Universitat Autònoma de Barcelona

Effective Resource Management for Master-Worker Applications in Opportunistic Environments

Departament d'Informàtica
Unitat d'Arquitectura d'Ordinadors
i Sistemes Operatius

Thesis submitted by Elisa Heymann Pignolo in fulfillment of the requirements for the degree of *Doctor per la Universitat Autònoma de Barcelona*.

Barcelona (Spain), September 3rd 2001

Effective Resource Management for Master-Worker Applications in Opportunistic Environments

Thesis submitted by Elisa Heymann Pignolo in fulfillment of the requirements for the degree of *Doctor per la Universitat Autònoma de Barcelona*. This work has been developed in the Computer Science Department of the *Universitat Autònoma de Barcelona* and was advised by Dr. Miquel Ángel Senar Rosell,

Bellaterra September 3rd, 2001

Thesis Advisor

Miquel Ángel Senar Rosell

To Eduardo and our children, David and Ruth.
To my parents, Ezra and Sara

ACKNOWLEDGEMENTS

A lot of people have made this thesis possible. I wish to express my sincere gratitude to them all for being there, for working with me and for helping me.

First of all I would like to thank Miquel Ángel Senar for being my advisor throughout this work, for spending so much time and effort on it, and for his constant advice and never-ending encouragement.

Special thanks to Emilio Luque for all his support throughout the development of this work, for his valuable advice and for his opportune comments.

I would like to express my special gratitude to Miron Livny for his ideas regarding this work, as well as for his inestimable contribution to its development.

My deepest thanks go to Eduardo Cesar for being there at all times, for his support and affection. How could I have undertaken this work without everything he has done?

Thanks to Crisos Lopez who collaborated in the implementation of some of the programs related to this work.

I would also like to thank Anna Cortés, Tomàs Margallef, Dani Franco and Juan Carlos Moure for their encouragement and friendship.

I am grateful to the rest of my colleagues from the Computer Architecture and Operating Systems Group as well as to those who have passed through the Group during the *gestation* and preparation periods of this work.

I would also like to thank the members of the Condor Team at the University of Wisconsin-Madison, for their support with respect to my Condor doubts and problems, and for the good times I shared with them while I was working in Madison.

Special thanks to Barton Miller and his family for making us feel at home in Madison.

Last, but decidedly not least, I would like to thank my family for their support throughout this period, and especially my son David and my daughter Ruth for their understanding while mum was “at work”.

Contents

PREFACE

Continuos improvement in hardware and software technologies has led to an increased interest in distributed systems and their wider use in executing large-scale scientific and commercial applications. Distributed systems are often used in a non-dedicated way, i.e., the resources constituting the system are shared among different user applications. Additionally, opportunistic clusters of machines –a particular type of non-dedicated distributed systems– are becoming more popular. This is because, on the one hand, research and development is leading to the appearance of different environments and tools in order to facilitate the use of such systems, and, on the other hand, the number of computational resources has been growing tremendously. The results of this growth are large, heterogeneous and dynamic computer environments. This work focuses on the use of such opportunistic environments, characterized by harnessing idle times of machines for executing user jobs.

Management of the concurrent jobs constituting a parallel application is an integral part of such non-dedicated systems. In non-dedicated opportunistic environments, the resource manager's goal is to provide both a reasonable execution time (as users are interested in having their job finished as soon as possible), and good efficiency, i.e., good resource usage, which is the main goal of the system in order to obtain high throughput.

The development of effective resource management for parallel applications running on opportunistic systems involves a great number of issues. In particular, this work deals with three of them:

- Determining and allocating the number of machines, from the pool of machines belonging to the opportunistic system, needed for executing an application obtaining both a good execution time and a good efficiency. These machines will constitute a variable partition of the whole pool of resources that will be allocated to the application in a dynamic way.

- Scheduling application tasks to the assigned computational resources (partition).
- Reducing the negative effects produced on an application when a machine belonging to a non-dedicated environment, and allocated to the partition given to the application, is reclaimed by its owner, and should therefore be released by the task running on it.

Throughout this work master-worker applications have been considered, because many real problems naturally fit into this parallel programming paradigm. In these applications, there is a master that sends tasks to workers and collects the results. This process is repeated over a number of cycles or iterations until several bags of tasks have been executed.

In order to assign tasks belonging to a master-worker application to machines belonging to an opportunistic environment, dynamic tasks scheduling techniques were used. This was decided upon not only because of the changing nature of the executing environment, but also because we do not have any *a priori* information about the execution time of the tasks to be executed by the workers. Execution times of the tasks are not known *a priori* and may change from one cycle to another. Considering both the previous application behavior and execution environment, static task scheduling techniques are not suitable.

The proposed scheduling policy presented in this work does not need information about the execution time of the tasks. It works by sorting tasks according to their average execution time, using the information obtained at runtime, that is, by using information related to the execution of previous cycles or iterations of the master-worker application. This policy was first evaluated by simulation, taking into account different workloads, in order to cover most types of master-worker applications. The goal of this simulation was to understand how different scheduling policies behave in the best case, i.e. when no machine is reclaimed by its owners and all machines are homogeneous. The simulation results showed that the proposed policy exhibits a similar behavior with respect to other policies requiring information in advance about the execution time of the tasks.

From the scheduling policy simulation we derived the existence of the *ideal interval*, which corresponds to the interval comprised between the minimum and maximum number of machines for executing the application that obtain a good trade-off between execution time and efficiency. At a later stage of this discussion we will propose an algorithm for dynamically adjusting the number of machines, for executing any master-worker application, to a number of machines belonging to the *ideal interval*.

Two different self-adjusting strategies were designed and evaluated. One of the strategies used an empirical table in its decision-making process, while the second one was able to determine the desirable number of machines in a completely dynamic way. This strategy was implemented and evaluated on both homogeneous and heterogeneous environments, with a master-worker thinning application running on a real opportunistic environment. Resources were managed through the services provided by the *Condor* batch system, and the application was written with *MW* (Master-Worker tool), a *C++* library especially suited to the easy development of master-worker applications.

In an opportunistic environment machines can join and leave the computation as they are released or reclaimed by their owners. When a machine is reclaimed, the job running on that machine must be stopped and vacated. If this job belongs to a parallel application, the whole performance will be negatively affected. We evaluate this impact both on the efficiency and execution time, and then, in order to alleviate it, propose strategies based on using extra machines and task replication. These strategies were first evaluated by simulation and then implemented and tested in a real environment.

This thesis is organized as follows:

- The first chapter gives an overview of parallel programming paradigms and architecture models, introduces the scheduling problem in distributed systems and reviews the solutions found in the literature. Finally, it presents some example systems.
- The second chapter describes the master-worker programming model which corresponds to the programming model considered throughout this work. Later, it shows the opportunistic execution environment model considered in this work, and the systems and tools used in the experimentation phases of this work: *Condor* as resource management system, and the *MW* programming environment. After that, it introduces the performance model considered and explains the main challenges of executing master-worker applications on opportunistic environments, remarking the challenges that will be covered in this work.
- In chapter three the proposed dynamic scheduling algorithm is introduced, and evaluated by simulation. It is compared to other scheduling policies, which differ in the amount of information that they handle concerning the application. During the simulations, execution time and efficiency are always measured by considering these main factors: workload, number of tasks per cycle, number of cycles and the variation per cycle of the task execution times.
- Chapter four contains the proposal and implementation of the self-adjusting algorithm for dynamically adapting the number of workers needed by a master-worker application. In order to evaluate the strategy, a master-worker thinning application is implemented on a real opportunistic environment. The self-adjusting algorithm is first evaluated on an opportunistic environment composed of homogeneous machines, and is then modified in order to support heterogeneous environments.

- Chapter five first evaluates the impact of losing a machine that is participating in the computation of a parallel application. Following this, we propose strategies for alleviating such an impact. These strategies are based on task replication and on having extra machines. This chapter then evaluates them by simulation, and then subsequently comments on their scalability. Finally, mention is made of the results obtained when these strategies are implemented on a real opportunistic environment and a master-worker *PovRay* application is executed.
- Chapter six summarizes the main conclusions derived from this thesis, outlining, in addition, current and future work.
- Finally, the complete bibliography is provided. Complementary result tables and the master-worker thinning application used to test the proposed main ideas are included in the appendices.

Chapter 1

Introduction

Abstract

This chapter first gives an overview of parallel programming paradigms and architecture models. Subsequently, the main scheduling strategies for distributed systems are surveyed. Finally, examples of existing schedulers are briefly described for both intranets and internets.

