first target architecture to be examined in the study of divisible loads was a linear daisy chain. This was done based on a perception that such a reduced case might be tractable. The original application was for sensor networks where the "sensors" were networked computers that shared information. In a linear network the processor p_0 (the root) is connected to processor p_1 via link L_1, p_1 is connected to p_2 via L_2 and so on until p_{m-1} is connected to the boundary processor p_m via L_m. If a divisible load originates at one end of a daisy chain of m processors then a set of m linear equations can be set up to solve for the optimal fraction of load to be assigned to each processor in order to minimize the "finish time". Here finish time is the time when all processing has stopped. Other variations to this problem deal with a load that originates at an processor and also when the time for processors to report solutions back to the originator is non negligible.

This optimal assignment of load is done in the context of the schedule of load distribution that the equations are based on. One can have load distribution strategies that involve round robin distribution of load to the processors or strategies that simply distribute load to each processor in turn once. One can also distinguish between a store and forward reception of load and those strategies that are more akin to the virtual cut through switching strategy of networking.

Finally, linear daisy chains that are infinite in extent can be solved to obtain performance bounds. Actually finish time tends to saturate as more processors are added to the network because of the repetitive overhead in communicating load down a chain. Using concepts from algebra, combinatorics, and even electric circuit theory, both the optimal load allocations and finish time of infinite sized daisy chains can be obtained. Such infinite networks represent a performance benchmark that finite chains can be compared against. However, it soon became apparent that other topologies such as tree and buses would yield superior performance.

As an illustration, we consider a linear network with three processors. The equations, however, are easily generalizable to m processors. Each processor is equipped with a front-end. The timing diagram, given in Figure 2.9, shows communication delay above the time axis and computation time below. The finish times for all processors are assumed to be equal for optimal load distribution

$$\alpha_0 + \alpha_1 + \alpha_2 = 1,$$
$$\frac{\alpha_i T_{cp}}{S_i} = (1 - \alpha_0 - .. - \alpha_i)\frac{T_{cm}}{B_{i+1}} + \frac{\alpha_{i+1} T_{cp}}{S_{i+1}}, \quad i = 0, 1.$$

These linear equations can be reduced to a series of product forms that are easy to compute recursively and whose elegance aids analysis. Closed form solutions are available when the network is homogeneous (that is, equal link speeds and processor speeds).

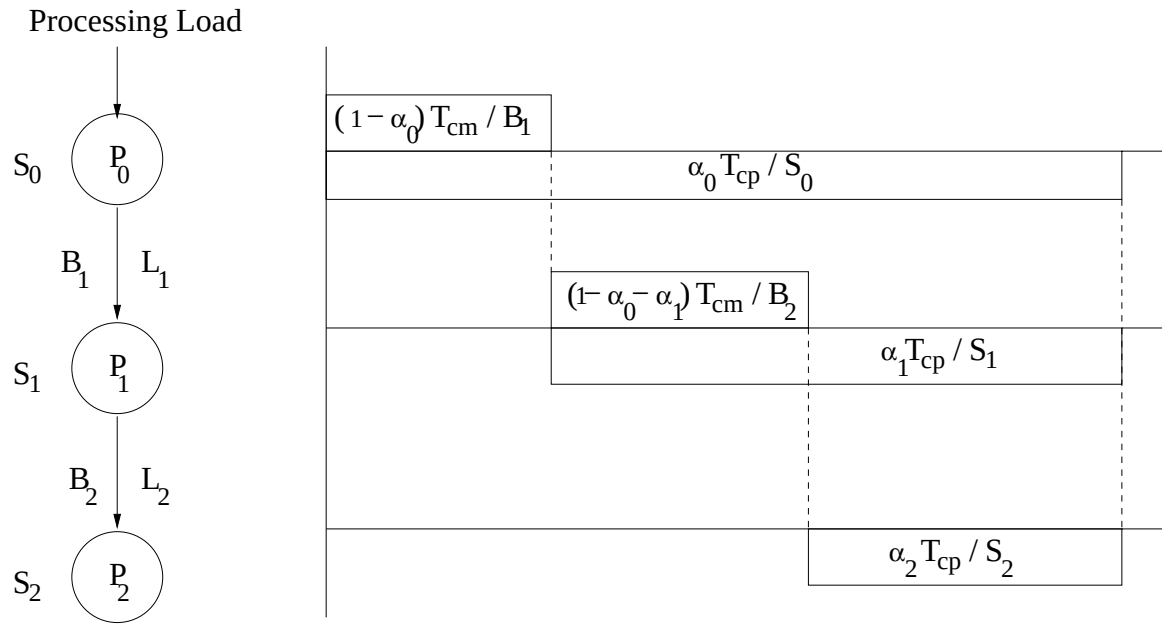


Figure 2.9: Load distribution in a three-processors linear network

Tree and bus networks

The researches in the early stages of divisible load theory considered this type of network. In these studies, a bus network architecture was conceived as a special case of single-level homogeneous tree networks. This is, of course, expected as our modelling ensures that when all the link speeds are identical in single-level tree network (SLTN) or in star network architectures, the resultant network becomes identical to a bus network. Thus, all the results that are valid for SLTN also hold for bus networks. The treatment to obtain optimal finish time solution follows the same technique as the linear network.

There is a possibility in tree networks of varying the order or sequence of load distribution among the child processors. There are $m!$ such sequences possible with m processors connected to the root. An optimal sequence is that in which the load distribution follows the order in which the link speeds decrease. However, from a network designers perspective, if architectural rearrangement is permissible, then the best way one can arrange the processors and links would be to connect the fastest processor to the fastest link and the next fastest processor to the next fastest link and so on and then follow the optimal sequence of load distribution. Further, when front-ends exist the root must be the fastest of all the processors in the network. The extension of many of these results to multi-level tree networks is also available.

In bus networks, three possible configurations are of interest:

- bus equipped with a control processor or a *bus controller unit* (BCU),
- bus not equipped with a BCU, but processors with front-ends, and

- bus not equipped with a BCU, and processors without front-ends.

While the analytical treatment and the load distribution process remains identical, the study of divisible load scheduling on bus networks is of interest since the network is simple in nature and allows one to design and study the performance of complex scheduling strategies. As an illustration we consider a tree network with one root processor and two

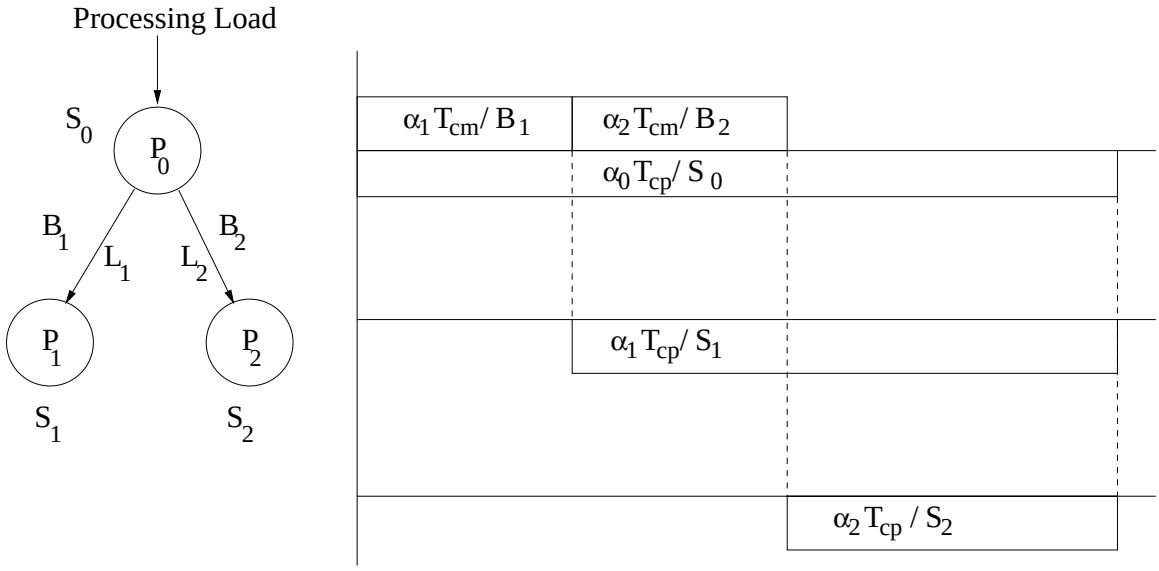


Figure 2.10: Load distribution in a three-processors tree network

child processors in Figure 2.10. The corresponding load distribution equations are,

$$\alpha_0 + \alpha_1 + \alpha_2 = 1,$$
$$\frac{\alpha_i T_{cp}}{S_i} = \frac{\alpha_{i+1} T_{cm}}{B_{i+1}} + \frac{\alpha_{i+1} T_{cp}}{S_{i+1}}, \quad i = 0, 1.$$

These equations too can be solved recursively. Closed-form solutions are also possible.

As a numerical example, consider a single-level heterogeneous tree with 4 processors where

$$S_0 = 1/2, S_1 = 1/3, S_2 = 1, S_3 = 1/2, B_1 = 1/2, B_2 = 2, B_3 = 1/5, T_{cm} = T_{cp} = 1.$$

If we use all the processors then $\alpha = 0.4321, 0.1728, 0.3457, 0.0494$ and $T(\alpha) = 0.8642$. But this distribution is not optimal. Suppose p_2 is given $\alpha_1 + \alpha_2$ and p_1 is not given any load, then the new load distribution is $\alpha' = 0.4321, 0, 0.5185, 0.0494$ and the processing time is $T(\alpha') = 0.8642$. In this case, the processors do not stop at the same time. If we redistribute load to the reduced network consisting of p_0, p_2, p_3 only so that they all have the same finish times then the new load distribution is

$$\alpha'' = 0.3962, 0, 0.5283, 0.0755; T(\alpha'') = 0.7924$$

which is indeed the optimal finish time solution to this problem.

The optimality principle

In the above discussions we assumed that to obtain optimal processing time all the participating processors must stop computing at the same instant in time. This was the basic optimality principle in the case of divisible load scheduling problems. This assertion is supported by an intuitive observation that when all processors do not stop computing at the same time, it is possible to redistribute some load from processors that stop computing later to those that stop computing earlier. While the above claim seems to have an intuitive validity, it was subsequently shown that, for a single level tree network, the optimal processing time can be achieved by distributing the load only among the "fast" processor-link pairs.

An exact expression that distinguishes the "fast" processor-link pairs from the "slow" processor-link pairs has been derived. A reduced network can then be obtained after eliminating the slow processor-link pairs and the load is distributed among the remaining processors using the optimality principle.

Based on the above, it is reasonable to say that although the optimality principle remains valid for even an arbitrary network topology, the optimal time performance depends crucially on the selection of a proper subset of the available processors. Thus, using a larger set of nodes may yield an inferior performance compared to an optimal subset of nodes among which the load is distributed according to the optimality principle. In the case of homogeneous single-level tree networks (and also for the bus networks) all the processor-link pairs are identical and hence all of them must be used to process the load.

Multi-installment strategy

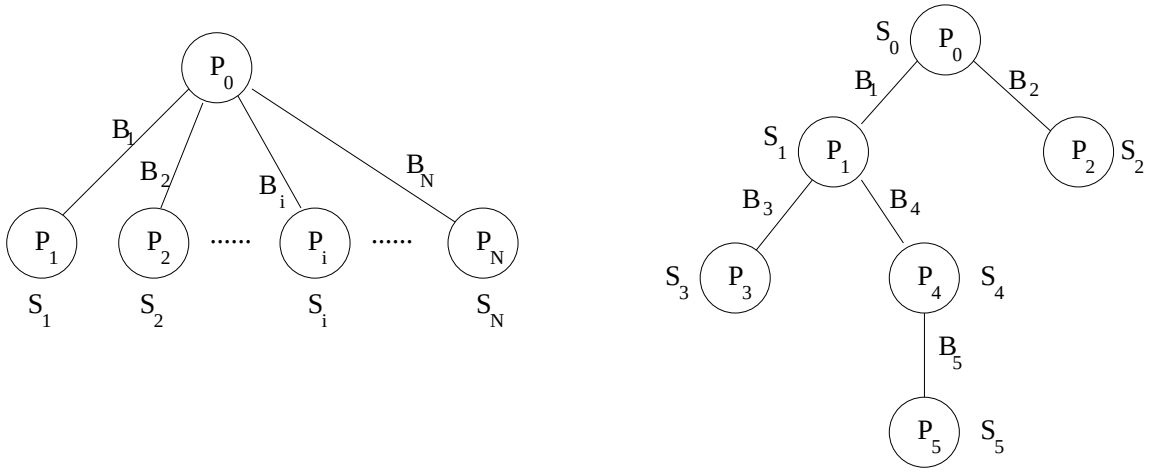
The load distribution model discussed above underwent a fair bit of revision when it incorporated pipelining in the form of a multi-installment strategy, where a processor need not wait till the complete load fraction to its predecessor has been transferred. Exploiting the divisible nature of the load, each fraction was further subdivided and distributed in a repetitive sequence. This strategy reduced the idle time of the processors at the farthest end of the load distribution sequence. In addition to a resulting reduction in processing time, one can also have a control on the finish time by selecting the number of installments. This capability is crucial for real-time processing of certain types of loads.

2.6 Divisible Load Scheduling Problem

2.6.1 Framework

We limit our study to *star-shaped* logical network topologies (Figure 2.11(a)), because it often represent the solution of choice to implement master-worker computations. We refer the reader to [4, 15] for more details regarding to other topologies such as *tree* (Figure 2.11(b)).

Note that the star network encompasses the case of a bus, which is a homogeneous star network. In this section we consider two types of model for communication and computation: *linear* or *affine* [15, 16] in the data size. In most contexts, this is more accurate than the fixed cost model, which assumes that the time to communicate a message is independent of the message size.



(a) Heterogeneous star topology, linear cost model

(b) Tree topology, linear cost model

Figure 2.11: Star and Tree network

As illustrated in Figure 2.11(a), a star network $G = (P_0, P_1, P_2, ..., P_N)$ is composed of a *master*P_0 and N *workers*: $(P_0, P_1, P_2, ..., P_N)$. There is a communication link from the master P_0 to each worker P_i.

In the *linear* cost model, each worker P_i has a relative computing power S_i: it takes X/S_i time units to execute X units of load on worker P_i. Similarly, it takes X/B_i time unites to send X units of load from P_0 to P_i. Without loss of generality we assume that the master has no processing capability (otherwise, add a fictitious extra worker paying no communication cost to simulate computation at the master).

In the *affine* cost model, a latency is added to computation and communication costs: it takes $cLat_i + X/S_i$ time units to execute X units of load on worker P_i, and $nLat_i + X/B_i$ time units to send X units of load from P_0 to P_i. It is acknowledged that these latencies make the model more realistic.

For communications, the *one-port* model is used: the master can only communicate with a single worker at a given time-step. We assume that communications can overlap computations on the workers: a worker can compute a load fraction while receiving the data necessary for the execution of the next load fraction. This corresponds to workers equipped with a *front end* as mentioned above. A *bus* network is a star network such that all communication links have the same characteristics: $B_i = B$ and $nLat_i = nLat$ for each worker P_i, $(1 \leq i \leq N)$.

2.6.2 The Complexity

Consider the Divisible Load Scheduling (DLS) problem and its complexity, with a particular emphasis on *master-worker* platforms with a *star* topology. We have an application that consists of an amount of load, which can be arbitrarily divided into any number of *chunks*, where each chunk consists of some amount of input data and some computation to perform on this data. The objective of the DLS problem is to assign load chunks to workers, which are accessible from a master over a network, so that the *makespan*, that is the overall application execution time, is minimized. The entire load initially resides at

the master.