1.1 Introduction

Parallel applications consist of one or more tasks that may communicate and cooperate to form a single application. Scheduling these applications involves a number of activities [Ber99]:

1. Selecting a set of resources on which to schedule the application tasks.
2. Assigning application tasks to resources.
3. Distributing data or place data and computation.
4. Ordering tasks on compute resources.
5. Ordering communication between tasks.

In the literature, item 1 is referred as *resource location* or *resource discovery*. Item 2 may be called *mapping*, *partitioning* or *placement*. For data parallel programs, *load-balancing* is often the scheduling policy chosen (item 3). Items 1 through 3 are generally termed *mapping* and focus on the allocation of computation and data *in space*; items 4 and 5, generally termed *scheduling*, deal with the allocation of computation and communication *over time*. Depending on both the adopted architecture model and the adopted programming model, each one of the five items is more or less relevant.

A *scheduling model* consists of a *scheduling policy*, a *program model, an architecture model* and a *performance model*. The scheduling policy corresponds to the set of rules for producing schedules. The program and the architecture model abstract the sets of programs to be scheduled and the underlying system, respectively. The performance model abstracts the behavior of the program on the system considered for the purpose of evaluating the candidate schedules.

In the following sections, we will briefly describe the possibilities for both program and architecture models, and will then discuss the alternatives for scheduling in distributed systems. Finally, we will comment on some example systems.

1.2 Parallel Programming Models and Paradigms

Each programming paradigm is a class of algorithms that has the same control structure [Han93], and that can be implemented using a parallel programming generic model. In the following sections we will introduce the most important parallel programming models and parallel programming paradigms in use. For the programming models, we will comment on how they solve the distribution of code and the interconnection between execution units or tasks. For the programming paradigms, we will describe their basic structure.

1.2.1 Parallel Programming Generic Models

Generic models for parallel programming are divided into the following main classes [VA+94], [Fos95]:

- *Message Passing* [AS91]: This is a widely used model. In this model, programmers organize their programs as a collection of processes with private local variables and the ability to send and receive data between processes by passing messages. PVM [GB+94] and MPI [GL+96] constitute examples of libraries supporting the message passing programming model.
- *Shared Memory* [Ben90]: In this model, programmers view their programs as a collection of processes accessing local variables and a central pool of shared variables. Each process accesses the shared data by asynchronously reading from or writing to shared variables. As more than one process may access the same shared variables at one

time, mechanisms to resolve mutual-exclusion problems need to be provided, such as locks or semaphores.

- *Data Parallelism* [HS91]: This model is suitable for programs that perform the same operation on different data elements. A typical data parallel program consists of a list of certain operations to be carried out on a data structure. From here, each operation on each data element can be thought of as an independent task. High Performance Fortran has been widely used for implementing this type of parallelism [CMZ94].

These parallel programming models can be used in order to implement parallel programming paradigms. In theory, both programming models, that is, message passing and shared memory, could serve to implement every paradigm, but the performance of each resulting combination will depend on the underlying execution model. In the following section we describe the most common paradigms.

1.2.2 Parallel Programming Paradigms

There are many high level abstractions that provide support for parallel paradigms. Programming paradigms are a class of algorithms that solve different problems but have the same control structure [Han93]. A programming skeleton corresponds to the instantiation of a specific parallel paradigm, and encapsulates the control and communication primitives of the application into a single abstraction.

In the literature there are many different paradigm classifications such as [Pri88], [Han93], [Wil95] and [Fox89] and [BK+93]. Although not all of these contemplate the same set of paradigms, they do not differ greatly. In [Buy99] there is a classification that includes the superset of the most habitual paradigms:

- *Master-Workers* (or Task-farming): This consists of two entities: the master and multiple workers. The master is responsible for decomposing the problem into small tasks, then distributes these tasks among the workers and finally collects the partial results in order to produce the final computation results. The workers receive a task, process it and send the result back to the master. Figure 1-1 presents a schematic representation of this paradigm. This is the paradigm considered throughout this work; therefore in the next chapter it is described in more detail.

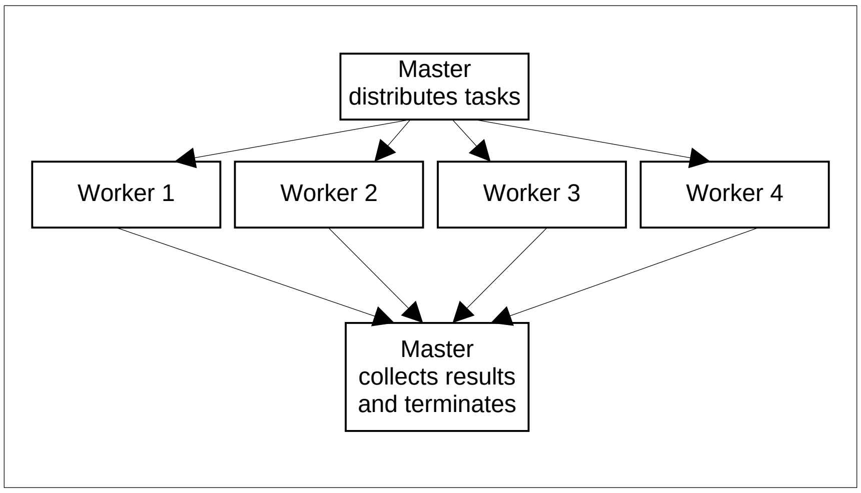


Figure 1-1. Basic master-worker structure.

- *Single Program Multiple Data* (SPMD): Each process executes the same piece of code, but with different data. This means splitting the application data among the available machines participating in the computation. This type of parallelism is also referred to as *geometric parallelism*, *domain decomposition* or *data parallelism*. Figure 1-2 presents a schematic representation of this paradigm.

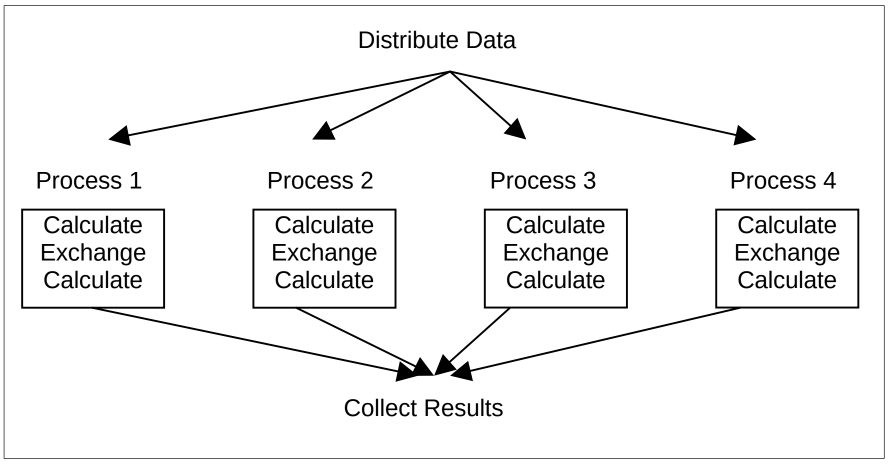


Figure 1-2. Basic structure of an SPMD program.

- *Data Pipelining*: This is based on a functional decomposition approach. Each process corresponds to a stage of the pipeline, and is responsible for a particular task. All the processes are executed concurrently and the data flows along the stages of the pipeline, as shown in Figure 1-3.

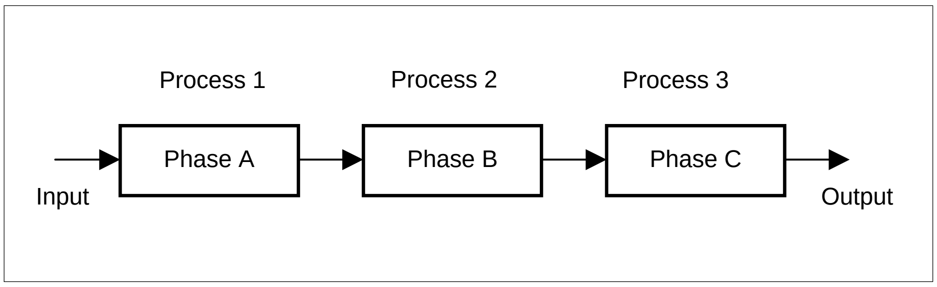


Figure 1-3**.** Data pipeline structure.

- *Divide and Conquer*: A problem is divided into several sub-problems. Each of these sub-problems is solved independently and their results are combined to produce the final result. If the sub-problems are smaller instances of the original problem, then a *recursive decomposition* is produced. Three generic operations are needed for the divide and conquer paradigm: split, compute and join, as shown in Figure 1-4.