StarLinear Problem

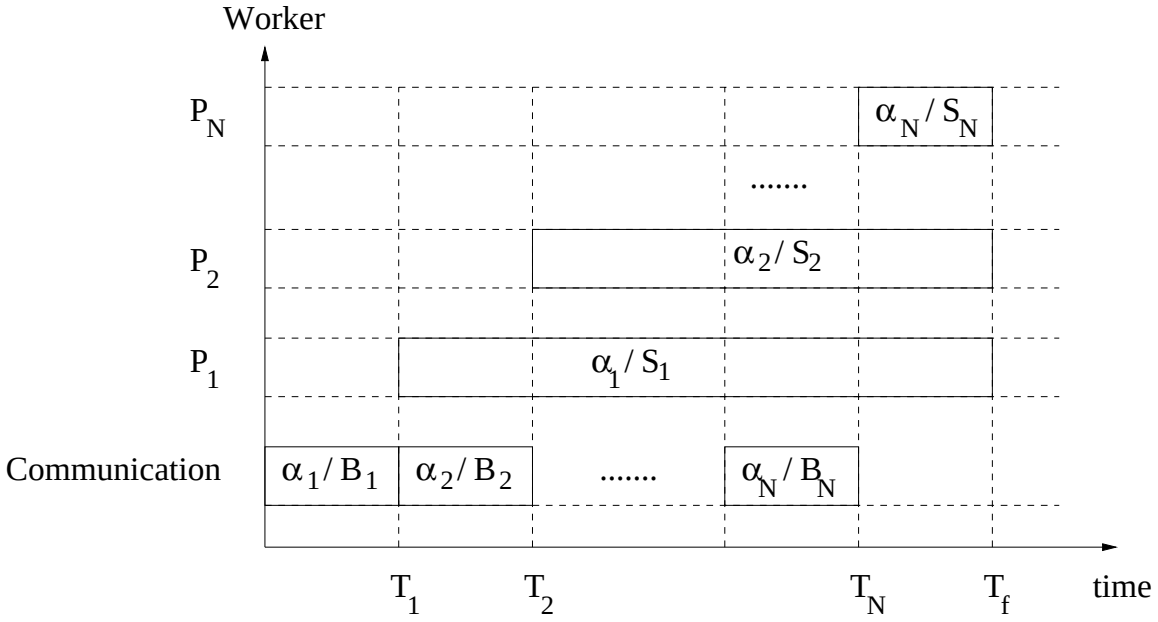


Figure 2.12: Single-round scheduling with star network and linear cost model

This is the simplest platform combination, denoted as StarLinear. Let α_i denote the number of units of load sent to worker P_i, such that $\sum_{i=1}^{N} \alpha_i = L_{total}$. Figure 2.12 depicts the execution, where T_i denotes idle time of P_i, i.e. the time elapsed before P_i begins its processing. The goal is to minimize the total execution time, $P_f = max_{1 \leq i \leq N}(T_i + \frac{\alpha_i}{S_i})$, according to the linear model defined in the above section. In Figure 2.12, all the workers participate in the computation, and they all finish computing at the same time (i.e. $T_i + \frac{\alpha_i}{S_i} = T_f, \ \ \forall i$). We outline here some general results:

Theorem 1 *[15] In any optimal solution of the StarLinear problem, all workers participate in the computation, and they all finish computing simultaneously.*

Theorem 2 *[15] An optimal ordering for the StarLinear problem is obtained by serving the workers in the ordering of decreasing link capacities G_i.*

Theorem 3 *[15] The optimal solution for the StarLinear problem is given by the solution of the follow linear program.*

Minimize T_f subject to:

$$\begin{cases} (1) & \alpha_i \geq 0 & 1 \leq i \leq N \\ (2) & \sum_{i=1}^{N} \alpha_i = L_{total} & \\ (3) & \alpha_1/(B_1 + S_1) \leq T_f & \\ (4) & \sum_{j=1}^{i} \frac{\alpha_j}{B_j} \leq T_f & 2 \leq i \leq N \end{cases}$$

The more details of these theorems were presented in [15]. We note that inequalities (3) and (4) will be in fact equalities in the solution of the linear program, so that we can

easily derive a closed-form expression for T_f. We point out that this is linear programming with rational numbers, hence of polynomial complexity. Similarly, it is also known that if the costs model is affine in the chunk size and the platform is homogeneous, then the problem is polynomial-time [4].

Finally, in the case of affine costs and of a heterogeneous platform, per [25], the Divisible Load Scheduling problem (DLS) is defined as follow :

Problem 4 *Consider a set of N worker processors and a master processor. The master must send out L_{total} units of load to workers to be computed. Each worker P_i can compute S_i, $i \in [1, N]$ load units per second, and each transfer to a worker P_i takes a fixed amount of time, $nLat_i$ seconds. Is it possible to compute all L_{total} load units within T_0 seconds after the master starts sending out the first load unit? ($T_0, L_{total}, nLat_i, S_i$ are all rational numbers, due to the divisible nature of the load.)*

In summary, A. Legrand [25] proves that DLS problem for a heterogeneous star network with affine costs is a NP-complete problem. This justifies the use of heuristics such as the ones employed in [16, 17]. It is important to note that the proof not only works for one-round schedules, but also for multi-round schedules.

Chapter 3

Static Algorithm for Divisible Load Scheduling Problem

3.1 Preliminary

Numerous studies in the literature have been targeting the problem of scheduling divisible workloads (those loads that are amenable to partitioning in any number of chunks). However, such algorithms have a number of shortcomings such as the sole reliance in their computations on CPU speed, and the assumption that a definite set of workers are available and must participate in processing the load. These constraints limit the utility of such algorithms and make them impractical for a Non-Dedicated computing platform such as the Grid. In this chapter, we propose an algorithm, MRRS [40, 41], that overcomes these limitations and adopts a worker selection policy that aims at minimizing the execution time. The MRRS has been evaluated against other scheduling algorithms such as UMR and LP and showed better results.

Per the Divisible Load Theory [1], the scheduling problem is identified as: "Given an arbitrary divisible workload, in what proportion should the workload be partitioned and distributed among the workers so that the entire workload is processed in the shortest possible time?"

Any scheduling algorithm should address the following issues:

- Workload Partitioning Problem. This problem is concerned with the method by which the algorithm should divide the workload in order to dispatch to workers.
- Resource Selection Problem. This problem is concerned with how to select the best set of workers that can process the workload partitions such that the makespan is minimal.

Single round algorithms [4], [5] are an early and simple way for scheduling. As shown in [4], for a large workload, the single-round approach is not efficient due to the large idle time attributed to the last worker as it waits for receiving its workload chunk.

Multi-round algorithms, first introduced by Bharadwaj [1], further utilize the overlapping between communication and computation processes at workers. The master delivers chunks of loads to workers one after another, over multiple rounds, until all load partitions are delivered.

In the first multi round algorithm, called MI [1], the number of rounds is fixed and predefined. It overlooks communication and computation latencies. Beaumont [4] proposes a multi round scheduling algorithm that fixes the execution time for each round. This enabled the author to give analytical proof of the algorithm's asymptotic optimality. But the influence of this assumption on the utilization of transfer-execution overlap is questionable. Yang et al. [2], [6], through their UMR (Uniform Multi-Round) algorithm, designed a better algorithm that extends the MI by considering latencies. However, in the UMR, the size of workload chunks delivered to workers is solely calculated based on worker's CPU power; the other key system parameters, such as network bandwidth, are not factored in.

One apparent shortcoming in the many scheduling algorithms [1], [2], [4], [5], [7] that exist in the literature is the abandon of designing a solid selection policy for generating the best subset of available workers. The first studies [1], [5] did not mention about this topic. Part of the reason is that the main focus of these algorithms is confined to the LAN environment, which makes them not perfectly suitable for a WAN environment such as the Grid [3]. In the Grid, computation resources (workers) join and leave the computing platform dynamically. Unlike other algorithms, we cannot assume in the Grid that all available resources, which may be in thousands, must participate in the scheduling process. The more recent algorithms discussed in [2], [6] very tersely allude to this problem by proposing primitive intuitive solutions that are not back up by any analytical model.

In this section, we propose a new scheduling algorithm, MRRS [40, 41](inspired by UMR [2], [6]), which is better and more realistic. MRRS is superior to UMR with respect to two aspects. First, unlike UMR that relies primarily in its computation on the CPU speed, MRRS factors in several other parameters, such as bandwidth capacity and all types of latencies (computation and communication) which renders the MRRS a more realistic model. Second, the MRRS is equipped with a worker selection policy that finds out the best workers. As a result, our experiments show that our MRRS algorithm outperforms previously proposed algorithm including the UMR.

3.2 The Heterogeneous Computing Platform

Let us consider a computation Grid in which a master process has access to N worker processes and each process runs in a particular computer. The master can divide the total load L_{total} into arbitrary chunks and delivers them to appropriate workers. We assume that the master uses its network connection in a sequential fashion, i.e., it does not send chunks to some workers simultaneously. Workers can receive data from network and perform computation simultaneously.

Let us formalize our model by listing the notations which will be used throughout this dissertation:

- W_i: worker number i.
- N: total number of available workers.
- n: total number of workers that are actually selected to process the workload.
- m: the number of rounds.

- $chunk_{j,i}$: the fraction of total workload that the master delivers to worker W_i in round j $(i = 1, .., n\ ;\ j = 1, .., m)$

- S_i: computation speed of the worker i measured by the number of units of workload performed per second $(flop/s)$

- B_i: the data transfer rate of the connection link between the master and worker W_i $(flop/s)$

- $Tcomp_{j,i}$: we model the time required for worker i to perform the computation $chunk_{j,i}$ as:

$$Tcomp_{j,i} = cLat_i + \frac{chunk_{j,i}}{S_i}$$

- $cLat_i$: the fixed overhead time, in seconds, for starting a computation (e.g. for starting a remote process) in the worker W_i. The computation, including the $cLat_i$ overhead, can be overlapped with communication.

- $nLat_i$: the overhead time, in seconds, incurred by the master to initiate a data transfer to W_i (e.g. pre-process application input data and/or initiate a TCP). We denote total latencies by $Lat_i = cLat_i + nLat_i$.

- $Tcomm_{j,i}$: we model the communication time spent by the master to send $chunk_{j,i}$ units of data to worker W_i as:

$$Tcomm_{j,i} = nLat_i + \frac{chunk_{j,i}}{B_i}$$

- $round_j$: the fraction of workload dispatched during round j

$$round_j = \sum_{i=1}^{n} chunk_{j,i}$$

We fix the time required for each worker W_i to perform communication and computation during each round j

$$cLat_i + \frac{chunk_{j,i}}{S_i} + nLat_i + \frac{chunk_{j,i}}{B_i} = const_j \tag{3.1}$$

We set

$$A_i = \frac{B_i S_i}{B_i + S_i}$$

so we have

$$chunk_{j,i} = \alpha_i round_j + \beta_i \tag{3.2}$$

where

$$\alpha_i = \frac{A_i}{\sum_{k=1}^{n} A_k};\quad \beta_i = A_i \frac{\sum_{k=1}^{n} A_k (Lat_k - Lat_i)}{\sum_{k=1}^{n} A_k} \tag{3.3}$$

This model is flexible enough that it can be instantiated to model several types of network connections. For instance, setting the $nLat_i$ values to 0 models a pipelined network such as the one used in [26]. The model can also be instantiated with non-zero $nLat_i$ values as in [16]. This is representative of distinct connections being established for each individual transfer, with no pipelining. Zero $nLat_i$ corresponds to the work in [4, 5]. We provide an analysis of our scheduling algorithm using this generic platform model, and thereby validate our approach for various platforms.

3.3 Previous work: the UMR algorithm

In the first multi round algorithm MI [4], the number of rounds is fixed and predefined. MI does not consider communication and computation latencies. Yang et al. extend the MI by considering latencies which better models the real situations. Their UMR (Uniform Multi Round) algorithm [17] assumes that the total of transfer time and execution time of every worker are equal within each round. This approach enables them to analyze the constraints and find the near-optimal number of rounds as well as the size of chunks in each round. Among the existing scheduling algorithms for divisible load, UMR is the only algorithm that obtains the approximately optimal number of rounds and the size of load chunks. We choose algorithm UMR as the backbone to develop our static scheduling algorithm MRRS. In this section, we sketch the key concepts of the UMR algorithm, referring the reader to [17, 18] for more details. Next, we explain how the UMR partitions its load into chunks, how it decides on the size of each chunk, what are the initial parameters for the first round, and outline the simple worker selection policy adopted by the UMR.

Load Partitioning Policy

In some theoretical models of computing platform such as [18], the authors propose a parameter called $tLat$ that defined as "the time interval between when the master finished pushing data on the network to worker i and the time when worker i receives the last byte of data". The $tLat$ is overlappable as show in Figure (3.1) and can be used to represent the network latency between the master and the worker. However, because of $tLat$ is overlappable and the effect of $tLat$ is just to shift the running time by $tLat$ (second) therefore, like [17, 16], we ignore this parameter in our derivation.

UMR adopts a load partition policy that ensures that each worker spends the equal CPU time like others through a round, network bandwidth is not taken into account. We can express this policy as follows:

$$cLat_i + \frac{chunk_{j,i}}{S_i} = const_j \tag{3.4}$$

so we derive

$$chunk_{j,i} = \frac{S_i}{\sum_{k=1}^{n} round_j} + \delta_i \tag{3.5}$$

where

$$\delta_i = \frac{S_i}{\sum_{k=1}^{n} S_k} \sum_{k=1}^{n} (S_k cLat_k) - S_i cLat_i \tag{3.6}$$

It can be seen that, the bandwidth B_i does not appear in above expressions as it is not considered by the UMR algorithm. Lemma (8) in the next section will show that, by considering both of CPU power, S_i, and bandwidth, B_i, MRRS shows better performance than UMR.

Conditions for full platform utilization

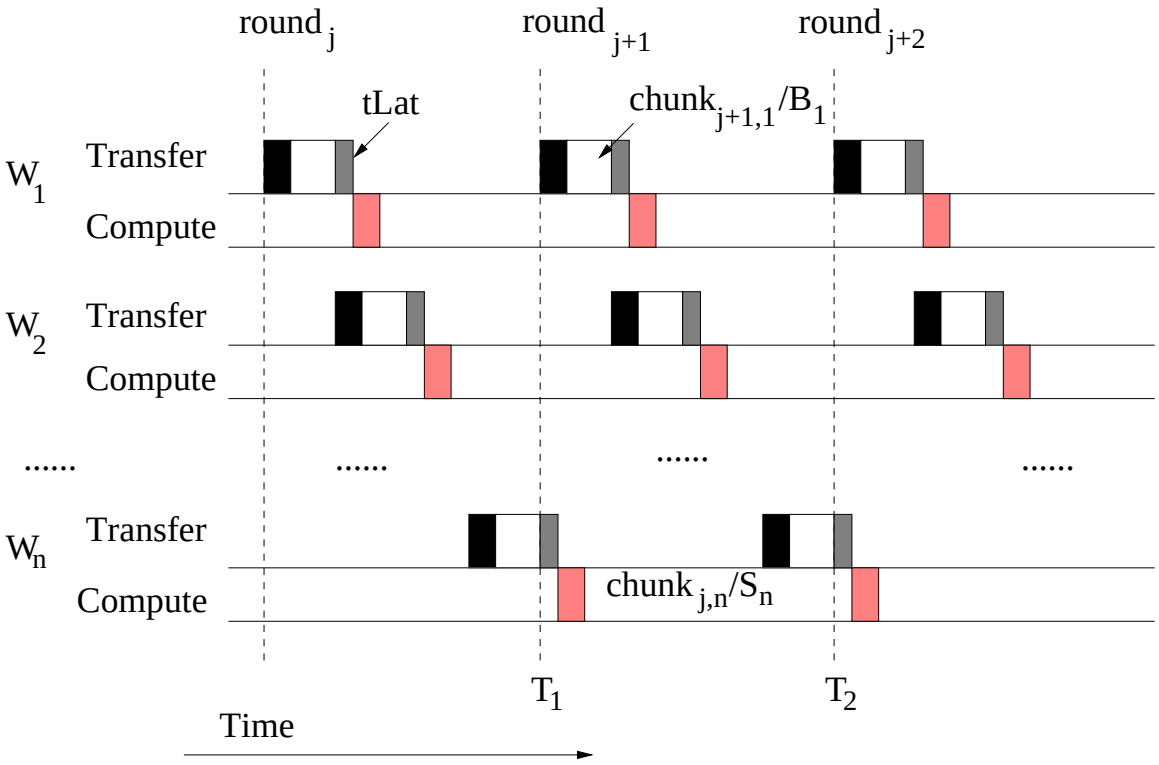


Figure 3.1: The load dispatching in UMR algorithm

Figure (3.1) depicts the operation of UMR algorithm. At time T_1, the master starts sending $chunk_{j+1,i}$ (i=1, 2,..., n) to all workers and the last worker W_n starts computation $chunk_{j,n}$ concurrently. To fully utilize the network bandwidth, the dispatching of the master and the computation of W_n should finish at the same time T_2. The induction relation on round j is:

$$round_j = \phi^j(round_0 - \mu) + \mu \tag{3.7}$$

where

$$\phi = \left(\sum_{i=1}^{n} \frac{S_i}{B_i}\right)^{-1} \tag{3.8}$$

$$\mu = \frac{\sum_{i=1}^{n}(S_i cLat_i) - \sum_{i=1}^{n} S_i \times \sum_{i=1}^{n}\left(\frac{\beta_i}{B_i} + nLat_i\right)}{\sum_{i=1}^{n} \frac{S_i}{B_i} - 1} \tag{3.9}$$

Constrained minimization problem

To start the scheduling process, $round_0$ and the number of rounds m should be determined first. Since the objective of the UMR is to minimize the makespan of the application, we

can write:

$$H(m, round_0) = \sum_{i=1}^{n} \left(\frac{chunk_{0,i}}{B_i} + nLat_i \right) + \sum_{j=0}^{m-1} \left(\frac{chunk_{j,n}}{S_n} + cLat_n \right) \tag{3.10}$$

At the same time, we also have the constraint that the chunk sizes sum up to the total workload:

$$K(m, round_0) = m\mu + (round_0 - \mu)\frac{1-\phi^m}{1-\phi} - L_{total} = 0$$

This optimization problem can be solved by the Lagrangian method [27]. After obtaining m and $round_0$, using equations (3.7) and (3.5) we can obtain the value of $round_j$ and $chunk_{j,i}$ respectively ($i = 1, 2, ..., n$; $j = 1, 2, ..., m$).

Worker Selection Policy of the UMR

The worker selection problem implies selecting the best n workers out of N workers that are available and accessible in such a way the makespan is minimal. Inspired by the work in [32] and based on the intuition, Y.Yang employs a simple method that sorts workers S_i/B_i in increasing order, and selects the first n workers out of the original N workers such that:

$$\sum_{i=1}^{n} \frac{S_i}{B_i} < 1 \tag{3.11}$$

Furthermore, UMR requires that, the computation-communication ratio B_i/S_i be larger than the number of workers n:

$$\frac{B_i}{S_i} \geq 1 \quad (\forall i = 1, 2, ..., N) \tag{3.12}$$

3.4 Multi-round Scheduling with Resource Selection (MRRS) Algorithm

In this section, we build a new scheduling algorithm, **MRRS** (*Multi-Round scheduling with Resource Selection*) [40, 41], which is inspired by the UMR algorithm. MRRS is superior to UMR with respect to two aspects. First, unlike the UMR, which relies primarily in its computation on the CPU speed, MRRS factors in several other parameters such as bandwidth capacity and all types of latencies (CPU and Bandwidth) which renders the MRRS a more realistic model. Second, the UMR assumes that all workers are available and should participate in the workload processing, which is not practical especially in a computing environment such as the Grid. The MRRS, on the other hand, is equipped with a worker selection policy that works on selecting the best workers that can produce shorter makespan. As a result, our experiments show that our MRRS algorithm outperforms previously proposed algorithm including the UMR.

3.4.1 Induction Relation for Chunk Sizes

Figure (3.2) depicts the operation of our algorithm, where the computation and communication have been overlapped. At time T_1, the master starts sending round $(j+1)$ to all

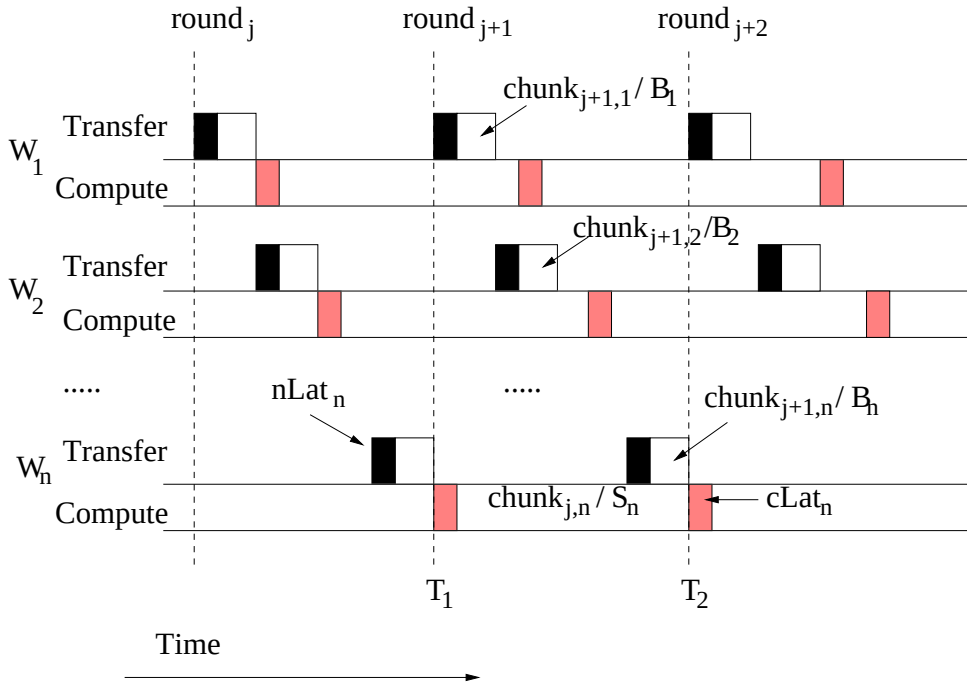


Figure 3.2: Dispatching load chunks using the MRRS algorithm

workers and the last worker W_n starts computation chunk j concurrently. To fully utilize the network bandwidth, the dispatching of the master and the computation of W_n should finish at the same time T_2:

$$\sum_{i=1}^{n}\left(nLat_i + \frac{chunk_{j+1,i}}{B_i}\right) = \frac{chunk_{j,n}}{S_n} + cLat_n$$

If we replace $chunk_{j+1,i}$ and $chunk_{j,n}$ by their expression in (3.1) we derive:

$$round_{j+1} = round_j\theta + \mu \tag{3.13}$$

where

$$\theta = \frac{B_n}{(B_n+S_n)\sum_{i=1}^{n}\frac{S_i}{B_i+S_i}};\quad \mu = \frac{\frac{\beta_n}{S_n} + cLat_n - \sum_{i=1}^{n}\left(nLat_i + \frac{\beta_i}{B_i}\right)}{\sum_{i=1}^{n}\frac{\alpha_i}{B_i}} \tag{3.14}$$

From induction equation (3.13) we can compute:

$$round_j = \theta^j(round_0 - \eta) + \eta \tag{3.15}$$

where

$$\eta = \frac{\beta_n + cLat_n - \sum_{i=1}^{n}\left(nLat_i + \frac{\beta_i}{B_i}\right)}{\sum_{i=1}^{n}\frac{\alpha_i}{B_i} - \frac{\alpha_n}{S_n}}$$

3.4.2 Determining the Parameters of the Initial Round

In this section we compute the optimal number of rounds, m, and the size of the initial load fragment that should be distributed to workers in the first round, $round_0$. If we let

$F(m, round_0)$ denote the makespan, then from Figure (3.2) we can seen that

$$F(m, round_0) = \sum_{i=1}^{n} \left(\frac{chunk_{0,i}}{B_i} + nLat_i \right) + \sum_{j=0}^{m-1} \left(\frac{chunk_{j,n}}{S_n} + cLat_n \right) = \tag{3.16}$$
$$= round_0 \left(\sum_{i=1}^{n} \frac{\alpha_i}{B_i} + \frac{\alpha_n(1-\theta^m)}{S_n(1-\theta)} \right) + \sum_{i=1}^{n} \left(\frac{\beta_j}{B_i} + nLat_i \right) +$$
$$+m \left(cLat_n + \frac{\alpha_n \eta + \beta_n}{S_n} \right) - \frac{\alpha_n \eta (1-\eta^m)}{1-\eta}$$

Our objective is to minimize the makespan $F(m, round_0)$, subject to:

$$\sum_{j=0}^{m-1} round_j = L_{total}$$

or

$$G(m, round_0) = m\eta + (round_0 - \eta)\frac{1-\theta^m}{1-\theta} - L_{total} = 0 \tag{3.17}$$

We use Lagrangian method [27] to solve this constrained minimization problem. The minimum value of function $F(m, round_0)$ can be found by solving the following equation system:

$$\frac{\partial L}{\partial \lambda} = G(m, round_0) = 0$$
$$\frac{\partial L}{\partial m} = \frac{\partial F}{\partial M} + \lambda \frac{\partial G}{\partial M} = 0$$
$$\frac{\partial L}{\partial round_0} = \frac{\partial F}{\partial round_0} + \lambda \frac{\partial G}{\partial round_0} = 0$$

where

- λ: Lagrange multiplier
- $L(m, round_0)$: Lagrangian function which is defined as:

$$L(m, round_0, \lambda) = F(m, round_0) + G(m, round_0)$$

After solving this equation system we obtain m. Using equation (3.17) one can then compute $round_0$. At last, using equations (3.15) and (3.2) we will obtain the value of $round_j$ and $chunk_{j,i}$ respectively ($i = 1..n$, $j = 1..m$).

3.4.3 Worker Selection Policy Using Greedy Method

Algorithm 3.4.1: WORKER_SELECTION(V)

main
Search $W_n \in V$ such that:
$B_n/(B_n + S_n) \leq B_i/(B_i + S_i) \quad \forall W_i \in V$
$V_1^* = Horowitz - Sahni; \quad /* \theta > 1 */$
$V_2^* = Greedy(V, \text{“}\theta < 1''); $
$V_3^* = Greedy(V, \text{“}\theta = 1''); $
select $V^* \in \{V_1^*, V_2^*, V_3^*\}$ such that
$m(V^*)$=min$\{m_1(V_1^*), m_2(V_2^*), m_3(V_3^*)\}$;
return(V^*);

Let V denote the original set of N available workers ($|V| = N$). In this section we explain our worker selection policy that aims at finding the best subset $V^*(V^* \subseteq V, |V^*| = n)$ that minimizes the makespan. If W_i denotes worker i, then W_n denotes the last worker receives load chunks in a round, and W_1 denotes the first worker that receives chunks in a round. Algorithm 3.4.1 outlines our selection algorithm. It starts with finding the last worker (W_n) that should receive chunks in a round. V^* is initialized by $\{W_n\}$. Afterward, the selection algorithm, depending on θ , examines three cases using different search algorithms aiming at finding the best algorithm that adds more workers to V^*. After obtaining the three candidate V^* sets, the algorithm chooses the V^* set that produces the minimum makespan. From equation (3.17) we compute the makespan as follows

- if $\theta = 1$:

$$makespan = \frac{L_{total}}{\sum_{i \in V^*} A_i} \left(\frac{1}{m} \sum_{i \in FS} \frac{S_i}{B_i + S_i} + \frac{B_n}{B_n + S_n} \right) + C \tag{3.18}$$

  where C is a constant

$$C = \sum_{i \in V^*} nLat_i + m.cLat_n$$

- if $\theta \neq 1$:

$$makespan = \frac{L_{total}}{\sum_{i \in V^*} A_i} \left(\frac{1 - \theta}{1 - \theta^m} \sum_{i \in FS} \frac{S_i}{B_i + S_i} + \frac{B_n}{B_n + S_n} \right) + C \tag{3.19}$$

Now, since

$$\lim_{m \to \infty} \frac{1 - \theta}{1 - \theta^m} = \begin{cases} 0 & if \quad \theta > 1 \\ 1 - \theta & if \quad \theta < 1 \end{cases}$$

and since m (the number of rounds) is usually large (in our experiments, m is in hundreds), we can write:

$$\frac{1 - \theta}{1 - \theta^m} \approx \begin{cases} 0 & if \quad \theta > 1 \\ 1 - \theta & if \quad \theta < 1 \end{cases}$$

when $\theta > 1$ and by substituting this term into (3.19) we get

$$makespan = L_{total} \frac{B_n}{(B_n + S_n) \sum_{i \in V^*} \frac{B_i S_i}{B_i + S_i}} + C \tag{3.20}$$

when $\theta < 1$ and by substituting the above term into (3.19) we get

$$makespan = L_{total} \frac{\sum_{i \in V^*} \frac{S_i}{B_i + S_i}}{\sum_{i \in V^*} \frac{B_i S_i}{B_i + S_i}} + C \tag{3.21}$$

Based on the above analysis, we have three selection policies for generating V^*:

- Policy I ($\theta > 1$): this policy aims at reducing the total idle time by progressively increasing the load processed in each round (i.e., $round_{j+1} > round_j$).
- Policy II ($\theta < 1$): this policy aims at maximizing the number of workers that can participate by progressively decreasing the load processed in each round (i.e., $round_{j+1} < round_j$).
- Policy III ($\theta = 1$): this policy keeps the load processed in each round constant (i.e., $round_{j+1} = round_j$).

As shown in Algorithm 3.4.1, the three policies will be examined in order to choose the one that produces the minimum makespan. In the coming subsections, we will explain the search algorithm adopted by each policy.

Policy I ($\theta > 1$)

From equation (3.20), we can see that under this policy, V^* is the subset that maximizes the sum

$$m_1(V^*) = \sum_{i \in V^*} \frac{B_i S_i}{B_i + S_i} \tag{3.22}$$

subject to $\theta > 1$ of

$$\sum_{i \in V^*} \frac{S_i}{B_i + S_i} < \frac{B_n}{B_n + S_n} \tag{3.23}$$

One can observe that this is *Binary Knapsack* [28] problem that can be solved using the *Horowitz-Sahni* algorithm [28].

Policy II ($\theta < 1$)

From equation (3.21), we can see that under this policy, V^* is the subset that minimizes the sum

$$m_2(V^*) = \left(\sum_{i \in V^*} \frac{S_i}{B_i + S_i} \right) / \left(\sum_{i \in V^*} \frac{B_i S_i}{B_i + S_i} \right) \tag{3.24}$$

subject to $\theta < 1$ or

$$\sum_{i \in V^*} \frac{S_i}{B_i + S_i} > \frac{B_n}{B_n + S_n} \tag{3.25}$$

To start with, we should initiate V^* with the first worker, W_0, that minimizes m_2.

Lemma 5 *$m_2(V^*)$ is minimum if $V^* = \{W_o\}$ such that $B_0 \geq B_i$, $\forall W_i \in V$.*

Proof. Consider an arbitrary subset $X \subseteq V$, $X = \{P_1, P_2, ...P_r\}$. We have:

$$B_0 \geq B_i \Rightarrow \sum_{i=1}^{r} \frac{B_0 S_i}{B_i + Si} > \sum_{i=1}^{r} \frac{B_i S_i}{B_i + Si} \quad (\forall P_i \in V) \Rightarrow B_0 \sum_{i=1}^{r} \frac{S_i}{B_i + Si} > \sum_{i=1}^{r} \frac{B_i S_i}{B_i + Si}$$
$$\Rightarrow \left(\frac{S_0}{B_0 + S_0}\right) / \left(\frac{B_0 S_0}{B_0 + S_0}\right) < \left(\sum_{i=1}^{r} \frac{S_i}{B_i + Si}\right) / \left(\sum_{i=1}^{r} \frac{B_i S_i}{B_i + Si}\right) \Rightarrow m_2(V^*) < m_2(X) \tag{3.26}$$

After adding W_0 to V^*, we should keep conservatively adding more workers until constraint (3.25) is satisfied. In fact, the next W_k that should be added to V^* is the one that satisfies the following inequality:

$$m_2(V^* \cup \{W_k\}) \leq m_2(V^* \cup \{W_j\}) \quad \forall W_j \in (V - V^*)$$

The *Greedy* algorithm described below progressively adds more P_k until V^* satisfies (3.25), i.e. until (θ¡1). The run time complexity of this search is $O(n)$.

Algorithm 3.4.2: GREEDY($V, thetaCondition$)

---

main
Search $W_n \in V$ such that:
  $B_n/(B_n + S_n) \leq B_i/(B_i + S_i) \quad \forall W_i \in V$
Search $W_0 \in V$ such that:
  $B_0 \geq B_i \quad \forall W_i \in V$
$V^* = \{W_n, W_0\}$;
$V = V - V^*$;
repeat
  Search worker W_k satisfy
    $m_2(V^* \cup \{W_k\}) \leq m_2(V^* \cup \{W_j\}) \quad \forall W_j \in V$
  $V^* = V^* \cup \{W_k\}$;
  $V = V - \{W_k\}$;
until *thetaCondition*;
return(V^*);

Policy III (θ=1)

Under this policy, we need to find V^* that minimizes the following makespan function:

$$m_3(V^*) = \left(\sum_{i \in V^*} \frac{S_i}{B_i + S_i}\right) / \left(\sum_{i \in V^*} \frac{B_i S_i}{B_i + S_i}\right) \tag{3.27}$$

subject to $\theta = 1$ or

$$\frac{B_n}{(B_n + S_n) \sum_{i \in V^*} \frac{S_i}{B_i + S_i}} = 1 \tag{3.28}$$

It is noticeable that $m_3()$ is the same as $m_2()$ (Policy II). However, the two objective functions differ with respect to their constraints. Therefore, we can use the same *Greedy* search algorithm explained earlier with the exception that the termination condition should be $\theta = 1$ (instead of $\theta > 1$).

3.4.4 Worker Selection Policy Using Branch and Bound Method

Similar to the above method, the Policy I ($\theta > 1$) can be solved by using Horowitz-Sahni algorithm [28]. It is noticeable that the objective functions of Policy II and III are the same. Therefore, we can use the same algorithm OSS (Optimal Subset Search) as presented below. The optimization problem mentioned in (3.25) and (3.28) can be stated as follow:

$$\pi(V^*) = \left(\sum_{i \in V^*} \frac{S_i}{B_i + S_i}\right) / \left(\sum_{i \in V^*} \frac{B_i S_i}{B_i + S_i}\right) \tag{3.29}$$

subject to $\theta = 1$ or

$$\sum_{i \in V^*} \frac{S_i}{B_i + S_i} \geq Z$$

where Z is a constant:

$$Z = \frac{B_n}{(B_n + S_n)}$$

This is an Integer Nonlinear Optimization [28] problem. We design a Branch and Bound algorithm called *OSS* (Optimal Subset Search) to solve it. To begin with, let us denote by Ω_V the set of subsets of V: $\Omega_V = \{X \subseteq V\}$.