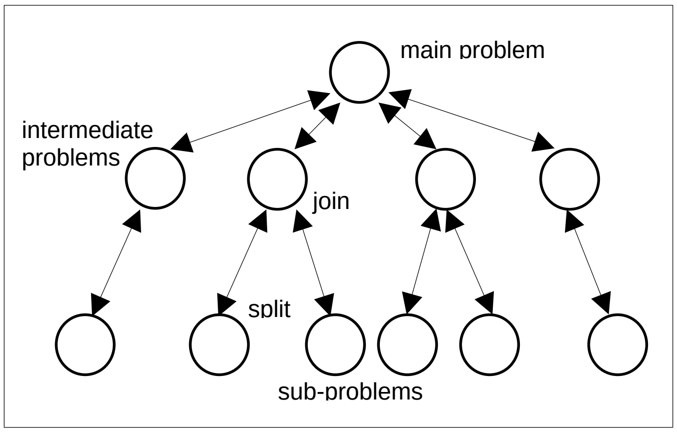


Figure 1-4. Divide and conquer example structure.

- *Speculative Parallelism*: This is used when it is quite difficult to obtain parallelism through any one of the previous paradigms. It includes several possibilities such as employing different algorithms for the same problem: the one providing the final solution first is then chosen.

  If the problem has complex data dependencies the problem can be executed in small parts, some speculation nevertheless being used to allow the parallelism.

Having introduced the different programming models and paradigms, we will now review the different possibilities with regard to the underlying systems.

1.3 Architecture Models

There are many ways in which parallel and distributed machines can be constructed. In this subsection, we present their classification based on control mechanisms and address-space organization [CPG99, Fos95].

- *Distributed Memory Multiple Instruction Multiple Data* (MIMD): In this model, each processor executes a separate set of instructions on its own local data. The memory is distributed throughout the processors, rather than being placed in a central location. A network connects processors (and their local memories). Processors exchange data between their memories when a remote variable value is required. Examples of this class of machine include the IBM SP, Thinking Machines CM5, Cray T3E, SGI Origin2000 and ASCI Red.

- *Shared-Memory MIMD* or *multiprocessors*: In this model all processors share access to a common memory. Users need not be concerned about the current place where the data is stored, as all processors access the same data space. Examples of this class of machine include the Silicon Graphics Challenge, the multiprocessor workstations, Cray J90, Cray T90, Digital Alpha and Tera-MTA.
- *Single Instruction Multiple Data* (SIMD): In this model, all processors execute the same instructions on a different piece of data. This class of parallel machines is suitable for problems presenting a high degree of regularity, such as image processing and certain numerical simulations.

From the perspective of a management model, we introduce four types of systems, of increasing scale and complexity as described in [FK99]:

- The *End System*: Individual end systems (processors, storage systems and other devices) are characterized by a small scale and a high decree of homogeneity and integration. They represent the simplest environment, where the operating system has absolute control over the resources of the computer. Resource management must handle process creation, operating system signal delivery and operating system scheduling.
- The *Cluster*: This consists of a collection of computers connected by a high-speed local area network. It is controlled by a single administrative entity that has complete control over each end system. Parallel process creation and scheduling are the main issues concerning resource management at this level.

- The *Intranet*: This comprises a large number of resources that nevertheless belong to a single organization, therefore there is no one single site for coordination. It introduces the additional issues of heterogeneity and geographical distribution. Resource management should in addition handle resource discovery, signal distribution networks and provide high throughput. Uniform access for computing resources can be provided by distributed queuing systems such as Condor [LLM88], Codine (recently renamed as the *Sun Grid Engine*) [SGE01] or Load Sharing Facility (LSF) [ZZ+93].
- The *Internet*: Internets lack centralized control and are geographically widely distributed (they may cross international borders, for example). They use a public network that has variable behavior depending on particular time conditions, such as load and routers. Resource management should add brokers, negotiation and trading issues. Many internet systems, usually referred as *Grids*, are nowadays under development, and probably the most well-known are Globus [FK97] and Legion [GW+94].

From all the systems described above, throughout this work we will consider distributed memory MIMD machines organized in an intranet and constituting an opportunistic environment for the users.

1.4 Scheduling Strategies in Distributed Environments

In non-dedicated distributed environments, several applications could be executed on the same set of available resources. Scheduling functionality can be divided into the external and internal [McL97], also referred as two-level scheduling [Fei94], or generally termed mapping and scheduling [Ber99]:

- *External Scheduling*: This is concerned with the assignment of applications to compute resources, i.e., with resource allocation.

- *Internal Scheduling*: This is involved with the assignment of tasks belonging to an individual computation to the machines assigned to that application, that is, with the use of the resources already allocated.

As stated in [FR+97], with regard to external scheduling, each job belonging to a parallel application is executed in a partition that consists of a number of processors. The size of such a partition may depend on the machines available, the application and the load of the machines. The size of the partition of a specific job may change during the job's lifetime. Different types of partitions can be considered:

- *Fixed*: The partition size is defined by the system administration and can be modified only by reboot.
- *Variable*: The partition size is determined at job submission time based on user request and system capabilities.
- *Adaptive*: The partition size is determined by the scheduler at the time the job is initiated, based on system load, machine availability and by taking user request into account.
- *Dynamic*: The partition size may change during the execution of a job, to reflect changing requirement and machine availability.

On the other hand, as [FR+97] explains, from the point of view of job flexibility, applications can be characterized as follows:

- *Rigid jobs*: The partition assigned to a job is specified external to the scheduler (by the user or the system administrator for example), and that number of machines is made available to the job throughout its execution.
- *Moldable jobs*: The partition assigned to a job is determined by the scheduler, and the job uses that number of machines throughout its execution.

- *Evolving jobs*: The job has different phases that require different number of machines, so the number of machines allocated may change during the execution of the job in response to job needs.
- *Malleable jobs*: The partition assigned to a job may change during the job's execution, as a result of the system giving it additional machines or requiring the job to release some of them.

Once a partition has been assigned to a parallel application, internal scheduling is performed in order to assign jobs composing the application to the machines belonging to the partition. Internal scheduling can be divided into static and dynamic.

Static Scheduling consists of assigning all application tasks to compute resources before execution begins. Once a task has been assigned to a machine, it will finish its execution on that machine, unless the machine in question fails. Static scheduling usually views the application either as a static graph with fixed and specified dependencies (commonly known as DAG: Directed Acyclic Graph),or as a static graph with fixed and specified interaction/communication patterns (commonly known as TIG: Task Interaction Graph) [HS+01c].

Dynamic Scheduling techniques assign tasks to compute resources as tasks are being created or as resources become available, i.e. when either a new task appears in the system or a machine becomes idle, then the decision of where to execute this task will be taken. Dynamic scheduling may also change the assignment of a task after it has begun its execution. These dynamic techniques can be divided into Load Balancing, Preemptive Scheduling and Non-Preemptive Scheduling. The strategies we now introduce are representative and in general, have been proposed in theory or in practice either for clusters of machines or MPP (massively parallel processors) systems.

- *Load Balancing* [SC+01]: This is based on the redistribution of load among the processors during execution time, so that each processor would have the same or nearly the same amount of work to do. This redistribution is performed by transferring load units from the heavily loaded processors to the lightly loaded processors with the aim of obtaining the highest possible execution speed [SC+01]. The load units that are redistributed are handled as independent elements, without there being any relation of dependence or affinity among them.

- *Preemptive Scheduling*: Depending on whether jobs may be relocated during their execution time, preemptive scheduling is divided into:
  - *Local Preemption*: Tasks composing a job may be preempted, but each task can only be resumed later on the same machine. This kind of preemption does not require any data movement between machines.
  - *Migratable Preemption*: Job tasks may be suspended on one machine and subsequently resumed on another. This implies the overhead of checkpointing a parallel task and restoring it on other machine.

  Examples of preemptive schedulers that can be implemented with or without migration are CO-scheduling [Ous82, Fei94] (also known as *Gang Scheduling*) and *Handoff Scheduling* [Bla90, TSS88].

  CO-scheduling tries to simultaneously schedule tasks that work in close coordination with each other during execution. In doing so, it attempts to avoid trashing that could occur if such tasks were not run together.

  Handoff Scheduling allows a task to suggest the identity of the task that a machine should be given to when that task relinquishes it. The idea is to take benefit from the cache entries built up by the current executing task. A related approach is *Affinity Scheduling* [SL93], which tries to schedule tasks on the same processor on which they most recently ran, under the assumption that this particular processor might still have some relevant data in its cache.

- *Non-Preemptive scheduling*: Once a task has been assigned to a machine, it will remain executing on that machine either until it finishes or until the machine leaves the computation (because this machine was reclaimed by its owner or because it failed). *Self-Scheduling* [TY86] and *Open Shop Scheduling* [KSW98] belong to this category.

  Self-Scheduling is used to schedule a set of parallel tasks that are independent iterations of a computational loop. The loop iterations are divided into a set of tasks, and these tasks are placed in a single global work queue that is available to all machines involved in the computation. When a machine becomes available, it removes tasks from the work queue for processing. This is repeated until the queue is empty. *Pure Self-Scheduling* is a strategy that assigns a single loop iteration to each task in the global work queue. *Chunk-Scheduling* [KW85] creates a set of equal-sized tasks that contain an arbitrary number of loop iterations. Variants of this method include the “bag-of-tasks” policy [BS95] and the Piranha Scheduling [GJK93], both of the Linda system [Gel95]. Another variant used in *Calypso* [BDK95] is Eager Scheduling. This allows a machine to be assigned a task that has already been assigned to other machine if no unassigned tasks remain. In this case, only the results produced by the first machine to finish the task will be considered.

  Open Shop Scheduling uses heuristics to produce a schedule for each machine (shop) indicating which tasks will be performed and in which order.

Dynamic scheduling strategies in general (both preemptive and non-preemptive), have the ability to adapt the scheduling they provide to a changing set of available resources.

A scheduler should provide a performance measure index, which describes the performance activity to be optimized. Possible performance-measurement indices include [Ber99]:

- *Application completion time*: This is the time an application takes to execute a task, from start to finish.
- *Efficiency*: This represents how well resources are being used. If there is a lot of resource waste, then efficiency will be low.
- *Throughput*: This is understood as the executing of a large amount of jobs over a larger amount of time.
- *Average Response Time*: This corresponds to the amount of time that users have to wait until they begin to receive output or results from this application.
- *Other performance activities*: For example operating system overhead or resource fragmentation, among others.

The performance measure index chosen depends on whether scheduling promotes application performance (application-aware) or system performance (system-aware). These goals may conflict with each other.

This thesis focuses on the problem of dynamic internal scheduling, i.e., assigning tasks to the machines available to an application. It should be noted that the number of available machines may dynamically change, depending both on the application needs and the number of resources the system is able of allocating to an application at a specific time. In other words, we are considering both dynamic partitions and malleable jobs. The performance measure index we consider is basically application-aware when performing internal scheduling, but introduces system-aware considerations when determining the number of machines to be assigned to an application

1.5 Example of Existing Schedulers

We will now review some existing scheduling systems by considering the different types of system organization with respect to the scale classification discussed in section 1.3.

1.5.1 Clusters – Intranet

In spite of their differences, the typical way in which both cluster of machines and intranets) are currently used is the following:

1. The system is divided into partitions consisting of different numbers of machines (classified as explained in section 1.4).
2. A number of queues are established, each one corresponding to a specific combination of job characteristics.
3. Each partition is associated with one or more queues, and its machines serve as a pool for those queues. Whenever machines are free, a job belonging to one of the associated queues is executed on the partition.

In practice *load-sharing packages* implement this functionality. A load-sharing package in general consists of the following main components [Ant97]:

- *Batch Queue System*: This allows users to submit jobs to some form of queuing system.
- *Scheduler*: This places the job on certain machines, with adequate resources to run the job, and
- *Job Management Tools*: These allow both users and administrators to interact with the job queues, status, priorities, etc.

Examples of these packages include:

- *Condor* [LLM88]: This is a high-throughput computing environment that can manage a large collection of computers such as PCs, workstations and clusters owned by different individuals. It harnesses idle computer CPU cycles (cycle stealing) and offers resource management for parallel and sequential applications. *Condor* is described in more detail in the next chapter, as it is the system used in our experimentation.

- *LoadLeveler* [Pre96], *Network Queuing Environment (NQE)* [NQE00, Haz97], *Network Queuing System (NQS)* [NQS94], *Load Sharing Facility (LSF)* [ZZ+93] and *Distributed Queuing System (DQS)* [DQS00]: These packages present many common features, such as being designed for large and heterogeneous systems, and allowing users to run jobs by matching their processing needs to available resources. As part of their job-management system, they serve as a job scheduler and provide a facility for building, submitting and processing jobs in a dynamic environment. They support both sequential and parallel jobs.

- *Piranha* [GK92, CF+95]: This is an execution model for Linda, developed to reclaim idle cycles from networked workstations for use in executing parallel programs. A Piranha program or job is assigned to a collection of workstations by a scheduler. On each workstation, the job has a representative worker process or *piranha*, which lies dormant until the node becomes available for computation. When the node becomes available, the local *piranha* becomes active and begins computing on behalf of the job. Piranha programs often follow a master-worker paradigm. A master or feeder process outputs tasks descriptions representing the work to be done by the job to a Linda tuple space.

- *Hector* [RF+96]: The Hector parallel run-time environment is designed to run MPI-based parallel programs while actively maintaining a balanced load and returning workstations to their owners. Hector is a complete job-scheduling and parallel run-time environment, intended to present to the user many features both to parallel and sequential jobs, including dynamic load balancing, checkpointing, near-real-time resource awareness, and transparency.

1.5.2 Internet

There are some low-level environments built with the aim of providing basic infrastructure for the grid, i.e., in order to obtain dependable, consistent, pervasive and inexpensive access to high-end computational capabilities [FK99]. On top of these users can develop their schedulers, in accordance with their needs. These schedulers should be more than extensions to MPP or intranet schedulers because, in an internet, the resource pool changes dynamically and the scheduler is not in control of all resources. Globus and Legion are examples of well-know systems that provide infrastructure for the grid.

- *Globus* [FK97]: The *Globus* system is intended to achieve a vertically integrated treatment of applications, middleware, and network. A low-level toolkit provides basic mechanisms such as communication, authentication, network information and data access. These mechanisms are used to construct various higher-level metacomputing services, such as parallel programming tools and schedulers. The long term goal of the Globus project is to build an *Adaptive Wide Area Resource Environment*, that is, an integrated set of higher-level services that enable applications to adapt to heterogeneous and dynamically-changing metacomputing environments.
- *Legion* [GW+94]: This is a metasystem project developed at the University of Virginia, designed to present users with a transparent interface to the available resources, both at the programming interface level as well as at user level. Legion addresses issues such as parallelism, fault tolerance, security, autonomy, heterogeneity, resource

management, and access transparency in a multi-language environment.

In addition to the above systems that provide basic services for accessing the grid, there are many problem-solving environments, which are oriented to particular applications. Examples of these environments include:

- *AppLeS* (Application-Level Scheduling) [BW+96, BW97]: This focuses on the development of scheduling agents for parallel metacomputing applications. Each agent is written on a case-by-case basis and each agent will perform the mapping of the user's parallel application [SWB98]. To determine schedules, the agent must consider the requirements of the application and the predicted load and availability of the system resources at scheduling time. Agents use the services offered by the NWS (Network Weather Service) [WSH99] to monitor the varying performance of available resources. *AppLeS* includes templates for master-worker applications and parameter sweep studies.
- *NetSolve* [CD97, CD98]: This is a client-agent-server system, which enables the user to solve complex scientific problems remotely. The NetSolve agent does the scheduling by searching for those resources that offer the best performance in a network. The applications need to be built using one of the API's provided by NetSolve in order to perform RPC-like computations. There is an API for creating task farms [CK+99] but it is targeted at very simple farming applications that can be decomposed by a single bag of tasks.
- *Nimrod/G* [AF+97, AGK00] is a resource management and scheduling system that focuses on the management of computations over dynamic resources scattered geographically over wide-area networks. It is targeted at scientific applications based on the "exploration of a range of parameterized scenarios", which is similar to our definition of master-worker applications, although our definition allows a more generalized

scheme of farming applications. The scheduling schemes under development in Nimrod/G are based on the concept of computational economy developed in the previous implementation of Nimrod, where the system tries to complete the assigned work within a given deadline and cost. The deadline represents a time by which the user requires the result, and the cost represents an abstract measure of what the user is willing to pay if the system completes the job within the deadline. Artificial costs are used in its current implementation to find sufficient resources for meeting the user's deadline.

- *Ninf* [SS+96, NSS99]: This is a client-server based network infrastructure for global computing. It allows access to multiple remote compute and database servers. The system schedules accesses from a client application to the remote library, with the goal of optimizing application performance.
- *PUNCH* [KF99]: This infrastructure consists of a collection of technologies and services that allow seamless management of applications, data and machines distributed across wide-area networks. *Punch* employs non-preemptive, decentralized, adaptive scheduling.

Other systems providing schedulers for internet found in the literature are: Prophet [WZ97] and Dome [AB95] for SPMD programs, VDCE [TH+97] for programs composed of tasks from mathematical task libraries, IOS [BRS97] for realtime, iterative automatic target recognition applications, SEA [SM97] for dataflow-style program dependence graphs, I-SOFT [FG+98] for applications that couple supercomputers, remote instruments, immersive environments and data systems, and MARS [GR96] for phased message-passing programs.