Lemma 6 *The following function is a lower bound of function $\pi()$, i.e. $Lower(X) \leq \pi(X)$ ($\forall X \subseteq V$)*

$$\begin{aligned} Lower: \ & \Omega_V \longrightarrow R \\ & X \longmapsto \tfrac{1}{B_k} : B_k = max\{B_i : W_i \in X\} \end{aligned}$$

Proof. Assume that $X = \{W_1, W_2, ..., W_r\}$. We have:

$$\forall i : B_k \geq B_i \Rightarrow B_k \sum_{i=1}^{r} \frac{S_i}{B_i + S_i} \geq \sum_{i=1}^{r} \frac{B_i S_i}{B_i + S_i}$$

$$\Rightarrow \left(\frac{S_k}{B_k + S_k}\right) / \left(\frac{B_k S_k}{B_k + S_k}\right) \leq \left(\frac{S_i}{B_i + S_i}\right) / \left(\frac{B_i S_i}{B_i + S_i}\right) \Rightarrow L(X) \leq \pi(X)$$

Let us denote:

- $\hat{u} = (\hat{u_1}, \hat{u_2}, ..., \hat{u_n})$: the current solution, $\hat{u_i} \in \{0, 1\}$. $\hat{u_i} = 1$ if worker i is selected (W_i belongs to V^*), otherwise $\hat{u_i} = 0$
- $(u = u_1, u_2, ...u_n)$: the best solution so far, $u_i \in \{0, 1\}$.
- $\hat{a}$: the value of the current solution, i.e. the value of $\pi()$ at the subset correspond with $\hat{u}$

$$\hat{a} = \left(\sum_{i=1}^{n} \frac{S_i}{B_i + S_i} \hat{u_i}\right) / \left(\sum_{i=1}^{n} \frac{B_i S_i}{B_i + S_i} \hat{u_i}\right)$$

- a: the value of the best solution so far, i.e. the value of $\pi()$ at the subset correspond with u

$$\hat{a} = \left(\sum_{i=1}^{n} \frac{S_i}{B_i + S_i} u_i\right) / \left(\sum_{i=1}^{n} \frac{B_i S_i}{B_i + S_i} u_i\right)$$

The following functions have been used for computing the value of $\pi(\hat{u})$ at the current solution $\hat{u}$

Function Numerator($\hat{u}$)
/* *return the value of the numerator of* $\pi()$ *at the subset of* V *that correspond with* $\hat{u}$ */
begin
y=0;
for i:=1 **to** n **do**
if ($\hat{u}_i$=1) **then** y:= $y + S_i/(B_i + S_i)$;
return(y);
end
Function Pi($\hat{u}$)
/* *return the value of* $\pi()$ *at the subset of* V *that correspond with* $\hat{u}$ */
begin
y=0;
for i:=1 **to** n **do**
if ($\hat{u}_i$=1) **then** y:= $y + B_i S_i/(B_i + S_i)$;
return(Numerator($\hat{u}$)/y);
end

In OSS algorithm, a *forward move* consists of inserting the next worker W_j into the current solution $\hat{u}$. A *backtracking move* consists of removing the last inserted worker from the current solution. After a *backtracking move*, the lower bound *Lower()* corresponding to the current solution is computed and compared with the value of the best solution so far a in order to check whether a further forward move could lead to a better one. If the answer was affirmative, a new forward move is performed, otherwise a backtracking follows. When the last worker W_n has been considered, the best solution is updated. The algorithm terminates when no further backtracking can be performed.

Algorithm 3.4.3: OPTIMAL_SUBSET_SEARCH (OSS)(V)

Input: V
Output: u, a, V^*
main
begin
/* *Initialize* */
$a := \infty$; $\hat{a} := 0$; $j := 1$;
while (true) **do**
begin
1. /* estimate the lower bound */
find B_{max}=max$\{B_{j+1}, B_{j+2}, ..., B_n\}$;
$Lower := 1/B_{max}$;
if $a \leq \hat{a} + Lower$ **then go to** step 2;
/* forward */
$\hat{u}_j := 1$;
$\hat{a}$:=Pi($\hat{u}$);
$j := j + 1$;
if $j \leq n$ **then go to** step 1;
if Numerator($\hat{u}$) $> Z$ **then**
/* update the best solution */
begin
$u := \hat{u}$;
$a := \hat{a}$;
end
/* remove worker j from the current solution */
$\hat{u}_n := 0$
$\hat{a}$:=Pi($\hat{u}$);
2. /* back track */
find $i = max\{k \mid k < j$ and $\hat{u}_k = 1\}$
if no such i **then return**;
$\hat{u}_i := 0$;
$j := i + 1$;
go to step 2;
end
end

3.4.5 MRRS vs. UMR: Analytical Comparison

In this section we analytically show how MRRS is always better than UMR through the following lemmas.

Lemma 7 *If the MRRS and UMR algorithms end up with the same set of selected workers (V^*) then* $makespan_{MRRS}(V^*) \leq makespan_{UMR}(V^*)$

Proof. Let us sort the n workers of V^* by S_i/B_i in an increasing order:

$$\frac{S_1}{B_1} < \frac{S_2}{B_2} < ... < \frac{S_n}{B_n} < \frac{1}{n} \tag{3.30}$$

We can write

$$\frac{B_n}{S_n} > n \Rightarrow \frac{B_n}{B_n + S_n} > n\frac{S_n}{B_n + S_n} \tag{3.31}$$

Concurrently, from inequality (3.30) we derive

$$\frac{S_n}{B_n + S_n} > \frac{S_i}{B_i + S_i} \quad (\forall i = 1, 2, ...n) \tag{3.32}$$

Combining inequalities (3.31) and (3.32) we derive:

$$\frac{B_n}{B_n + S_n} > \sum_{i=1}^{n} \frac{S_i}{B_i + S_i} \Rightarrow \theta > 1$$

In the case of $\theta > 1$, $makespan_{MRRS}(V^*)$ is computed by equation (3.20)

$$makespan_{MRRS}(V^*) = L_{total}\frac{B_n}{(B_n + S_n)\sum_{i\in V^*}\frac{B_iS_i}{B_i+S_i}} + C \tag{3.33}$$

From equation (3.10) we derive:

$$makespan_{UMR}(V^*) = \frac{L_{total}}{\sum_{i\in V^*} S_i}\left(1 + \frac{1-\phi}{\phi - \phi^{m+1}}\right) + C$$

From equation (3.30) we have:

$$\sum_{i=1}^{n}\frac{S_i}{B_i} < 1 \Rightarrow \phi > 1 \Rightarrow lim_{m\to\infty}\frac{1-\phi}{\phi - \phi^{m+1}} = 0$$

and since m (the number of rounds) is usually large (in our experiments, m is in hundreds), we can write:

$$1 + \frac{1-\phi}{\phi - \phi^{m+1}} \approx 1$$

So we have

$$makespan_{UMR}(V^*) = \frac{L_{total}}{\sum_{i\in V^*} S_i} + C \tag{3.34}$$

From equation(3.30) we derive

$$\frac{B_n}{B_n + S_n} \leq \frac{B_i}{B_i + S_i} \quad (\forall i = 1, 2, ..., n) \Rightarrow \frac{B_n}{B_n + S_n}\sum_{i=1}^{n} S_i \leq \sum_{i=1}^{n}\frac{B_iS_i}{B_i + S_i} \Rightarrow$$
$$\Rightarrow \frac{B_n}{(B_n + S_n)\sum_{i=1}^{n}\frac{B_iS_i}{B_i+S_i}} \leq \frac{1}{\sum_{i=1}^{n} S_i}$$

By considering equations (3.33) and (3.34) we derive:

$$makespan_{MRRS}(V^*) \leq makespan_{UMR}(V^*)$$

Lemma 8 *With an arbitrary set of workers V, we have* $makespan_{MRRS}(V) \leq makespan_{UMR}(V)$

Proof. Let us denote:

- $P \subseteq V$is the subset which chosen by UMR
- $Q \subseteq V$ is the subset which chosen by MRRS
- V_1^*, V_2^* and V_3^* are subsets chosen by MRRS algorithm in three cases: $\theta > 1, \theta = 1$ and $\theta < 1$, respectively.

Using Lemma (7) we have:

$$makespan_{UMR}(P) \geq makespan_{MRRS}(P) \tag{3.35}$$

As discussed in the last section, V_1^* is an optimal solution of the Knapsack system produced by the *Horowitz-Sahni* algorithm. So we have:

$$makespan_{MRRS}(P) \geq makespan_{MRRS}(V_1^*) \tag{3.36}$$

Because B is chosen by MRRS by comparing V_1^*, V_2^* and V_3^* so we have

$$makespan_{MRRS}(V_1^*) \geq makespan_{MRRS}(Q) \tag{3.37}$$

From equations (3.35), (3.36) and (3.37) we derive

$$makespan_{UMR}(P) \geq makespan_{MRRS}(Q) \tag{3.38}$$

Finally we derive

$$makespan_{UMR}(V) \geq makespan_{MRRS}(V) \tag{3.39}$$

3.4.6 Experimental Results

In order to evaluate our new algorithm, MRRS, we developed a simulator using the SIMGRID toolkit [29, 30, 31], which has been used to evaluate the UMR algorithm. We conducted a number of experiments that aim at

- showing the validity of our approximation assumptions discussed in subsection 3.4.3
- showing that the MRRS algorithm is superior to its predecessor multi-round algorithms, namely LP and UMR.

Validity of Approximation Assumptions

The experiments we conducted show that the absolute deviation between theoretically computed makespan, as analyzed in sub section 3.4.3, and the makespan observed through the simulation experiments is negligible. This confirms that the approximation assumptions adopted in our analysis are plausible. Table (3.1) outlines the parameters that we used in our experiments. Let us denote:

- MK_e is the makespan obtained from the experiments.

Table 3.1: Experiment Parameters

Parameter	**Value**
Number of workers	$N = 50$
Total workload (flop)	10^6
Computation speed (flop/s)	Randomly selected from $[S_{min}, 1.5 \times S_{min}]$, where $S_{min} = 50$
Communication rate (flop/s)	Randomly selected from $[0.5 \times N \times S_{min}, 1.5 \times S_{min}]$

Table 3.2: The Absolute Deviation between the Experimental and Theoretical Makespans

$nLat, cLat$ (s)	D_1 (%)	D_2 (%)	D_3 (%)
1	3.15	2.42	3.34
10^{-1}	2.23	1.75	2.27
10^{-2}	1.51	0.92	1.94
10^{-3}	0.82	0.51	1.25

- MK_1, MK_2, MK_3 are the makespans computed by formulas (3.18), (3.20, (3.21) respectively.
- D_i (i=1, 2, 3) is the absolute deviation between the theoretical makespan, MK_i, and the experimental makespan MK_e. Therefore:

$$D_i = 100 \times \frac{|MK_i - MK_e|}{MK_e}(\%) \quad i = 1, 2, 3$$

Table (3.2) summarizes the absolute deviations computed for different latencies. From these results we can make the following remarks:

- The absolute deviation between the theoretical and the experimental makespans ranges from 0.5% to 3.1%, which is negligible.
- We notice that $D_2 < D_1 < D_3$. The justification is that the absolute deviation (D) is proportional to the number of participating workers in a given selection policy. The more workers participate, the larger D become. As we recall that D_2 represents the deviation caused by policy II ($\theta > 1$), which is the most conservative policy with respect to the number of workers allowed to participate. D_3 represents the deviation caused by policy III ($\theta < 1$), which is the most relaxed policy with respect to the number of participating workers. D_1 of policy I ($\theta = 1$) falls in the middle with respect to the number of participating workers and according the observed deviation.

3.5 Comparison with Previous Algorithms

We compare MRRS with the most powerful scheduling algorithm, namely UMR [17, 18] and LP [15, 16]. Table 3.3 outlines the configuration parameters used in the simulation

Table 3.3: Simulation parameters

Parameter	**Value**
N number of workers	N = 10, 12, 14,..., 50
Total workload (flop)	5×10^5
Computation speed (flop/s)	Randomly selected from $[S_{min}, 1.5 \times S_{min}]$, where S_{min} = 5, 10, 15, 20
Communication rate (flop/s)	Randomly selected from $[0.5 \times N \times S_{min}, 1.5 \times S_{min}]$
Computation and communication latencies (s)	10, 10^{-1}, 10^{-2}

Table 3.4: Performance comparisons among MRRS, UMR and LP Algorithms

Algorithm	Normalized Makespan	Rank	Degradation from the best
MRRS	1	0.12	0.65
UMR	1.21	0.88	21.4
LP	1.59	2	59.8

experiments. The performances of these algorithms have been compared with respect to three metrics:

- The normalized makespan, that is normalized to the run time achieved by the best algorithm in a given experiment;
- The rank which ranges from 0 (best) to 2 (worst);
- The degradation from the best, which measures the relative difference, as a percentage, between the makespan achieved by a given algorithm and the makespan achieved by the best one.

These metrics are commonly used in the literature for comparing scheduling algorithms [17, 18].

The results summarized in Table 3.4 suggests that MRRS can outperform its competitors in most of the cases. MRRS's rank drop from the best to the second in 12% of the cases with 5.4% performance degradation in comparison with the UMR.

Figure (3.3) shows that UMR has chances of outperforming MRRS only if the number of workers is small ($n \leq 20$), because in those cases, the worker selection module of MRRS does not have enough workers, which denies the MRRS from adopting one of the worker selection policies, namely Policy II. LP has almost no chance to win. This is due to the fact that LP does not have any effective strategy of reducing the idle time of workers at the end of each round.

Figure 3.3: The relation between the number of workers n and the Makespan

3.6 Summary

The ultimate goal of any scheduling algorithm is to minimize the makespan. UMR and LP are among these algorithms that have been designed to schedule divisible loads in heterogeneous environments where workers have different CPU speeds connected to links with different bandwidths. However, these algorithms do not take into account a number of the chief parameters such as bandwidths and the inevitable latencies of communication and computation. Furthermore, present algorithms are not equipped with a resource selection mechanism as they assume that all available workers will participate in processing the workload. In this work, we presented the MRRS algorithm that divides the workload into chunks in light of more realistic parameters mentioned earlier.

We explained the MRRS's worker selection policy which is, to the best of our knowledge, the first algorithm that addresses the resource selection problem. Having such policy is indispensable in large commuting platform such as the Grid, where thousands of workers are accessible but the best subset must be chosen.

The simulation experiments show that MRRS is superior to its predecessors especially when it is put into operation in a colossal WAN environment such as the Grid, which agglomerates an abundant pool of heterogeneous workers. The original UMR might still be the best choice in a LAN environment where the number of workers is typically limited.

Chapter 4

Dynamic Scheduling Method for Divisible Load in Non-Dedicated Distributed Environments

4.1 Introduction

The main issue of scheduling process is how to find an optimal division of total workload into workers. Up to now many approaches have been proposed. A simple solution, called *Single Round* scheduling algorithm [15], divides the workload in as many part as workers, after that send each part to appropriate worker. Because of each worker receives its load only one time, hence this method is named Single Round. *Multi Round* scheduling algorithms, mentioned in the last chapter and [4, 16, 17], split the overall process into many consecutive rounds. In each round, the master delivers chunks to each of workers in turn.

However, all of the above approaches assume that computational resources at workers are dedicated, i.e. the performances of workers are stable during execution time. This assumption renders these algorithms impractical in Non-Dedicated environments such as Grids where computational resources are expected to serve local tasks in addition to the Grid tasks. The inevitable variation of workers' power in the Grid embodies a non-trivial challenge for scheduling (split and distribute) workloads to workers.

Algorithm RUMR [19] is designed to tolerate performance prediction errors by using *Factoring* method, however all of its parameters are fixed before RUMR starts, which makes RUMR a non-adaptive scheduling algorithm. In [2], the authors use M/M/1 queue to model the tasks processing, however, [2] lacks an efficient prediction strategy because it is merely based on probability parameters. In addition, the work in [2] does not address the needs of divisible workloads scheduling, but of DAG workload type.

Apparently, dynamic algorithms [33, 34] are more appropriate for Grids. Up to our knowledge, [35] is the first effort to develop a dynamic scheduling algorithm for divisible workloads. In [35] we use *Mixed-Tendency Based* [37, 38] prediction strategy to estimate worker's power. However, Mixed-Tendency Based strategy considers the aggregate execution of applications, while our computation model presented in this paper, is based on the M/M/1 queuing model [39] that considers both local and Grid task in their own, rendering it a more realistic model.

In this chapter we develop a dynamic multi-round scheduling algorithm for non-

dedicated environments. In order to find the optimal division of workload in each round, we need to forecast, as accurate as possible, the available CPU power of each worker before the division happens. Thus we need an effective prediction strategy. This chapter can be summarized as follows:

- First, we presents a model to represent worker's activities with respect to processing local and external Grid tasks. Unlike the work done in [4, 16, 17], this model helps estimate the computing power of a worker under the fluctuation of number of local and Grid applications in the system.
- Second, we develop new strategy 2PP for predicting the computing power of processors, i.e., the portion of original CPU power that the owner can donate to Grid applications.
- Third, we propose a new dynamic scheduling algorithm 2PP by incorporating the performance model and the prediction method described above into the static algorithm MRRS that mentioned in the last Chapter, which is originally a static scheduling algorithm.
- Fourth, we apply an existing prediction algorithm, Mixed Tendency-Based Prediction [37, 38], in developing dynamic scheduling algorithm DSA.
- Lastly, we conduct the experiments for comparing between proposed dynamic scheduling algorithms 2PP and DSA as well as for comparing these proposed algorithms with the existing static scheduling algorithm such as UMR.

4.2 Problem Statement

The task scheduling problem in non-dedicated environments can be defined as follows. Given:

- Divisible workload L_{total} that resides at the master.
- Non-dedicated computational platform consists of the master and n workers, computational speed of the worker W_i is S_i with latency $cLat_i$.
- Data transfer rate of the connection link between the master and worker W_i is B_i with latency $nLat_i$
- S_i varies over time ($i = 1, 2, ..., n$). This is nature of non-dedicated environments.

Our ultimate question is: given the above platform settings, in what proportion should the workload L_{total} be split up among the heterogeneous, dynamic workers so that the overall execution time is minimum?

Formally, we need to minimize the following objective function:

$$max_{i=1,2,...,n}\left[\sum_{k=1}^{i} Tcomm_{1,k} + \sum_{j=1}^{m} Tcomp_{j,i}\right] \rightarrow min \tag{4.1}$$

where the expression between brackets is the total running time, i.e., the sum of waiting time, communication time and computation time of worker W_i.

4.3 Non-Dedicated Computation Model

4.3.1 Heterogeneous Computation Platform

Let us consider a computation Grid, in that, a master process controls n worker processes and each process runs on a particular computer. The Grid runs on a heterogeneous platform, i.e., workers can have different CPU powers and different network bandwidths. The master can divide the total workload, L_{total}, into arbitrary chunks and deliver them to the appropriate workers. We assume that the master uses its network connection in a sequential fashion, i.e., it does not send chunks to some workers simultaneously. Workers can receive data from network and perform computation simultaneously.