1.6 Discussion

In the light of the literature available, it is clear that scheduling of parallel applications on distributed systems is a highly varied subject, and includes works that tackle it from different perspectives, many having very different (sometimes contradictories) assumptions. Nevertheless, in view of what has been discussed in this chapter, our work is focused not only on the development of application-aware scheduling strategies but also on taking into account system-aware considerations, for malleable parallel applications that follow the master-worker paradigm. In this respect our work adopts similar assumptions to those made by certain problem-solving environments for internets, such as *AppLeS* and *NetSolve*. However, these scheduling techniques will be adapted to the features of typical Load Sharing Packages for intranet systems, such as *Condor*, which is characterized by using available idle resources, thereby providing a set of resources that can be seen as a dynamic partition.

INTRODUCTION

Chapter 2

Master-Worker Programming Model and Problem Description

Abstract

This chapter first introduces the simple master-worker programming model and then presents the generalized master-worker model considered throughout this work. Subsequently, the fundamental aspects of the execution environment in which the applications will be executed are described. We then detail the features of the systems used in the development phase –Condor and MW— which are an example of opportunistic systems. Finally, the problem of scheduling master-worker applications on such environments is introduced.

2.1 Introduction

In this chapter we will introduce and explain the relevance of the programming model that has been considered in this work, specifically, the master-worker parallel programming model. We will describe the main characteristics of this adopted programming model. We begin with a basic description of the simplest formulation of the model, and later present a generalized master-worker model.

The execution environment considered in this work was an opportunistic environment of heterogeneous resources. Such environments are composed of a variable number of machines, and machines can join or leave a computation dynamically (at run time). We used *Condor* as resource manager, and master-workers applications were implemented using *MW* (Master-Worker Tool), a set of libraries that simplifies the process of writing master-worker applications, in comparison to directly using *C* plus *PVM* or *MPI*. In fact, MW has been modified in order to support the requirements of the proposals of this work.

Once we have introduced both the programming model and the execution environment, we present the problems arising when executing applications following that programming model on such environments, and, in particular we introduce the problem that we are interested in, namely, the scheduling of master-worker applications on distributively-owned machines belonging to an opportunistic environment. Different problems arise due to the nature of the scheduling problem itself, and due to the opportunistic and heterogeneous environment considered.

2.2 Master-Worker Programming Model

In this section the simple master-worker paradigm is first presented, followed by the generalized version of this paradigm.

The master-worker model is used in many scientific, engineering and commercial applications, such as: software building and testing, sensitivity analysis, parameter space exploration, image and movie rendering, high energy physics event reconstruction, the processing of optical DNA sequencing, neural networks training and stochastic optimization among others [Can98, WW95, AI98].

2.2.1 Simple Master-Worker Model

As was described in the first chapter, the basic master-worker programming model consists of two entities: the master and multiple workers. The master is responsible for decomposing the problem into tasks and distributing these tasks among a farm of workers, as well as for gathering the partial results in order to produce the final computation result. The workers execute in a very simple cycle: receive a message with the task (input), process the task, and send the result back to the master (output). Usually communication only takes place between the master and the workers.

We will now show a detailed graphical example of how the simple master-worker paradigm works. First of all the master has a bag of tasks that can be understood logically by using the *parbegin-parend* constructors, as is shown in Figure 2-1. The main features of this high-level structure are, on the one hand, that there is a strong synchronization point (*parend)* and, on the other hand, that there are no communications among tasks. The corresponding algorithmic representation of this paradigm is shown in Figure 2-2. In practice, when working at a level of processes, this structure can be decomposed into master and workers processes communicating task descriptions and task results by messages, as is exemplified in Figure 2-3, which depicts one

Master process and three *Worker* processes (worker1, worker2 and worker3). In this example, initially (as shown in Figure 2-3a) task T1 to T8 are ready to be executed.

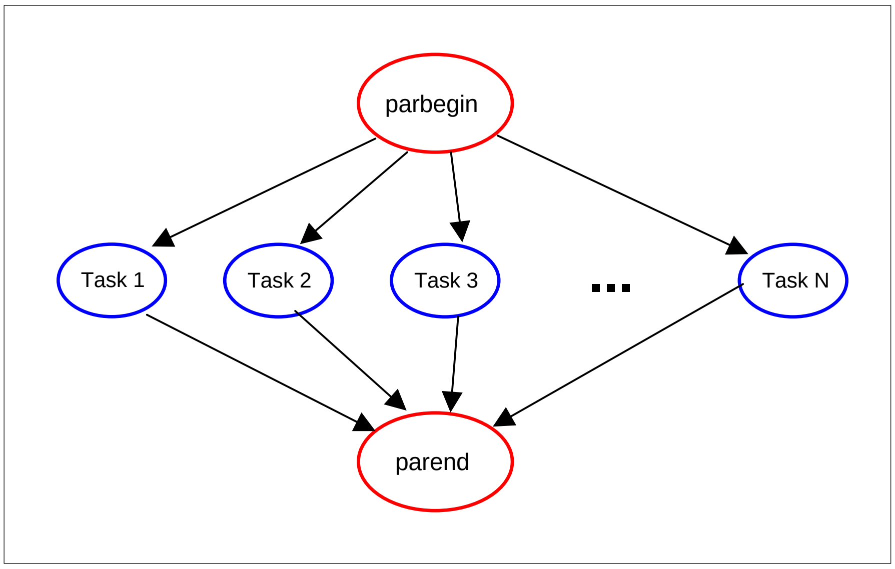


Figure 2-1. High-level structure corresponding to the master-worker paradigm.

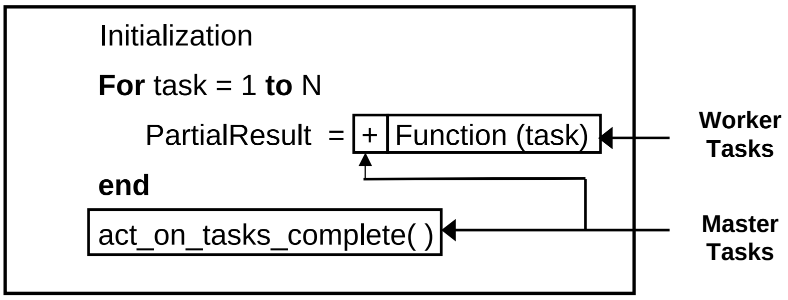


Figure 2-2. Simple Master-Worker algorithm

The master then sends one task to each of the available workers (Figure 2-3b). In this case, it sends the first three ready tasks that are ready in the task list (T1, T2 and T3) to worker1, worker2 and worker3 respectively. The *Next Task to be Scheduled* (NTS) pointer represents the next task in the task list to be executed when a worker finishes the current task being executed; in this example T4 is the next task to be executed.

Each time a worker finishes, it sends the results back to the master and the master sends it a new task (Figure 2-3c). In this case, the execution of T1 and T3 finish, and the master receives those results (R1 and R3), and sends tasks T4 and T5, respectively, to those workers. As a consequence, the next task to be executed is T6.

This process is repeated until all tasks have been completed (Figure 2-3d), in particular in this example, until the results of the eight tasks R1 to R8 have been received by the master.

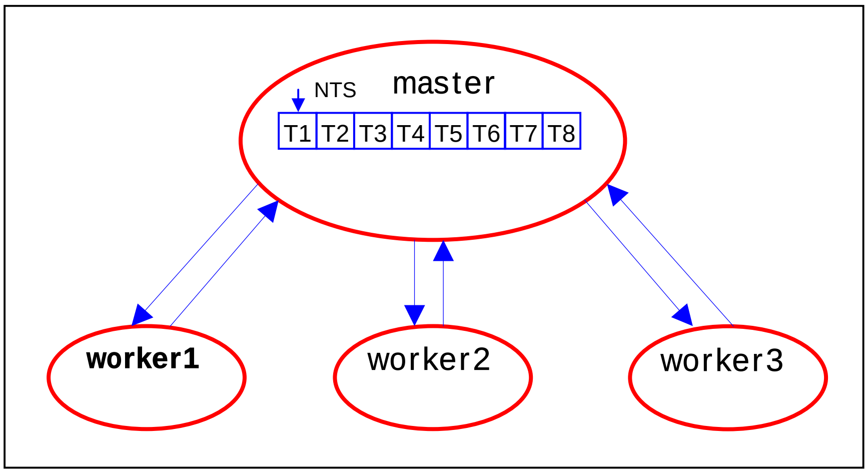


Figure 2-3a

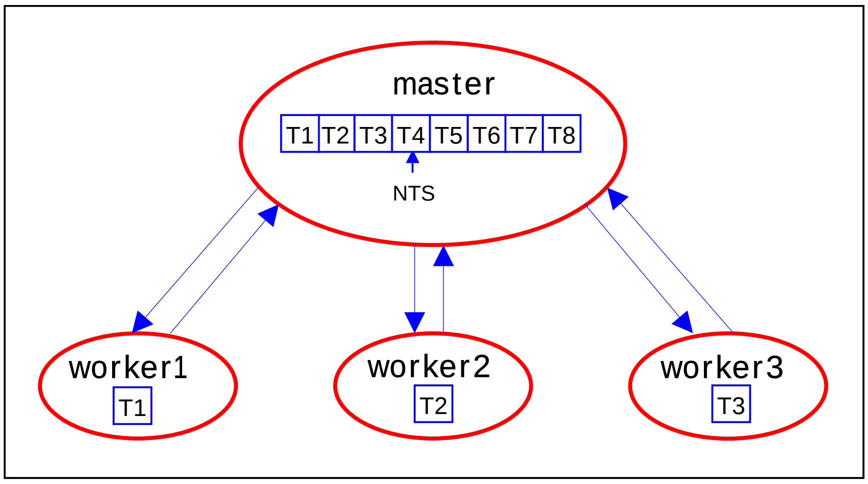


Figure 2-3b

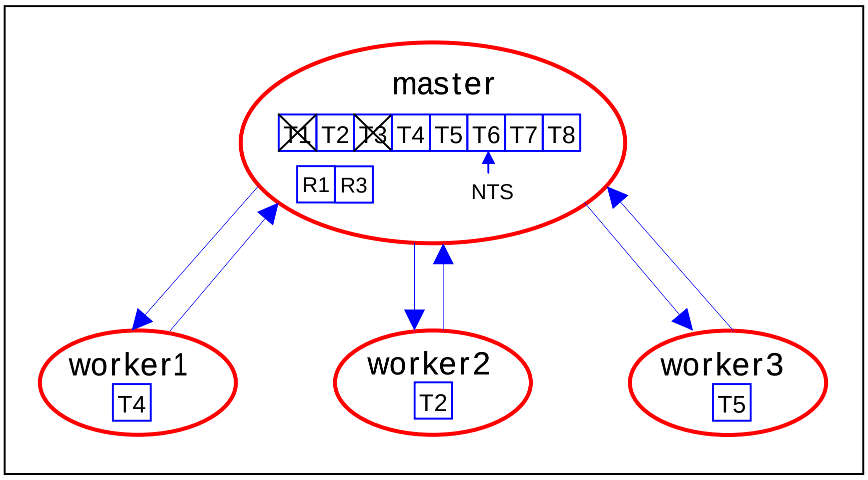


Figure 2-3c

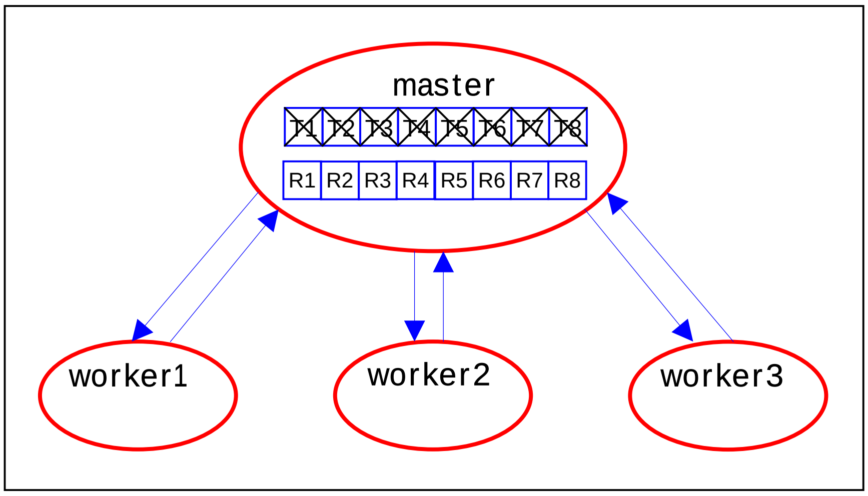
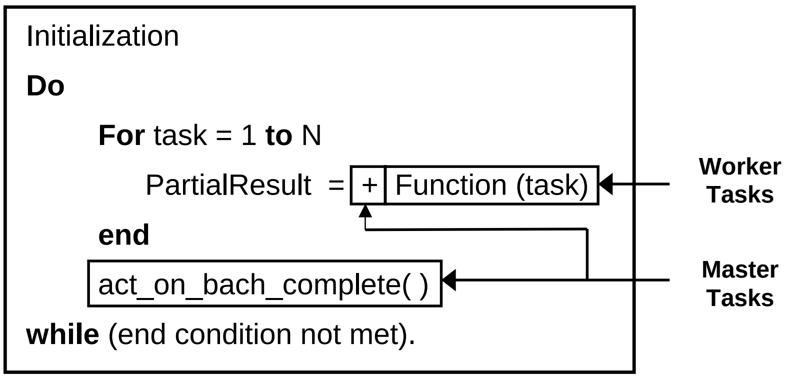


Figure 2-3d

Figure 2-3**.** Basic master-worker model

2.2.2 Generalized Master-Worker Paradigm

In contrast to the simple master-worker model in which the master solves one single set of tasks, the generalized master-worker model can be used to solve problems that require the execution of several batches of tasks. Figure 2-4 shows an algorithmic view of this paradigm, and Figure 2-5 shoes its corresponding representation using the *parbegin-parend* constructors.



```
Initialization
Do
        For task = 1 to N
              PartialResult  = + Function (task)
        end
        act_on_bach_complete( )
while (end condition not met).
```

Figure 2-4. Generalized Master-Worker algorithm

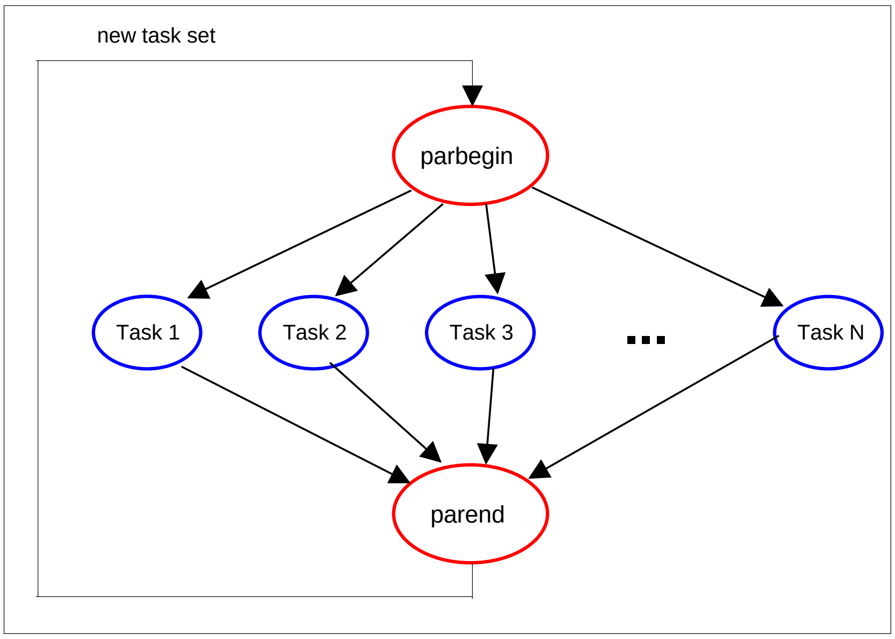


Figure 2-5. *Parbegin-parend* structure corresponding to the generalized master-worker paradigm.

A master process will solve the *N* tasks of a given batch by looking for Worker processes that can run them. The master process may carry out certain intermediate computations with the results obtained from each worker as well as some final computations (that cannot be parallelized), when all the tasks of a given batch are completed. After this a new batch of tasks is assigned to the master and this process is repeated several times until completion of the problem (*K* cycles, which are later referred to as *iterations*). Workers execute *Function (task)* and *PartialResult* is collected by the master. The completion of a given batch induces a synchronization point in the iteration loop, followed by the execution of a sequential body.

This paradigm is very attractive, first because it is very easy to program and second, because there are many problems that can be naturally mapped onto it. N-body simulations [GF96], genetic algorithms [Ban98], Monte Carlo simulations [BRL99] and material science simulations [PL96] are just a few examples of natural computations that fit into this generalized master-worker paradigm.