We keep using the heterogeneous computation platform mentioned in the last chapter, with some addition.

- L_{total}: the total amount of workload.
- W_i: worker number i, some time called P_i.
- n: total number of workers that are actually selected to process the workload
- m: the number of rounds.
- $chunk_{j,i}$: the fraction of total workload that the master delivers to worker W_i in round j $(i = 1, .., n;\ j = 1, .., m)$
- S_i: computation speed of the worker i measured by the number of units of workload performed per second $(flop/s)$
- ES_i: estimated average speed of worker W_i for Grid tasks on the next round. ES_i is derived from equation (4.15).
- B_i: the data transfer rate of the connection link between the master and worker W_i $(flop/s)$
- $Tcomp_{j,i}$: we model the time required for worker i to perform the computation $chunk_{j,i}$ as:

$$Tcomp_{j,i} = cLat_i + \frac{chunk_{j,i}}{ES_i}$$

- $cLat_i$: the fixed overhead time, in seconds, for starting a computation (e.g. for starting a remote process) in the worker W_i. The computation, including the $cLat_i$ overhead, can be overlapped with communication.
- $nLat_i$: the overhead time, in seconds, incurred by the master to initiate a data transfer to W_i (e.g. pre-process application input data and/or initiate a TCP). We denote total latencies by $Lat_i = cLat_i + nLat_i$.
- $Tcomm_{j,i}$: we model the communication time spent by the master to send $chunk_{j,i}$ units of data to worker W_i as:

$$Tcomm_{j,i} = nLat_i + \frac{chunk_{j,i}}{B_i} \tag{4.2}$$

- $round_j$: the fraction of workload dispatched during round j

$$round_j = \sum_{i=1}^{n} chunk_{j,i} \tag{4.3}$$

We fix the time required for each worker to perform communication and computation during each round

$$cLat_i + \frac{chunk_{j,i}}{S_i} + nLat_i + \frac{chunk_{j,i}}{B_i} = const_j \tag{4.4}$$

We set

$$A_i = \frac{B_i \times ES_i}{B_i + ES_i} \tag{4.5}$$

so we have

$$chunk_{j,i} = \alpha_i round_j + \beta_i \tag{4.6}$$

where

$$\alpha_i = \frac{A_i}{\sum_{k=1}^{n} A_k}; \quad \beta_i = A_i \frac{\sum_{k=1}^{n} A_k (Lat_k - Lat_i)}{\sum_{k=1}^{n} A_k} \tag{4.7}$$

Most static scheduling algorithms [4, 18, 16] assume that the execution time of a workload chunk is well-known based on the assumption that workers have guaranteed availability of fixed, predefined CPU power. On a non dedicated, dynamic platform such as Grid, these assumptions are not realistic. Thus in this section we present a model of executing local and Grid tasks at a given, non-dedicated worker.

4.3.2 Markovian Queue M/M/1

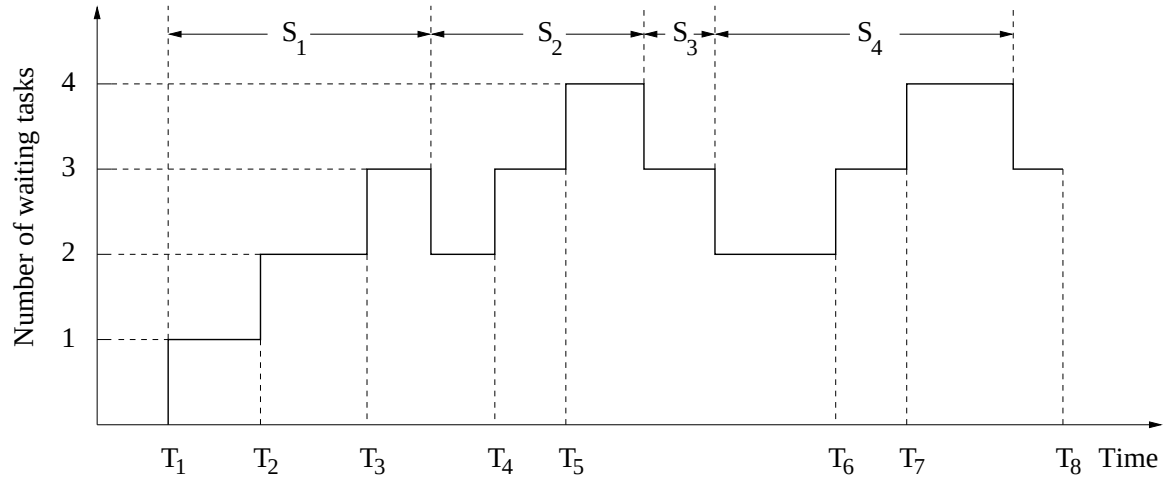


Figure 4.1: Arrivals and departures at a queue. $\{T_i\}$ refer to the arrival instants, $\{S_i\}$ refer to the service times

Per [39] a queue or a waiting line is formed by arriving customers (local tasks and Grid tasks in our case) requiring service from a service station (workers W_i in our case). If service is not immediately available, the arriving tasks may join the queue and wait for service and leave the system after being served (see Figure (4.1). In the meantime, other tasks may arrive for service. We assume that the service system has unlimited capacity (waiting room capacity) for holding both local tasks and Grid tasks. The basic features of our queue are

- The input process: the arriving tasks consist of Grid tasks and local tasks. Grid tasks are the portions of total load L_{total} that are delivered by the server. The local tasks are produced by local applications at the workers.

- The service mechanism: during the execution of a Grid task on a certain worker, some local tasks may arrive causing to interrupt the execution of the lower priority Grid tasks. We consider the execution of the local tasks as preemptive, i.e. a local task must be executed until completion once it is started. The execution of the local tasks follow the rule of *first come first served*.

- The worker's capacity. From the view point of the Grid tasks, the state of a worker alternates between available and unavailable. When the worker is executing its own local tasks, it is unavailable for the Grid tasks, otherwise its state is available. The original computation of worker W_i is S_i.

We assume that the arrival of the local tasks of worker W_i is assumed to follow a Poisson distribution with arrival rate λ_i, their execution process follows an exponential distribution with service rate μ_i and the local task process in the worker is an M/M/1 [39] queuing system ($i = 1, 2, ..., n$) (Figure (4.2)).

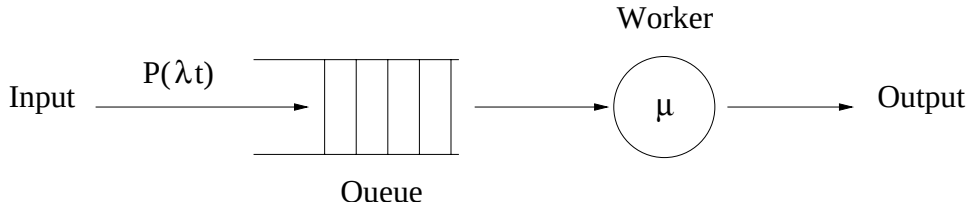


Figure 4.2: M/M/1 queue

The execution time of $chunk_{j,i}$ on the worker W_i can be expressed as:

$$Tcomp_{j,i} = X_1 + Y_1 + X_2 + Y_2 + ... + X_{NL} + Y_{NL} \tag{4.8}$$

where:

- NL: the number of local tasks which arrive during the execution of workload $chunk_{j,i}$.

- Y_k: execution time of the local task k ($k = 1, 2, ..., NL$), these are independent identical distribution random variables.

- X_k: execution time of k^{th} section of $chunk_{j,i}$ ($k = 1, 2, ..., NL$). We have:

$$X_1 + X_2 + ... + X_{NL} = \frac{chunk_{j,i}}{S_i} \tag{4.9}$$

From the M/M/1 queuing theory [39] we have:

$$E\left(NL\right) = \frac{\lambda_i chunk_{j,i}}{S_i}; \quad E\left(Y_k\right) = \frac{1}{\mu_i - \lambda_i} \tag{4.10}$$

Because of NL and Y_k are independent random variables ($k = 1, 2, ..., NL$) we derive

$$\begin{aligned} E\left(Tcomp_{j,i}\right) = E\left(Tcomp_{j,i}|NL\right) &= \sum_{k=1}^{NL} X_k + \sum_{k=1}^{NL} E(Y_k) \\ &= \frac{chunk_{j,i}}{S_i} + E(NL) \times E(Y_k) = \frac{chunk_{j,i}}{S_i(1-\rho_i)} \end{aligned} \tag{4.11}$$

where $\rho_i = \lambda_i/\mu_i$ represents the CPU utilization. For a worker W_i with the CPU utilization ρ_i we can express the computation time of the $chunk_{j,i}$ as

$$Tcomp_{j,i} = \frac{chunk_{j,i}}{S_i(1-\rho_i)}$$

However λ_i, μ_i, ρ_i are representative of the dynamicity of the environment during a long time. They do not exactly reflect the dynamicity of the environment during a short interval such as the execution time of an application. Therefore, we introduce the *adaptive factor* δ_i, which represents the credibility of performance prediction for worker W_i and it is initialized to 1 at the beginning of the scheduling process (i.e., in the first round). At the end of each round afterward, δ_i is computed as follows:

$$\delta_i = \frac{FS_i}{ES_i} \tag{4.12}$$

where FS_i denotes the factually measured available CPU power. Now the expected value of the execution time of $chunk_{j,i}$ is

$$Tcomp_{j,i} = \frac{chunk_{j,i} \times \delta_i}{S_i(1-\rho_i)} \tag{4.13}$$

Since the actual power of workers available to the Grid tasks varies over time, we have to predict how δ_i changes. In the next section we describe 2 ways for prediction smoothing parameter δ_i, i.e. the CPU utilization:

- Prediction δ_i by using proposed 2PP strategy.
- Prediction δ_i by using an existing strategy called *Mixed Tendency Based*.

4.4 The 2-Phase Prediction (2PP) Strategy

4.4.1 Prediction Strategy

In order to minimize the execution time, we have to carry out two steps:

1. First, we estimate the available CPU power of each worker (ES_i) before each round commences using the 2PP strategy [36].

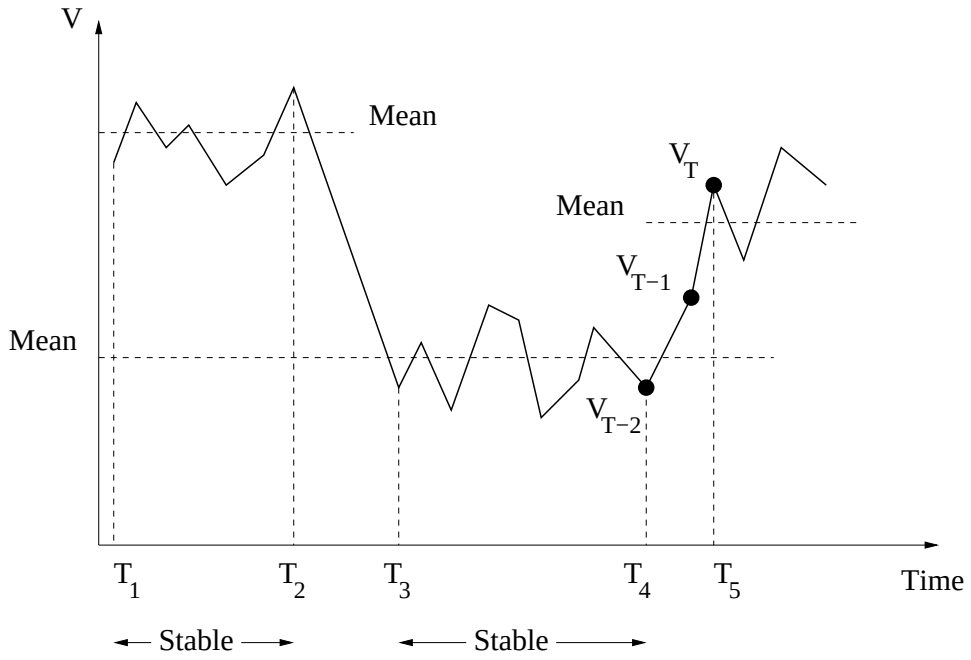


Figure 4.3: 2-Stage prediction strategy

2. Second, in light of the estimated ES_i, we carry out our dynamic scheduling algorithm that integrates the MRRS static algorithm with our 2PP strategy.

For the sake of clarity, in this section we drop the use of the subscript i that refers to worker W_i. In order to estimate the next δ for a particular worker, we consider the historically measured time series $c_1, c_2, ..., c_n$. Data point c_t is value of δ at time point t. This time series of δ is sampled at some frequency (e.g., 0.1 Hz) during the execution of a round. However, we are interested in estimating δ for the upcoming round, not for the upcoming time tick. Therefore, we need to compress the original time series into interval time series by aggregating the former as follows:

If we denote D as the aggregation degree, where

M = execution time of a round $\times$ frequency of original time series

Then the interval time series V_1, $V_2, ...V_k$ $(k = \lceil n/D \rceil)$ can be calculated as follows:

$$V_r = \frac{\sum_{j=1}^{D} \delta_{n-(k-r+1)D+j}}{D} \qquad (r = 1, 2, ..., k) \tag{4.14}$$

Each value V_r is the average value of the adaptive factor δ over a round. The 2PP strategy [36] operates on this V_r time series in order to predict V_{k+1} of the next round.

Since δ plays the role of a smoothing factor that progressively adjusts the estimated CPU power available for Grid tasks, we should expect that its interval average, V_r , should alternate between some periods of *stability* and others of *conversion* as shown in Figure 4.3. In the stable stage, the available CPU power exhibit less variation and approaches some mean. The time intervals (T_1, T_2) and (T_3, T_4) in Figure 4.3 are examples of the stable stage. In the conversion stage, the available CPU power tends to experience major changes due to increase or decrease in the arrival rate of local tasks. The time intervals (T_2, T_3) and (T_4, T_5) represent conversion stages. Toggling between different stages can be identified by comparing the current absolute deviation $|V_T - Mean|$ with a threshold value *threshold*.

Algorithm 4.4.1 depicts the 2PP strategy where:

- V_T: the value of current data point.
- V_{T-1}: the value of last data point.
- V_{T+1}: the estimated value of the next data point.
- $Mean$: the mean value of data points in current stage.
- T: current time point
- B: the starting point of current stage

Algorithm 4.4.1: 2PP STRATEGY()

```
begin
Initialization:
    CurrentStage:= stable;
    Threshold:= 2 × (V_2 − V_1);
repeat
 if CurrentStage = stable
   if |V_T − Mean| > threshold
   /* Conversion stage is starting */
   then
    begin
      UpdateThreshold();
      CurrentStage:= conversion;
      V_{T+1} := 2.V_T − V_{T−1};
    end
   /* Stable stage continue */
   else
    begin
      UpdateMean();
      V_{T+1} := 2.Mean − V_T;
    end
 if CurrentStage = conversion
   if (V_T − V_{T−1}) × (V_{T−1} − V_{T−2}) < 0
   /* Stable stage is starting */
   then
    begin
      CurrentStage:= stable;
      B:=T-1;
      UpdateMean();
      V_{T+1} := 2 × Mean − V_T;
    end
     /* Conversion stage continue */
   else
    V_{T+1} := 2V_T − V_{T−1};
until all of W_total is processed;
end
```

The procedure UpdateMean() is very simple:

$$Mean = \frac{V_B + V_{B+1} + ... + V_T}{T - B + 1}$$

Using UpdateThreshold() *threshold* gets updated as follows. If N denotes the number of historical thresholds, and $|V_T - Mean|$ denotes the current threshold value, then the

updated *threshold* is:

$$threshold = \frac{N \times threshold + |V_T - Mean|}{N+1}$$

The predicted value of V_{T+1} is used as an estimate for the adaptive factor, δ, for the upcoming round. Consequently we can compute the average speed, ES_i, of worker W_i on the next round as follows:

$$ES_i = \frac{S_i \times (1 - \rho_i)}{\delta_i} \tag{4.15}$$

Henceforth, ES_i is deemed the speed of worker W_i over the next round.

4.4.2 Scheduling Algorithm

Algorithm 4.4.2: 2PP-BASED SCHEDULING()

---

begin
Initialization:
Collect the value of $\{B_i\}$, original CPU power $\{S_i\}$, $\{\lambda_i, \mu_i, \rho_i\}$ $(i = 1, 2, ..., n)$;
Use equation (4.15) to compute $\{ES_i\}$ $(i = 1, 2, ..., n)$;
Compute M, $round_0$, $\{chunk_{0,i}\}$ $(i = 1, 2, ..., n)$;
$W_{remains} = L_{total} - round_0$;
Deliver $\{chunk_{0,i}\}$ to $\{worker_i\}$ $(i = 1, 2, ..., n)$;
repeat
/* Processing on $round_j$ */
Collect items of the series C in the last round;
Use 2-Stages strategy to obtain $\{ \delta_i \}$ $(i = 1, 2, ..., n)$;
Use equations (4.19) and (4.15) to derive $round_j$ and $\{ES_i\}$ $(i = 1, 2, ..., N)$
if $(round_j > W_{remains})$ **then** $round_j = W_{remains}$;
$W_{remains} = W_{remains} - round_j$;
Deliver $\{chunk_{j,i}\}$ to $\{worker_i\}$ $(i = 1, 2, ..., N)$;
until $W_{remains} = 0$;
end

Algorithm 4.4.2 outlines the scheduling algorithm we present in this paper. The algorithm integrates the 2PP strategy, which estimates the available power of each worker during the coming round, with a slightly modified UMR scheduling algorithm. As shown in the figure, in order to schedule the workload, the master collects the initial values of $B_i, S_i, \lambda_i, \mu_i$ and ρ_i for each worker. It then computes ES_i and delivers the first round chunks to all respective workers.