It also has another attractive characteristic: it has a central control point (the master), a fact that will be exploited by the scheduling mechanism. In addition to these characteristics, empirical evidence has shown that, for a range of applications, the execution of each task in successive iterations tends to behave similarly, so that the measurements taken for a particular iteration are good predictors of near-future behavior [PL96]. This means that this kind of applications have a high degree of predictability and, therefore, it would be possible to take advantage of it in deciding both the use of the available resources and the allocation of tasks to workers in a dynamic and adaptive way. There are other works in the literature [BG96, NVZ96] whose experimental results also confirm that iterative parallel applications usually exhibit regular behaviors that can be used by an adaptive scheduler.

2.3 Parallel Execution Environment for Master-Worker Applications

According to the characteristics of the applications that follow a master-worker model, its natural manner of execution would be on a parallel/distributed environment, in such a way that the different workers execute tasks simultaneously.

The master-worker paradigm is an example of malleable applications that can adapt their execution to a changing number of machines. We will now comment on how the master-worker applications would be executed depending on the type of partition considered in the parallel environment:

- *Fixed*: If the number of machines is equal to or greater than the number of tasks, the master-worker application would be executed using the same number of machines as number of tasks, therefore assigning a worker to each machine. This constitutes the ideal case guaranteeing the minimal possible execution time, if the communication time between master and workers is small compared to each task's computation time. If the number of machines is less than the number of tasks, the tasks will be executed according to the order determined by the master process.

In the case of the size of the partition being known before the scheduling of the tasks, this value could be taken into account in order to distribute those tasks in a balanced way.

- *Variable* and *Adaptive*: In this case, for each execution of the same master-worker application, a different fixed number of machines would be used. Under this approach, it would be necessary either to distribute tasks at the beginning of the execution, once the number of machines constituting the partition assigned to the application was known, or to carry out a dynamic assignment as tasks are finished.
- *Dynamic* (opportunistic environments): This is the more complex case. The number of machines may change during the execution of the application. This requires dynamic task management that also includes the capacity to react to machine losses throughout execution time.

In the two former cases a single application that used the services provided by a message-passing library such as PVM or MPI, would be sufficient. In the third case, it would be necessary, in addition, to join the application with the services provided by an opportunistic resource management system.

An example of this union was shown in [PL96] where the Condor opportunistic environment was used close to PVM applications, allowing the development of any generic parallel application that uses opportunistic resources. Recently the MW tool was proposed in order to simplify the construction of master-worker applications on opportunistic environments. Initially, MW supported the execution of a bag of tasks, so MW was modified to support the proposed generalized master-worker model. The schematic vision of the main components used is shown in Figure 2-6, where the master process (or Driver) is executed on the owner's machine, and the worker process are executed on the remaining machines belonging to a Condor pool. Communication and resource management services are provided by *PVM* and *Condor* respectively.

The following subsections briefly describe the structure of a general resource management system for opportunistic environments, Condor and MW.

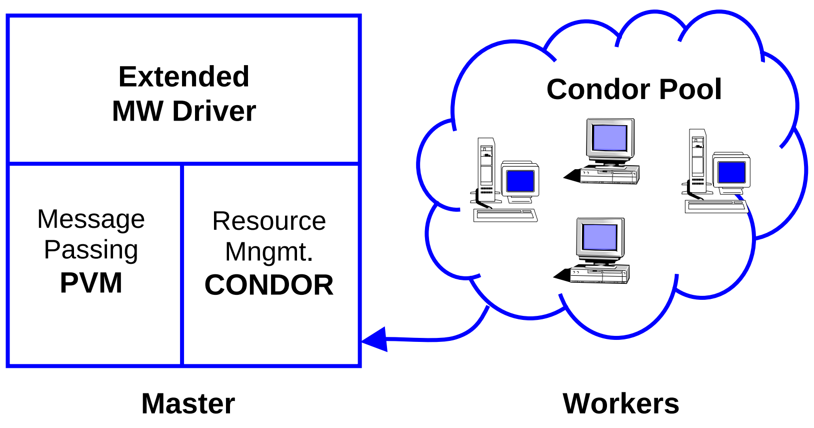


Figure 2-6. Executing and Programming Environment

2.3.1 General Structure of Opportunistic Resource Management Systems

When considering an opportunistic environment, resource management (RM) is a basic point. The principal layers of a resource management system proposed by Livny and Raman [FK99] are show in Figure 2-7 and are briefly explained below.

1. *Local RM layer*: This provides basic RM services for processes executing within the domain of that resource. It could be, for example, an operating system.

2. *Owner layer*: This represents the interest of the resource owner. It provides access control mechanisms to the resource, that is, it implements the owner policy for a particular resource, for example, so that the machine `aows10.uab.es` with `Memory=500` and `KFlops=150000` (resource) could be accessed from `2pm` to `10am` (when) by users belonging to the `friend_research` group (to whom).

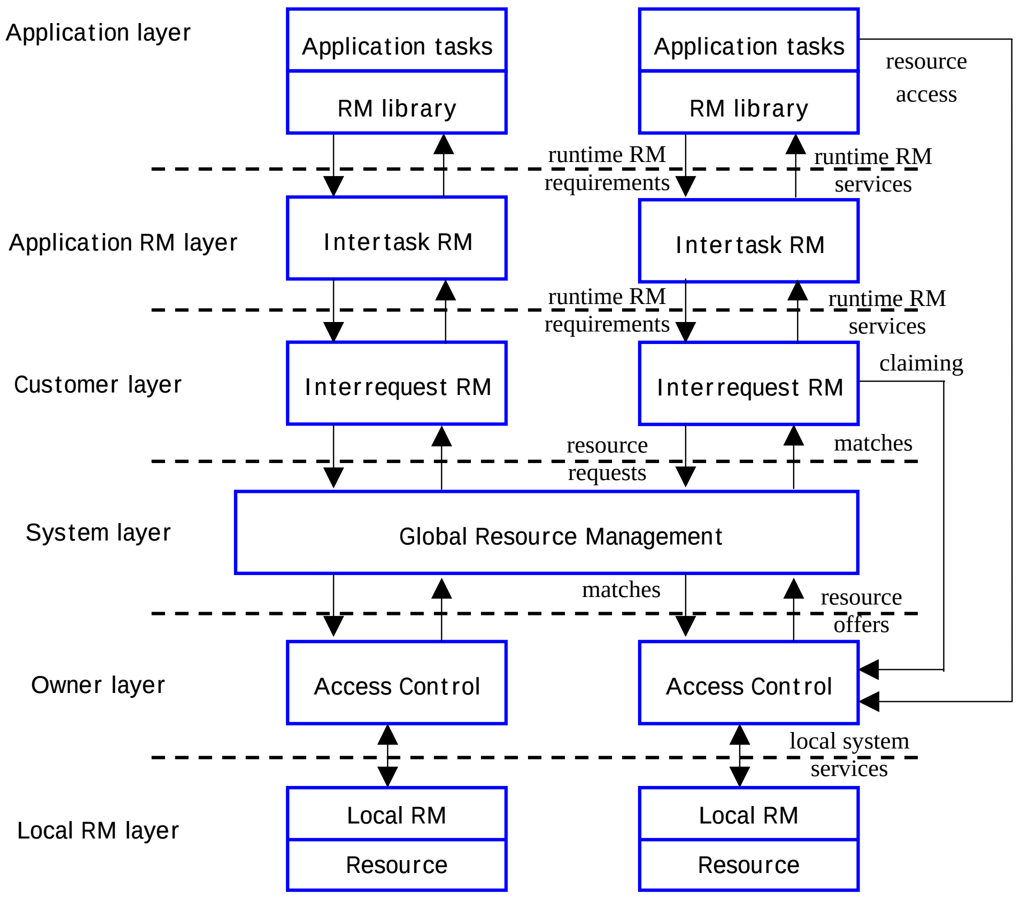


Figure 2-7. Layers of an RMS

3. *System layer*: This represents the global resource allocation layer. Here, the matching policy is implemented in order to match resource offers and resource requests in such a way that the constraints of both are satisfied. For example, a match is done with the request of user `David` and machine `aows10.uab.es`.
4. *Customer layer*: This layer represents the interest of the resource management system users, for example, that user `David` belonging to the `friend_research` group needs a machine with `Memory>250` and `Kflops>100000`. The duties of this layer are handling the users' resource requests and interacting with the system layer by sending it those requests, by considering a priority scheme.
5. *Application RM layer:* This is responsible for establishing the application task runtime environment on the claimed machines. For example, the environment for running `David's` job is set up on machine `aows10.uab.es`. This layer also supports adaptive applications that grow when more resources become available.
6. *Application layer:* This represents user application tasks. These tasks use the resource given by the application RM layer. In the example, `David's` job is now executed on machine `aows10.uab.es`. If this job subsequently needs another machine, then a request to the application RM layer will be generated.