The algorithm keeps running until no workload is remaining. It uses the 2PP strategy to estimate the ES_i for each worker before the start of each round. It then uses the UMR to decide how to split the workload into appropriate chunks to suit each worker's predicted availability of CPU power.

The original version of the UMR static algorithm can determine the exact number of rounds needed to distribute the entire workload a priori because it assumes fixed, dedicated CPU power of each worker. This is not the case in our non-dedicated environment where CPU powers vary over time. Consequently, the modified version of the UMR, as shown in the algorithm, halts when the remaining workload is zero ($W_{remains} = 0$), rather than checking whether a specific number of rounds has elapsed (as it is the case in the original UMR).

Another note to make is that the follow condition guarantees that all workers terminate at the same time:

if $(round_j > W_{remains})$ **then** $round_j = W_{remains}$;

4.5 Mixed Tendency Based Prediction Strategy

Prediction performance is necessary for efficient use of resources in Grid environments. Performance predictions can be useful to both applications and schedulers. Scheduler can use predictions to guide their scheduling strategies and thus to achieve higher application performance and more efficient resources use. Although the performance prediction strategies for divisible load are rare, the similar strategies are quite general in Grid computing.

There are many kind of environment parameter have been estimated, such as computation power, bandwidth rate, CPU load etc. Among CPU load prediction, there are some different kinds of strategies have been studied such as:

- Independent Dynamic Homeostatic prediction
- Independent Static Homeostatic prediction
- Mixed Tendency-based prediction [37, 38]
- ...

The last strategy, Mixed Tendency-based prediction, was chosen as prediction method for our dynamic scheduling algorithm. First, this strategy is designed for CPU load prediction in non-dedicated environments thus it corresponds with our goal (CPU utilization prediction). Second, as shown in [37, 38], Mixed Tendency-based prediction outperforms the nine predictors used within the Network Weather Service (NWS), a wide used performance prediction systems.

4.5.1 Prediction Strategy

We periodically measure δ and obtain the original preceding value time series $C = c_1, c_2, ..., c_n$. Data point c_i is value of δ at time point i.

M: aggregation degree, calculated as

M = *execution time of a round* $\times$ *frequency of original time series*

$\Delta = \delta_1, \delta_2, ..., \delta_k (k = \lceil n/M \rceil)$: the interval CPU load time series, calculated as

$$\delta_i = \frac{\sum_{j=1}^{M} c_{n-(k-i+1)M+j}}{M} \qquad (i = 1, 2, ..., k) \tag{4.16}$$

Each value δ_i is the average value of adaptive factor over a round. After collecting the original time series C and creating interval time series Δ, we apply the Mixed Tendency-based strategy [37, 38] to estimate the value in the next round δ_{k+1}.

Algorithm 4.5.1: MIXEDTENDENCY-BASED()

Procedure IncrementValueAdaptation()
begin
  $Mean := (\sum_{i=1}^{n} \delta_i)\,/n$;
  $RealIncValue := \delta_T - \delta_{T-1}$;
  $NormalInc := IncrementValue+$
    $+(RealIncValue - IncrementValue) \times AdaptDegree$;
  if $(\delta_T < Mean)$ **then** $IncrementValue := NormalInc$;
  else begin
    $PastGreater :=$ (number of data points greater than δ_T) / n;
    $TurningPointInc := IncrementValue \times PastGreater$;
    $IncrementValue := Min(NormalInc, TurningPointInc)$;
  end
begin
  if $(\delta_{T-1} < \delta_T)$ /* *Tendency is increase* */
  then begin
    IncrementValueAdaptation();
    $P_{T+1} := \delta_T +$ IncrementValue;
  end
  else if $(\delta_{T-1} > \delta_T)$ /* *Tendency is decrease*/
  then begin
    DecrementFactorAdaptation();
    $P_{T+1} := \delta_T \times$ DecrementFactor;
  end
end

Formally, Mixed Tendency-based prediction [37, 38] strategies can be expressed as above. The adaptation process in case of Increase and Decrease are similar. δ_T is the current value of adaptive factor, and P_{T+1} is the predicted value for δ_{T+1}. AdaptDegree is optional parameter that expresses the adaptation degree of the variation, its value can ranger from 0 to 1. Now we predict that the average speed ES_i of the worker W_i on the next round is

$$ES_i = \frac{S_i \times (1 - \rho_i)}{\delta_i}$$

where δ_i is predicted as explained above.

4.5.2 Scheduling Algorithm

Algorithm 4.5.2: DSA()

Collect the value of $\{B_i, S_i, \lambda_i, \mu_i, \rho_i\}$
Use equation (4.15) to derive $\{ES_i\}$ $(i = 1, 2, ..., n)$
Compute M, $round_0$, $\{chunk_{0,i}\}$ $(i = 1, 2, ..., n)$
$L_{remains} = L_{total} - round_0$;
Deliver $\{chunk_{0,i}\}$ to $\{worker_i\}$ $(i = 1, 2, ..., n)$
repeat
    /* Processing on round$_j$ */
    Collect items of the series C of last round
    Use Tendency-based Predictor to obtain $\{ \delta_i \}$ $(i = 1, 2, ..., n)$
    Use equation (4.19) and (4.15) to derive $round_j$ and $\{ES_i\}$ $(i = 1, 2, ..., n)$
    if $(round_j > L_{remains})$
  then $round_j = L_{remains}$;
    $L_{remains} = L_{remains} - round_j$;
    Deliver $\{chunk_{j,i}\}$ to $\{worker_i\}$ $(i = 1, 2, ..., n)$
until $L_{remains} = 0$;

By integrating the Mixed Tendency-Based prediction strategy and the static scheduling algorithm MRRS (mentioned in chapter 3), we develop a dynamic scheduling algorithm called DSA (Dynamic Scheduling Algorithm). The main idea of DSA is similar to the static algorithm MRRS except the following points:

- At the beginning, we collect all the values of parameters such as $\{B_i\}$, $\{S_i\}$. Note that these value are original value, i.e. the value in case of the system is dedicated.
- In the main iteration, the proposed algorithm DSA periodically collects the environments parameters to compute $\{\delta_i\}$.
- Chunks sizes is computed by applying the formulas presented in the below section. The partitions of load are executed at the beginning of each round with considering the updated values of the environments parameters such as $\{\delta_i\}$.
- The iterations will be continued until the total load L_{total} is finished.

4.6 Load Partition and Delivering

To partition workload among workers, we apply a scheduling algorithm similar to that described in the last chapter. Because of the computation power of workers varies during the execution time, the original CPU speed S_i is replaced by estimated value ES_i.

Induction Relation for Chunk Sizes

To fully utilize the network bandwidth, the dispatching of the master and the computation of W_n should finish at the same time:

$$\sum_{i=1}^{n}\left(nLat_i + \frac{chunk_{j+1,i}}{B_i}\right) = \frac{chunk_{j,n}}{ES_n} + cLat_n$$

If we replace $chunk_{j+1,i}$ and $chunk_{j,n}$ by their expression in (4.6) we derive:

$$round_{j+1} = \theta \times round_j + \mu \tag{4.17}$$

where

$$\theta = B_n\left((B_n + ES_n)\sum_{i=1}^{n}\frac{ES_i}{B_i + ES_i}\right)^{-1} \tag{4.18}$$

$$\mu = \left(\frac{\beta_n}{ES_n} + cLat_n - \sum_{i=1}^{n}\left(nLat_i + \frac{\beta_i}{B_i}\right)\right)\left(\sum_{i=1}^{n}\frac{\alpha_i}{B_i}\right)^{-1}$$

From induction equation (4.17) we can compute:

$$round_j = \theta^j(round_0 - \eta) + \eta \tag{4.19}$$

where

$$\eta = \frac{\beta_n + cLat_n - \sum_{i=1}^{n}\left(nLat_i + \frac{\beta_i}{B_i}\right)}{\sum_{i=1}^{n}\frac{\alpha_i}{B_i} - \frac{\alpha_n}{ES_n}}$$

Determining the Parameters of the Initial Round

In this section we compute the optimal number of rounds, m, and the size of the initial load fragment that should be distributed to workers in the first round, $round_0$. If we let $F(m, round_0)$ denotes the makespan, then we have

$$\begin{aligned} F(m, round_0) &= \sum_{i=1}^{n}\left(\frac{chunk_{0,i}}{B_i} + nLat_i\right) + \sum_{j=0}^{m-1}\left(\frac{chunk_{j,n}}{ES_n} + cLat_n\right) = \\ &= round_0\left(\sum_{i=1}^{n}\frac{\alpha_i}{B_i} + \frac{\alpha_n(1-\theta^m)}{ES_n(1-\theta)}\right) + \sum_{i=1}^{n}\left(\frac{\beta_j}{B_i} + nLat_i\right) + \\ &\quad + m\left(cLat_n + \frac{\alpha_n\eta + \beta_n}{ES_n}\right) - \frac{\alpha_n\eta(1-\eta^m)}{1-\eta} \end{aligned} \tag{4.20}$$

Our objective is to minimize the makespan $F(m, round_0)$, subject to:

$$\sum_{j=0}^{m-1} round_j = L_{total}$$

or

$$G(m, round_0) = m\eta + (round_0 - \eta)\frac{1-\theta^m}{1-\theta} - L_{total} = 0 \tag{4.21}$$

We use Lagrangian method [27] to solve this constrained minimization problem. The minimum value of function $F(m, round_0)$ can be found by solving the following equation system:

$$\frac{\partial L}{\partial \lambda} = G(m, round_0) = 0$$
$$\frac{\partial L}{\partial m} = \frac{\partial F}{\partial M} + \lambda \frac{\partial G}{\partial M} = 0$$
$$\frac{\partial L}{\partial round_0} = \frac{\partial F}{\partial round_0} + \lambda \frac{\partial G}{\partial round_0} = 0$$

where

- λ: Lagrange multiplier
- $L(m, round_0)$: Lagrangian function which is defined as:

$$L(m, round_0, \lambda) = F(m, round_0) + G(m, round_0)$$

After solving this equation system we obtain m. Using equation (4.21) one can then compute $round_0$. At last, using equations (4.19) and (4.6) we will obtain the value of $round_j$ and $chunk_{j,i}$ respectively ($i = 1..n$, $j = 1..m$).

4.7 Experimental Results

4.7.1 Simulation Results of the 2PP Scheduling Algorithm

In order to evaluate the new algorithm, we developed a simulator using the SIMGRID toolkit [29, 30, 31], which is specially designed for building simulations that help evaluate various scheduling algorithms in parallel and distributed environments. We compare the performance of our algorithm with the original UMR algorithm using two experimental configurations.

Configuration

In the first configuration, we have the following setup:

- Workers: we use 10 workers whose system properties are shown in Table 4.1.
- Total loads L_{total}: 1000 (Mega flops).
- The average processing time of the local tasks: 20 (second).

In order to add extra challenge to the behavior of the two scheduling algorithms, we intensify the arrival of local tasks on the strongest worker iRMX by ten times more than any other worker. As a result, this worker ends up, practically, being the weakest worker with respect to the available CPU power for the Grid tasks.

Unlike our 2PP-based algorithm, the UMR does not recognize this fact as it assumes that iRMX continually offers all of its power to the Grid tasks. Therefore, the UMR mistakenly keeps sending big chunks of workload to iRMX, which leads to performance

Table 4.1: Properties of workers in the first configuration

Name of workers	**CPU Power (Mflops/s)**	**Bandwidth (Mbps)**	**cLat (s)**	**nLat (s)**
iRMX	100	500	0.1	0.053
Kuenning	80	423	0.1	0.053
Bousquet	80	408	0.1	0.053
Soucy	30	169	0.1	0.053
Browne	30	153	0.1	0.053
Stephen	20	28	0.1	0.053
Robert	50	57	0.1	0.053
Sirois	40	45	0.1	0.053
Monique	20	24	0.1	0.053
Jackson	40	49	0.1	0.053

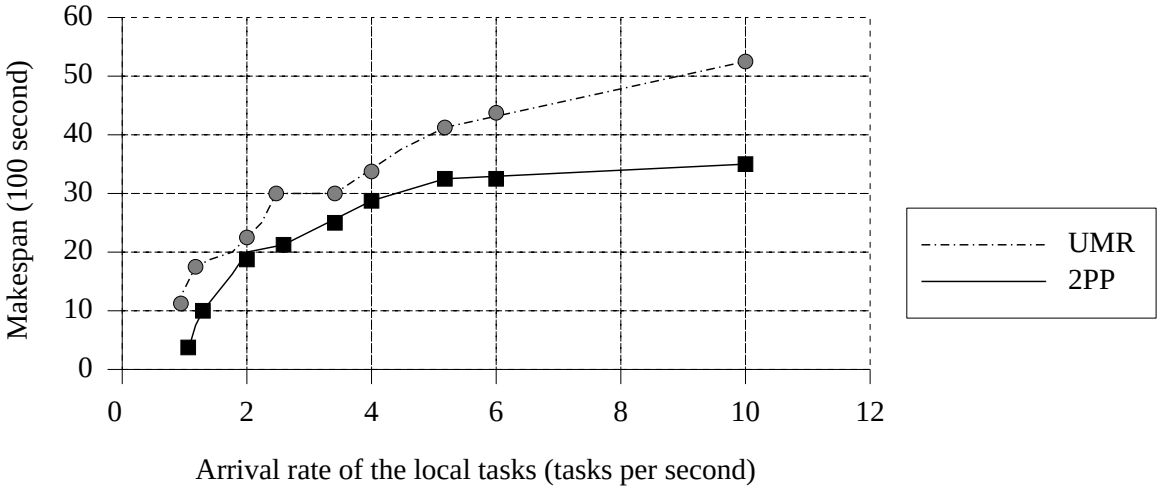


Figure 4.4: Configuration 1: 2PP vs. UMR

deterioration. Figure 4.4 shows the performance of the UMR vs. the 2PP-based algorithm under different arrival rate of local tasks. The 2PP algorithm keeps outperforming the UMR with respect to the task makespan.

Similarly, we experimented with second configuration setup that consists of:

- Number of workers: 30.
- The average power of worker: 60 (Mega flops per second).
- The average bandwidth: 50 (Megabyte per second).
- Total loads L_{total}: 5000 (Mega flops).
- Computation and communication latencies: 0.1 (second).
- The average processing time of the local tasks: 40 (second).

As we have done in the first configuration setup, we exposed the top 10% of the workers to higher concentration of arriving local tasks. Again, as shown in Figure 4.5, the 2PP outperforms the UMR as the latter is not aware of the run-time availability of the actual CPU power of workers.

Effect of local task on the makespan

Figures 4.4 and Figure 4.5 depict the comparison between the makespan of the proposed dynamic scheduling 2PP and the static scheduling algorithm UMR. It can be seen that:

- When the arrival rate of the local task intensifies, both of scheduling algorithm's performance decrease.
- However, the makespan deviation between 2PP and UMR increase proportion with the arrival rate of the local task. This mean the dynamic algorithm 2PP has the better adaptive capacity than static algorithm UMR.

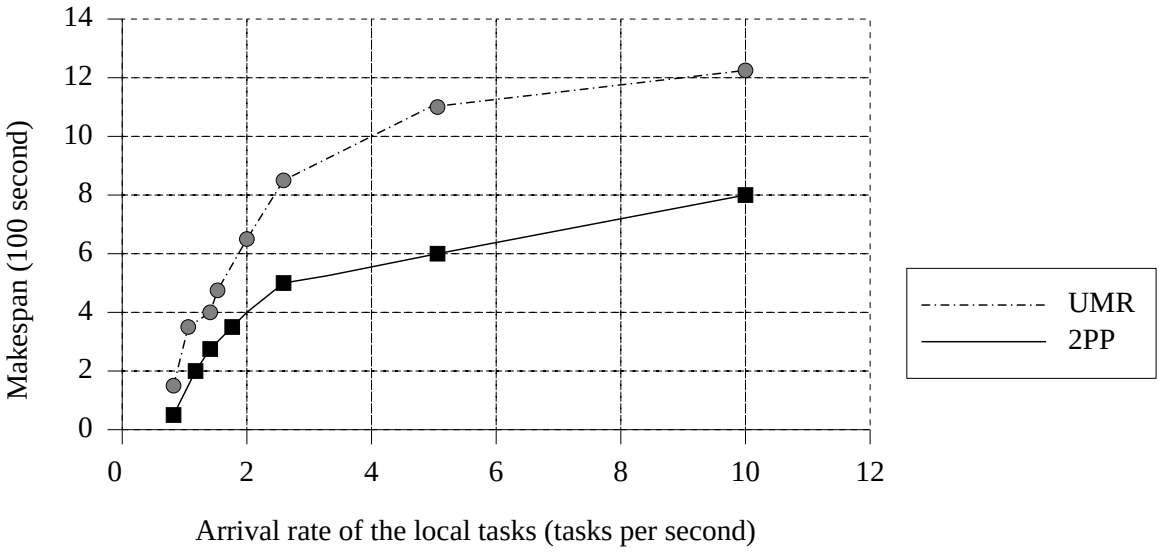


Figure 4.5: Configuration 2: 2PP vs. UMR

4.7.2 Simulation Results of the DSA Scheduling Algorithm

In order to evaluate the proposed algorithm DSA, we also use the SIMGRID toolkit [29, 30, 31] for developing the simulator. We conduct 6 experiments, numbered from 1 to 6 in the Table 4.2, in order to compare the performance of our algorithm DSA with the original UMR algorithm. These experiments using following configurations :

- The number of workers, the average power of worker and the total workload are listed in Table 4.2
- The average bandwidth: 50 (Megabyte per second).
- Computation and communication latencies: 0.1 (second).
- The ratio of Grid task's size to local task's size as described in each Table.

Table 4.2: The parameters of 6 experiments

Experiment number	1	2	3	4	5	6
Frequency of local tasks (tasks/s)	0.036	0.18	0.89	1.75	0.04	0.25
Total workload (Gflop)	1	1	2	2	5	5
Number of workers	10	15	20	25	50	90
CPU power on average (Mflop/s)	32	24	37	44	63	47

Table 4.3: Impact of the local tasks: properties of workers used in the experiment

Name of workers	**CPU Power (Mflops/s)**	**Bandwidth (Mbps)**	**Estimated CPU power**	**Factual CPU power**
iRMX	100	500	53	61
Kuenning	80	400	79	72
Bousquet	80	400	80	73
Soucy	30	200	23	26
Browne	30	200	22	25

Impact of local task on the makespan

Our experiments show that our DSA algorithm outperforms the static algorithm UMR with the makespan deviation from 10% to 30%. Figure 4.6, Figure 4.7 and Figure 4.8 depict the comparison between the makespan of the proposed dynamic scheduling algorithm DSA and the static scheduling algorithm UMR. It can be seen that, when the ratio Grid task size/Local task size go down, i.e. when the total amount of local task size increases, the makespans of DSA and UMR are increased too.

The main reason is that, in general, if the total amount of local task size increases then the negative impact of the local tasks on the processing of the Grid tasks expanded too. Because of the local task's appear are randomly, the balance and synchronization between workers may be broken. One of the chief differences between the UMR algorithm and DSA is the ability of the DSA to scheduling load chunks in light of the estimated CPU power for each worker.

In addition, the less important reason is, because of amount of Grid task size is predefined, so if the ratio Grid/Local increases then the total amount of workload (Grid task + Local task) will be decreased, which makes the makespan decrease too.

Consider a simple experiment. We use 5 workers named: iRMX, Kuenning, Bousquet, Soucy and Browne with the configuration described in the Table 4.3. In order to shown the negative impact of the local tasks on DSA and UMR algorithms, we intensified the arrival rate of local tasks on worker iRMX by 6 times more than any other workers. As a result, this worker become the weakest worker with respect to the available CPU power for the Grid tasks. Unlike our DSA algorithm, the UMR does not recognize this fact as it assumes that the iRMX continually offers all of its capacity to the Grid tasks. Therefore, the UMR mistakenly keeps sending the bigger chunks of workload to the iRMX, which leads to performance deterioration.

Table 4.4: The result of UMR algorithm under the impact of local tasks

Worker	**Total size of Grid task (flop)**	**Total size of local task (flop)**	**Finished time (s)**
iRMX	156250	110648	2652 (makespan)
Kuenning	125000	8187	1665
Bousquet	125000	8301	1660
Soucy	46875	7991	1828
Browne	46875	10230	1903

Table 4.5: The result of DSA algorithm under the impact of local tasks

Worker	**Total size of Grid task (flop)**	**Total size of local task (flop)**	**Finished time (s)**
iRMX	103113	69586	1727
Kuenning	153696	10091	2047
Bousquet	155642	10809	2080 (makespan)
Soucy	44747	8792	1858
Browne	42801	9743	1878

The other parameters are:

- The total workload: 500 (Kflops).
- The number of round: 50.
- Computation and communication latencies: 0.1 (second).
- Average frequency of local task is 2 (at iRMX) and 0.33 (at the remains workers).

The results of UMR and DSA algorithm are described in Table 4.4 and Table 4.5. As shown in the Table 4.6, it can be observed that the main factor cause DSA run faster than UMR is not the decrease of total size of local task (24.6%), but the decrease of finished time difference (64.4%). In case of UMR, the finished time difference between the first worker (Bousquet) and the last worker (iRMX) is: 2652-1660 = 992 (second). With the DSA, this difference is counted as: 2080 (Bousquet) - 1727 (iRMX) = 353 (second). This result illustrate the ability of DSA to estimate the size of local task on the workers, base on that, DSA self-turning the scheduling to adapt to the context. Moreover, by comparison between estimated value of CPU power and the factual value (Table 4.3), it can be seen that the prediction of the DSA is not quite exactly. The next section illustrates the difference between prediction ability of 2PP and DSA algorithms.

Table 4.6: The comparison between DSA and UMR algorithms

Factor	**UMR**	**DSA**	**The difference**
Time distance between the first worker and the last worker	992	353	64.4(%)
Makespan	2652	2080	21.6(%)
Total size of local tasks	144298	109023	24.6(%)

Table 4.7: The makespan of proposed algorithm DSA in comparison with UMR in experiment 1

Makespan of UMR (100s)	**Makespan of DSA (100s)**	**Size ratio: Grid task/Local task**
2769	2456	1.75
2460	2269	2.7
2339	2035	3.49
2340	1936	4.44
2259	1785	7.86
2112	1739	8.79
1969	1727	9.71
1950	1721	11.2
1873	1710	12.82

Table 4.8: The makespan of proposed algorithm DSA in comparison with UMR in experiment 2

Makespan of UMR (100s)	**Makespan of DSA (100s)**	**Size ratio: Grid task/Local task**
2972	2679	1.34
2923	2455	1.69
2769	2259	2.3
2516	2057	3.2
2341	1894	4.61
2192	1774	7.57
1973	1693	14.38

Table 4.9: The makespan of proposed algorithm DSA in comparison with UMR in experiment 3

Makespan of UMR (100s)	**Makespan of DSA (100s)**	**Size ratio: Grid task/Local task**
2456	2268	1.69
2205	2012	2.3
2039	1832	3.19
1899	1724	4.65
1828	1786	5.88
1804	1672	6.59
1781	1528	7.66
1763	1673	8.52
1748	1597	9.11
1725	1588	10.9

Table 4.10: The makespan of proposed algorithm DSA in comparison with UMR in experiment 4

Makespan of UMR (100s)	**Makespan of DSA (100s)**	**Size ratio: Grid task/Local task**
2830	2613	1.32
2747	2667	2.25
2521	2274	3.16
2414	2149	4.5
2237	2092	5.73
2042	1892	6.36
1937	1784	7.47

Table 4.11: The makespan of proposed algorithm DSA in comparison with UMR in experiment 5

Makespan of UMR (100s)	**Makespan of DSA (100s)**	**Size ratio: Grid task/Local task**
1735	1395	2.12
1532	1297	2.78
1428	1194	3.94
1433	1153	4.39
1259	1039	5.06
1116	1019	5.9
1184	1083	6.86
1262	1070	7.42
1087	1060	7.96

Table 4.12: The makespan of proposed algorithm DSA in comparison with UMR in experiment 6

Makespan of UMR (100s)	Makespan of DSA (100s)	Size ratio: Grid task/Local task
2532	2116	1.6
2402	2023	1.94
2302	2058	2.38
2210	1816	3.33
2154	1789	4.03
2099	1743	5.27
1961	1521	6.89
1845	1517	8.37
1625	1427	10.35

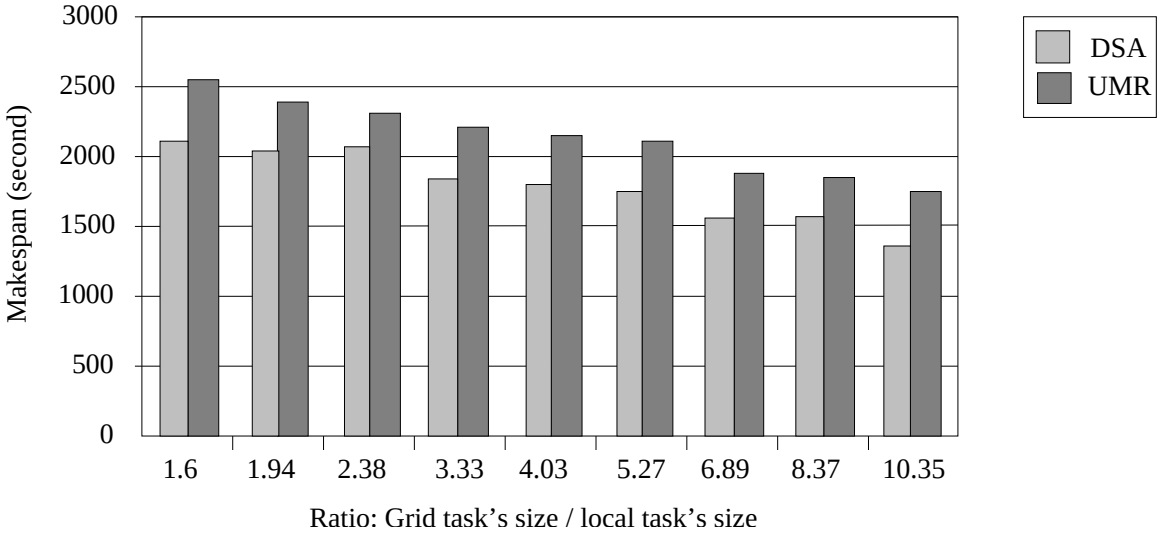


Figure 4.6: DSA vs. UMR. Number of workers =90

4.7.3 Comparison the DSA algorithm with the 2PP algorithm

Effect of local task on the makespan

We compare the performance of our scheduling algorithm DSA with algorithm 2PP by using the follow experiments configuration:

- Number of workers 20.
- The average power of worker: from 20 to 60 (Mflops/second).
- The average bandwidth: 50 (Mbyte/second).
- Computation and communication latencies: 0.1 (second).
- The ratio of Grid task's size to local task's size as described in Table 4.13.

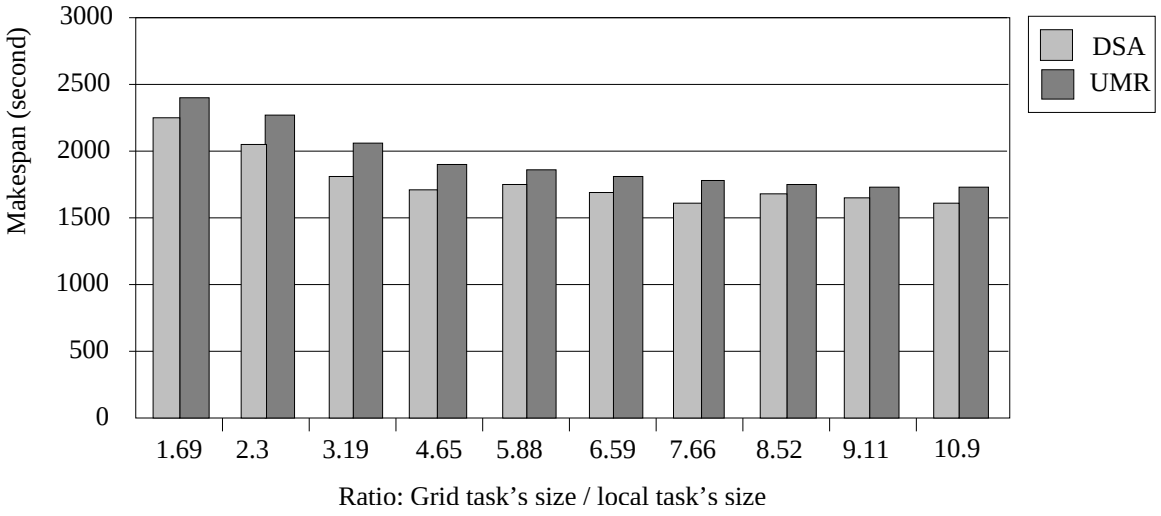


Figure 4.7: DSA vs. UMR. Number of workers =20

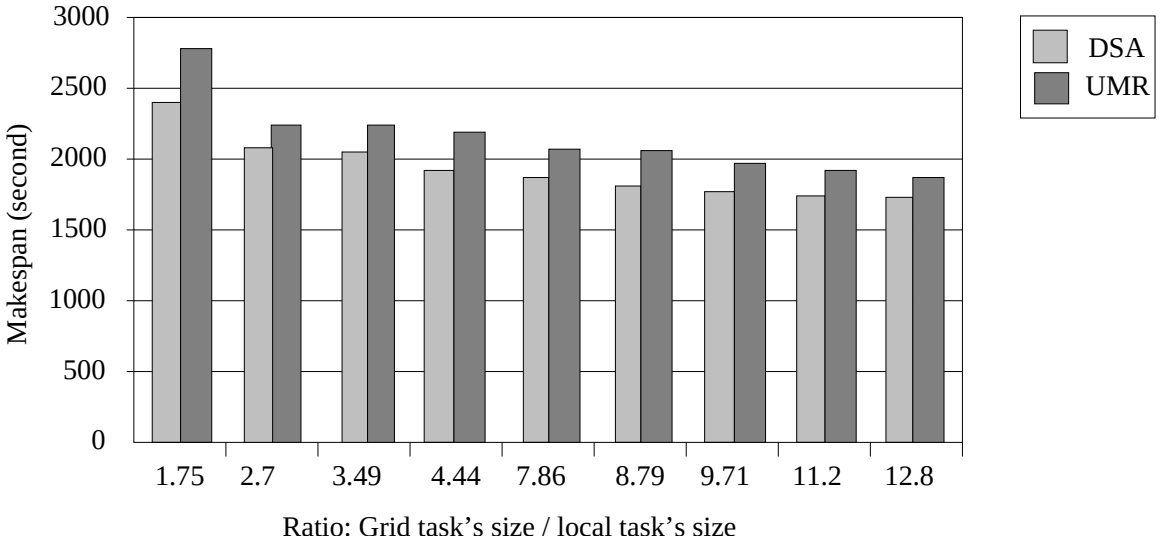


Figure 4.8: DSA vs. UMR. Number of workers =10

Figure 4.9 and Table 4.13 depict the comparison between the makespan of the proposed dynamic scheduling algorithms DSA and 2PP. From these results we can make the following remarks:

- With the low arrival rate, algorithm DSA is faster than 2PP. However, when the arrival rate run over a certain threshold (about 0.5 tasks/second in our experiment), 2PP outperforms DSA. 2PP algorithm based on 2PP prediction policy, it observer a long serious of history data point before extract the conclusion about the future value. In case of the arrival rate less than a certain threshold (0.5 tasks/second in our experiment), there are not enough history data points for 2PP to obtain an exactly value. Meanwhile, DSA relies on the prediction strategy Mixed Tendency-based which only need a short history data for estimating. Therefore, 2PP outperforms DSA only if it has enough history data points in order to find out the rule which control the varying of CPU Utilization.

Table 4.13: The makespan of algorithm DSA in comparison with 2PP, number of workers is 20

Arrival rate of the local task (task/second)	**Makespan of DSA (100s)**	**Makespan of 2PP (100s)**	**Size ratio: Grid task/Local task**
3.33	57.62	39.13	1.6
1.11	47.13	29.1	1.94
0.63	21.34	18.76	2.38
0.29	6.37	8.21	3.33

- The makespan deviation between 2PP and DSA increase proportion with the arrival rate of the local task. This mean the algorithm 2PP has the better adaptive capacity than DSA.

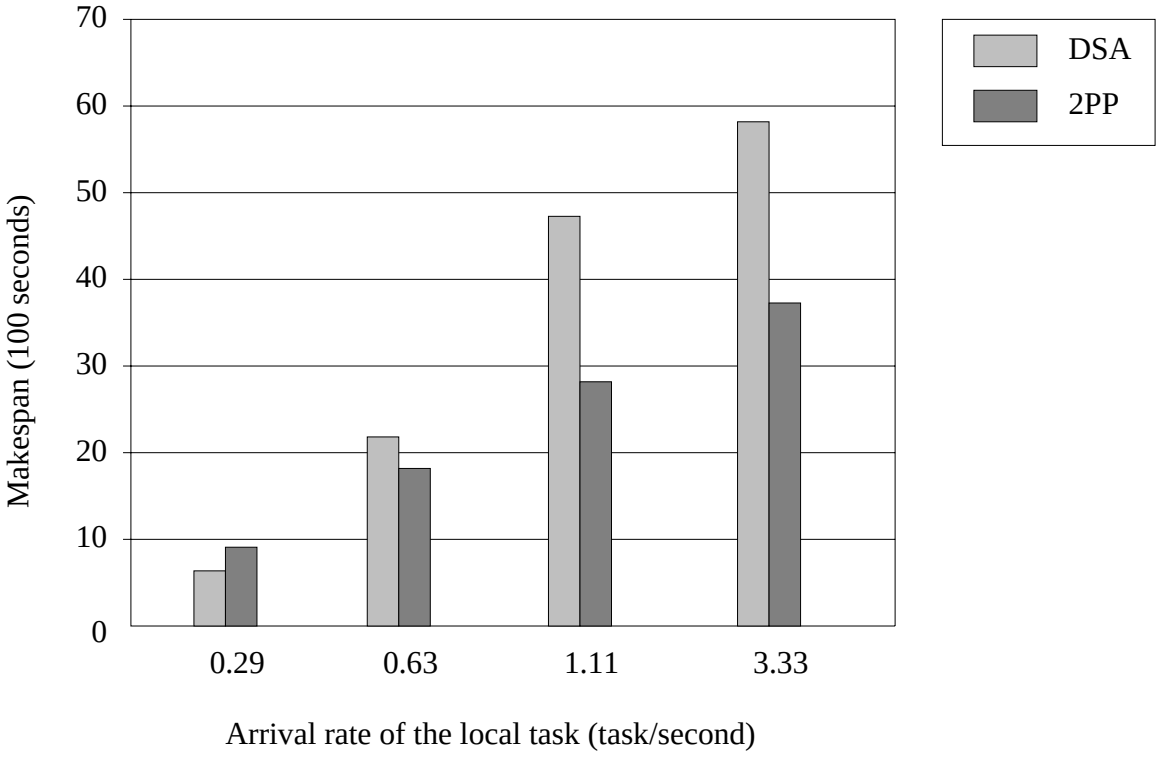


Figure 4.9: 2PP vs. DSA

4.8 Summary

In this chapter we address the issue of divisible load in Non-dedicate distributed environment. We develop a computation model to represent worker's activities with respect to processing local and external Grid tasks. The proposed model help estimate the computing power of a worker under the fluctuation of number of local and Grid applications in

the system. Based on this computation model we propose a new strategy for predicting the computing power of processors.

Proposed dynamic scheduling algorithm incorporates the performance model and the prediction method into the static algorithm MRRS that is mentioned in the last Chapter, which is originally a static scheduling algorithm. An alternative way is given by applying an existing prediction algorithm, Mixed Tendency-Based Prediction, in developing a dynamic scheduling algorithm. At last, we describe the experiments for comparing between proposed dynamic scheduling algorithms as well as for comparing the proposed algorithms with the existing static scheduling algorithm.

Finally, we conduct the experiments to verify the proposed scheduling algorithm. The results of our experiments demonstrate how our algorithm is adaptive to the inherit dynamicity of a heterogeneous, non-dedicated environment such as the Grid.

Chapter 5

Conclusion

Numerous studies in the literature have been targeting the problem of scheduling divisible loads in the distributed environment. However, such algorithms have a number of shortcomings such as the lack of an efficient resource selection strategy, or the assumption that computational resources at workers are dedicated. These constraints limit the utility of such algorithms and make them impractical for a dynamic computing platform such as the Grid.

The purpose of our research is to develop an efficient multi-round scheduling algorithm for non-dedicated environments such as Grids. Two kinds of studied platforms are Dedicated and Non-Dedicated platform and the corresponding Static and Dynamic algorithms are proposed for each of them. First of all, Chapter 2 provides an overall picture of divisible load scheduling problem and the divisible applications in distributed environments. It also describes the previous scheduling methods and their shortcomings.

The content of next chapters can be summarized as follow:

- Chapter 2 describes the general definition of the scheduling problem, as well as the taxonomy of workload and corresponding scheduling approaches. The main subject of this thesis, divisible load problem and its complexity, are discussed. Divisible Load Theory (DLT), the theoretical foundation of divisible scheduling algorithms, is given.

- In Chapter 3 we propose a new scheduling algorithm, MRRS (inspired by an existing algorithm called UMR), which is better and more realistic. MRRS is superior to the previous algorithms with respect to two aspects. First, MRRS factors in all platform parameters such as bandwidth capacity and all types of latencies (computation and communication) which renders the MRRS a more realistic model. Second, the MRRS is equipped with a worker selection policy that finds out the best workers. MRRS, to the best of our knowledge, is the first divisible load scheduling algorithm that addresses the resource selection problem. Having such policy is indispensable in large computing platform such as the Grid, where thousands of workers are accessible but the best subset must be chosen. We, theoretically and experimentally, show that MRRS is superior to previous algorithms such as UMR and LP, specifically in a WAN computing platform such as the Grid.

- Chapter 4 addresses the issue of divisible load scheduling in Non-Dedicated distributed environment. We develop a computation model to represent worker's activities with respect to local and external Grid tasks. The proposed model helps

estimate the computing power of a worker under the fluctuation of number of local and Grid applications in the system. Based on this computation model we propose a new strategy for predicting the computing power of processors. The proposed dynamic scheduling algorithm incorporates the performance model and the prediction method into the static algorithm MRRS that is mentioned in the chapter 3, which is originally a static scheduling algorithm. An alternative way is given by applying an existing prediction algorithm, Mixed Tendency-Based Prediction, in developing a dynamic scheduling algorithm DSA. Finally, we describe the experiments for comparing between proposed dynamic scheduling algorithms as well as for comparing the proposed algorithms with the existing static scheduling algorithm.

However, apart from contributions as mentioned above, the dissertation has also the following disadvantages:

- The lower bound Ω_V in the algorithm OSS is not very "strong". The experiments do not show the advantage of Branch and Bound OSS in comparison with the Greedy algorithm.

- The strategy 2PP, as a CPU utilization prediction, should be verified in the real Grid environments rather than a simulation tool as SIMGRID. Although there are many experiments under different configurations have been conducted, but they can not embrace all the respects and events of the real distributed environments.

- On the practice side, this dissertation does not present the experiment results enough. For instance, the dissertation lacks the experiments for comparison between the resources selection strategy OSS with Greedy strategy. Some of proposed algorithms are not fully explained, such as the case of algorithm DSA in Chapter 4.

This research can be presumed as the first study concerning the divisible scheduling problem in Grid environments. There are many areas in which the study presented in this dissertation can be extended.

- In the Divisible Load Theory, as well as in the static and dynamic computation models presented in this dissertation, the relation between size of workload and the computation time is linear. However, in fact there are many kinds of real issues where the above relations are non-linear, therefore we can obtain more effective schedule for a particular problem by developing the computation model suitable for that problem.

- To overcome the drawback of the selection strategy mentioned above, the "strong" lower bound function should be further studied. It can be seen that, similar to the other kind of workload, the heuristic may be play an important role in selection the best subset of workers like the traditional methods.

- To overcome the drawback of the experiments tools, as mentioned above, the implementation of the proposed dynamic scheduling algorithm in a real computation Grid would be worth to be executed.

Bibliography

[1] I. Foster and C. Kesselman: "Grid2: blueprint for a new computing infrastructure", (Second ed.) San Francisco, Morgan Kaufmann Publisher (2003).

[2] Y. Zhang, Y. Inoguchi, and H. Shen: "A Dynamic Task Scheduling Algorithm for Grid Computing System", Proc. of the Second International Symposium on Parallel and Distributed Processing and Applications (ISPA'2004) (Dec. 2004).

[3] P. Chretienne, E. G. Coffman Jr., J. K. Lenstra and Z. Liu: "Scheduling theory and its applications", John Wiley & Sons (1995).

[4] V. Bharadwaj, D.Ghose, V.Mani, and T. G. Robertazzi: "Scheduling divisible loads in parallel and distributed systems", IEEE Computer Society Press, 1996.

[5] V. Bharadwaj, D.Ghose, and T. G. Robertazzi: "A New Paradigm for Load Scheduling in Distributed Systems", Special Issue of Cluster Computing on Divisible Load Scheduling, vol.6, no.1, pp.7-17 (Jan. 2003)

[6] T. G. Robertazzi: "Ten reasons to use divisible load theory", IEEE Computer, vol 36, no.5, pp. 6368, (2003).

[7] T.G. Robertazzi: "Divisible Load Scheduling". Available online at: http://www.ece.sunysb.edu/$\tilde{t}$om/dlt.html.

[8] J. Sohn, T. G. Robertazzi, and S. Luryi: "Optimizing computing costs using divisible load analysis", IEEE Trans. on parallel and distributed systems, vol. 9, no.3, pp. 225234, (March 1998).

[9] C. Lee and M. Hamdi: "Parallel Image Processing Applications on a Network of Workstations", Parallel Computing, vol. 21, pp. 137-160 (1995).

[10] D. Altilar and Y. Paker: "An optimal scheduling algorithm for parallel video processing", Proc. of IEEE Int. conference on Multimedia Computing and Systems, IEEE Computer Society Press (1998).

[11] D. Altilar and Y. Paker: "Optimal scheduling algorithms for communication constrained parallel processing", Euro-Par 2002, LNCS 2400, pp. 197206, Springer Verlag (2002).

[12] M. Drozdowski: "Selected problems of scheduling tasks in multiprocessor computing systems", PhD thesis, Instytut Informatyki Politechnika Poznanska, Poznan (1997).

[13] J.Blazewicz, M.Drozdowski, and M.Markiewicz: "Divisible task scheduling - concept and verification", Parallel Computing, vol.25, no.1, pp.87-98 (1999).

[14] R. Y. Wang, A. Krishnamurthy, R. P. Martin, T. E. Anderson, and D. E. Culler: "Modeling communication pipeline latency", Proc. of Measurement and Modeling of Computer Systems (SIGMETRICS98), pp. 2232, ACM Press (1998).

[15] O. Beaumont, H. Casanova, A. Legrand, Y. Robert, and Y. Yang "Scheduling Divisible Loads on Star and Tree Networks: Results and Open Problems", IEEE Trans. on Parallel and Distributed Systems, vol. 16, no. 3, pp. 207-218 (2005).

[16] O. Beaumont, A. Legrand, and Y. Robert: "Scheduling divisible workloads on heterogeneous platform", Parallel Computing, vol.29, no.9, pp. 1121-1152 (2003).

[17] Y. Yang, K.V. Raart, and H. Casanova: "Multiround algorithms for scheduling divisible loads", IEEE Transaction on Parallel and Distributed Systems, vol.16, no.11, pp.1092-1104 (2005).

[18] Y. Yang and H. Casanova: "UMR: a multi-round algorithm for scheduling divisible workloads", Proc. of the International Parallel and Distributed Processing Symposium (IPDPS'03), Nice, France (2003).

[19] Y. Yang and H. Casanova: "RUMR: Robust Scheduling for Divisible Workloads", Proc. of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03) Seattle, Washington, USA (2003).

[20] P. Brucker: "Scheduling algorithms", (Second ed.) Springer (1997).

[21] T. D. Braun, H. J. Siegel, and N. Beck: " Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems", Parallel and Distributed Computing (2001).

[22] H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman: "Heuristics for Scheduling Parameter Sweep Applications in Grid environments", Proc. of 9th Heterogeneous Computing Workshop (HCW'2000), Mexico, pp. 349363 (2000).

[23] N. Fujimoto and K.Hagihara: "Near-Optimal Dynamic Task Scheduling of Precedence Constrained Coarse-Grained Tasks onto a Computational Grid", Proc. of the IEEE International Symposium on Parallel and Distributed Computing (ISPDC'03), pp. 80-87 (2003).

[24] Y. Chung and V. K. Prasanna: "Parallelizing Image Feature Extraction on Coarse-Grain Machines ", IEEE Trans. of Pattern Analysis and Machine Intelligence, vol. 20, no. 12, pp. 1389-1394 (1998).

[25] A. Legrand, Y. Yang, and H. Casanova: "NP-Completeness of the Divisible Load Scheduling Problem on Heterogeneous Star Platforms with Affine Costs", Technical report CS205-0825 (2005), available online at http://www-cse.ucsd.edu/Dienst/UI/2.0/Describe/ncstrl.ucsd_cse/CS2005-0818 .

[26] A. L. Rosenberg: "Sharing PartitionableWorkloads in Heterogeneous NOWs: Greedier Is Not Better", Proc. of the 3rd IEEE International Conference on Cluster Computing (Cluster 2001), pages 124131 (2001).

[27] D. P. Bertsekas: "Constrained optimization and lagrange multiplier methods", Belmont, Mass. : Athena Scientific, (1996).

[28] S. Martello and P. Toth: "Knapsack problems : algorithms and computer implementations", Chichester, West Sussex, England : Wiley (1990).

[29] Simulation toolkit for the study of scheduling algorithms for distributed application, SimGrid project, available online at http://simgrid.gforge.inria.fr

[30] H.Casanova: "Simgrid: a toolkit for the simulation of application scheduling", Proc. of the IEEE International Symposium on Cluster Computing and the Grid (CCGrid'01), Australia, pp.430-437 (2001).

[31] H. Casanova, A. Legrand ,and L. Marchal: "Scheduling Distributed Applications: the SimGrid Simulation Framework", Proc. of the third IEEE International Symposium on Cluster Computing and the Grid (CCGrid'03) (2003).

[32] O. Beaumont, L. Carter, J. Ferrante, A. Legrand, and Y. Robert: "Bandwidth-centric allocation on independent tasks on heterogeneous platform", Proc. of the International Parallel and Distributed Processing Symposium (IPDPS 2002), USA, pp.67-72 (2002).

[33] B.Hamidzadeh, L.Y. Kit, ans D.J.Lilja: "Dynamic Task Scheduling Using Online Optimization", IEEE Transaction on Parallel and Distributed Systems, vol.11 (Nov. 2000).

[34] H.Casanova, A. Legrand, D.Zagorodnov, and F. Berman: "Heuristic for Scheduling Parameter Sweep Applications in Grid Environments", Proc. of 9th Heterogeneous Computing Workshop (HCW) (May 2000).

[35] T. L. Nguyen, S. Elnaffar, T. Katayama, and T. B. Ho: "A scheduling method for divisible workload problem in Grid environments", Proc. of 6th International Conference on Parallel and Distributed Computing (PDCAT 05), Dalian China, pp.513-517 (2005).

[36] T.L. Nguyen, S. Elnaffar, T. Katayama, and T. B. Ho: "Grid Scheduling using 2-Phase Prediction (2PP) of CPU Power", Proc. of the IEEE Innovations in Information Technology (IIT06), IEEE Communication Society Press, IEEE Catalogue Number 06EX1543C, ISBN 1-4244-0674-9, Library of Congress 2006933185, Dubai, UAE (2006).

[37] L. Yang, J. Schopf, and I. Foster: "Conservative Scheduling: Using Predicted Variance to Improve Scheduling Decision in Dynamic Environments", SuperComputing 2003, Phoenix, Arizona USA (Nov. 2003).

[38] L. Yang, J. Schopf and I. Foster: "Homeostatic and Tendency-based CPU Load Predictions", International Parallel and Distributed Processing Symposium (IPDPS'03) Nice, France (Apr. 2003).

[39] A. Papoulis and S. U. Pillai: "Probbility, Random Variables and Stochastic Processes", 4th edition McGraw-Hill (2002).

[40] T.L. Nguyen, S. Elnaffar, T. Katayama, and T.B. Ho: "UMR2: A Better and More Realistic Scheduling Algorithm for the Grid", Proc. of the International Conference on Parallel and Distributed Computing and Systems (PDCS'06) USA, pp. 432-437, ISBN: 0-88986-638-4 (2006).

[41] T.L. Nguyen, S. Elnaffar, T. Katayama, and T.B. Ho: "MRRS: A More Efficient Algorithm for Scheduling Divisible Loads of Grid Applications", Proc. of the 2006 IEEE/ACM International Conference on Signal-Image Technology and Internet-based Systems (SITIS), Tuynidia (Dec. 2006) (to appear).

Publications

[1] S. Elnaffar and T.L. Nguyen: "A Methodology for Dynamic Scheduling of Divisible Workloads in Grid Environments", Proc. of the 5th WSEAS Int. Conf. on Simulation, Modeling and Optimization (SMO '05), Greece (Aug. 2005).

[2] S. Elnaffar and T.L. Nguyen: "Enabling Dynamic Scheduling in Computational Grids by Predicting CPU Utilization", WSEAS Trans. on Communications Issue 12, vol.4, ISSN:1109-2742, pp.1419-1426 (Dec. 2005).

[3] T.L. Nguyen, S. Elnaffar, T. Katayama, and T.B. Ho: "A Scheduling Method for Divisible Workload Problem in Grid Environment". Proc. of IEEE Computer Society Proceedings of the International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT '05), pp.513-517, China (2005), ISBN: 0-7695-2405-2.

[4] T.L. Nguyen, S. Elnaffar, T. Katayama, and T.B. Ho: "UMR2: A Better and More Realistic Scheduling Algorithm for the Grid", Proc. of the International Conference on Parallel and Distributed Computing and Systems (PDCS'06), pp. 432-437, ISBN: 0-88986-638-4, USA(2006).

[5] T.L. Nguyen, S. Elnaffar, T. Katayama, and T.B. Ho: "Grid Scheduling using 2-Phase Prediction (2PP) of CPU Power", Proc. of the IEEE Innovations in Information Technology (IIT 2006), IEEE Communication Society Press, IEEE Catalogue Number 06EX1543C, ISBN 1-4244-0674-9, Library of Congress 2006933185, Dubai, UAE (2006)

[6] T.L. Nguyen, S. Elnaffar, T. Katayama, and T.B. Ho: "Grid Scheduling using 2-Phase Prediction (2PP) of CPU Power", short listed by IIT 2006 for Special Issue of Journal of Communications (JCM), Academy Publishers.

[7] T.L. Nguyen, S. Elnaffar, T. Katayama, and T.B. Ho: " MRRS: A More Efficient Algorithm for Scheduling Divisible Loads of Grid Applications", Proc. of the 2006 IEEE/ACM International Conference on SIGNAL-IMAGE TECHNOLOGY and INTERNET- BASED SYSTEMS (SITIS), Tuynidia (Dec. 2006), pp. 280-290.

[8] T.L. Nguyen, S. Elnaffar, T. Katayama, and T.B. Ho: " A More Efficient Algorithm for Scheduling Divisible Workloads in Distributed Computing Platforms", submitted to IEICE Trans. on Information & System.