2.3.2 Condor Overview

Condor [LB+97] is a software system that runs on a cluster of workstations to harness wasted CPU cycles. It was initially developed at the University of Wisconsin-Madison in 1986. Condor was first developed for *Unix* systems, and can currently be executed on a wide range of machines. A Condor pool consists of any number of machines, possibly of heterogeneous architecture and that may or may not have different operating systems, which are connected by a network. One machine, the *central manager*, keeps track of all the resources and jobs in the pool.

Condor allows *High Throughput Computing* (HTC) to be attained, that is, large amounts of processing capacity sustained over long time periods. The resources, i.e. hardware, middleware and software, are large, dynamic and heterogeneous. Tasks are usually loosely coupled or independent, and the goal is to use processor cycles from idle machines. In an HTC the resource manager is only aware of the current state of resources (all resources are opportunistic), and therefore no future planning can be done.

The most relevant mechanisms included in the Condor system are [Con99]:

- *Queue Management:* When a user submits a job, it goes to the local job queue in the submitting machine.
- *Priority Schemes:* There are priorities assigned to each user. Machines are allocated to users according to that user's priority. In addition, Condor provides the user with the capability of assigning priorities to each submitted job. These job priorities are local to each queue.
- *Matchmaking:* This enables requests for services and resource owners to find each other. This is accomplished via the *ClassAd* mechanism, which works in a similar way to newspaper classified advertising. All machines in the Condor pool advertise their resource properties, such as available RAM memory, CPU type, CPU speed, operating system, current load average and other properties, into a "resource offer" ad. In the same way, when submitting a job, the "resource request" ad contains the required set of resources to run the job. Condor matches resources and requests thereby satisfying both parts.
- *Checkpointing:* Condor only uses idle machines to compute jobs. If the user returns when his machine is being used by a Condor job, the job being executed there leaves that machine. This job is checkpointed, i.e.

all the work it has already performed is saved, so the job can be moved onto another machine and continue executing in the new allocated machine. Checkpointing avoids waste of computational resources, and is the base for supporting dynamic process migration and fault tolerance. Checkpointing has certain limitations, such as not being able to be used in message-passing parallel applications, the process cannot use certain system calls (*fork* for example), and processes are subjected to limitations in the use of sockets.

- *Remote I/O:* Jobs executing on a remote site can access their local data, that is, in the environment from which the job was submitted. This allows crossing administrative domains. In order to use this service, users must link their application with the Condor libraries.
- *Flocking:* This allows jobs to be executed on more than one pool. Users would submit their jobs from their local machine, and if there are not enough machines available in their pool, and if the authorization rights are properly set, their jobs can be executed on remote pools.
- Enables *management of dynamic resources* (opportunistic ones). Condor handles machines that are joining and leaving the pool of available machines.

A machine belonging to a Condor pool could act as:

- *Central Manager*: There is only one Central Manager per pool. This machine is the collector of information, and it is where the match between resources and requests takes place.
- *Execute Machine*: This role corresponds to machines that will execute jobs in the pool.
- *Submit Machine*: This role corresponds to machines allowed to submit jobs in the pool.

- *Ckeckpoint Server*: The pool may have a centralized point to store all the ckecpoint files for the jobs submitted in the pool.

Most machines usually play more than one simultaneous role in a Condor pool.

This functionality is implemented by means of five daemons (*condor_master*, *condor_startd*, *condor_starter*, *condor_schedd* and *condor_shadow*). These daemons and their interactions are shown in Figure 2-8, which depicts the situation of a pool with N machines plus a Central Manager in which a job submitted on machine 2 is running on machine N:

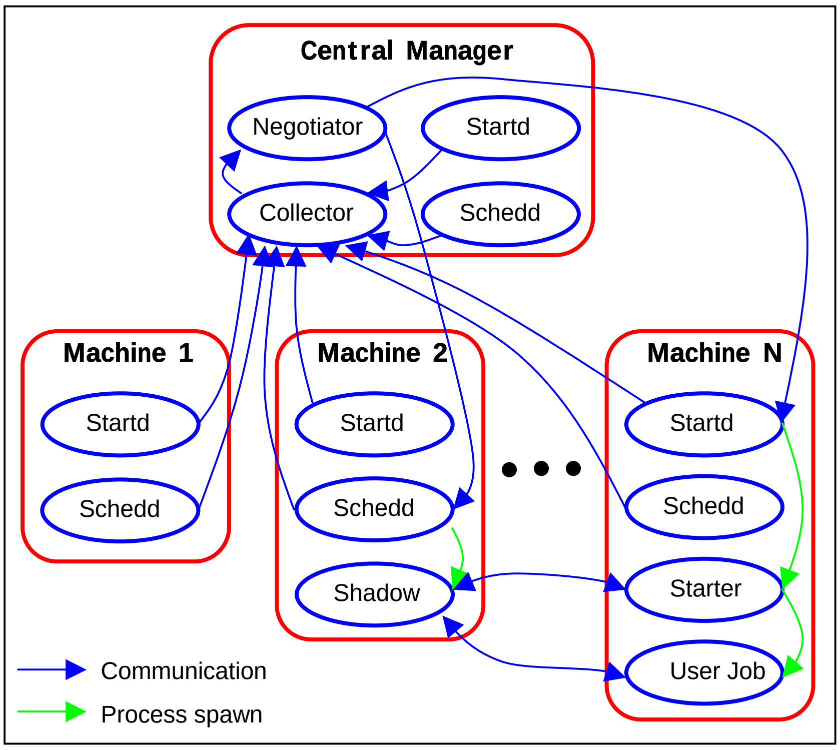


Figure 2-8. Architecture of a Condor Pool.

- `condor_master`: This daemon is responsible for keeping the rest of the Condor daemons running on each machine in the pool. It spawns the other daemons, and if any of them crashes, it restarts them.

`condor_master` is executed on every machine in the pool, regardless of the role each machine is playing. For the sake of simplicity, this daemon is not shown in Figure 2-8.

- *`condor_startd`*: This daemon represents a resource, i.e. a machine capable of running jobs. It advertises certain attributes concerning the resources being used to match it with pending resource requests. It will run on any machine of the pool that is able to execute jobs. When the *`condor_startd`* is ready to execute a job, it spawns the *`condor_starter`*.
- *`condor_starter`*: This program spawns the remote Condor job on a given machine, and monitors it once it is running. When a job is completed, *`condor_starter`* sends back any status information to the submitting machine and exits.
- *`condor_schedd`*: This daemon represents resource requests to the Condor pool. It will run on any machine that is able to submit jobs. When a job is submitted, *`condor_schedd`* stores it in the "job queue". *`Condor_schedd`* advertises these jobs. Once one of these has been matched with a given resource, it spawns a *`condor_shadow`*.
- *`condor_shadow`*: This program runs on the machine where a given request was submitted, and acts as the resource manager for the request. It handles remote system calls: any system call performed on the remote-execute machine is sent over the network back to the *`condor_shadow`*, which performs the system call locally, and the result is sent back to the remote job.
- *`condor_collector`*: This daemon collects all the information about the status of the pool. It only runs at the Central Manager.
- *`condor_negotiator`*: This daemon is responsible for all the matchmaking within the system. It also only runs at the Central Manager.

2.3.3 Overview of MW

MW [GK+00] is a programming framework, composed of a set of C++ abstract classes, which allows fast and easy development of master-worker applications. With MW, users need not address issues such as fault tolerance, while interprocess communication is simplified for users. In this way, MW provides an API for implementing simple master-worker applications. In this work, we have extended MW in order to support the generalized master-worker model.

MW uses Condor as a resource manager. A resource manager in this context includes: resource request and detection, infrastructure querying, fault detection and remote execution.

As regards communications, there are MW versions that perform communications by using PVM, the file system and sockets. In this work, MW was used with PVM [GB+94]. MW workers are independent jobs spawned as PVM programs.

An application in MW has three base components: Driver, Tasks and Workers. The Driver is the master, who manages a set of user-defined tasks and a pool of workers. The Workers execute Tasks. To create a parallel application, the programmer needs to implement some pure virtual functions for each component.

Driver: This is a layer that sits above the program's resource management and message passing mechanisms. (Condor and PVM, respectively in the implementation used). The Driver uses Condor services for getting machines to execute the workers and to get information about the state of those machines. It creates the tasks to be executed by the workers, sends tasks to workers and receives the results. It handles workers joining and leaving the computation and rematches running tasks when workers are lost. To create the Driver, the user needs to implement the following pure virtual functions:

- **get_userinfo():** Processes arguments and does initial setup.

- **setup_initial_tasks():** Creates the tasks to be executed by the workers.
- **pack_worker_init_data():** Packs the initial data to be sent to the worker upon startup.
- **act_on_completed_task():** This is called every time a task finishes.

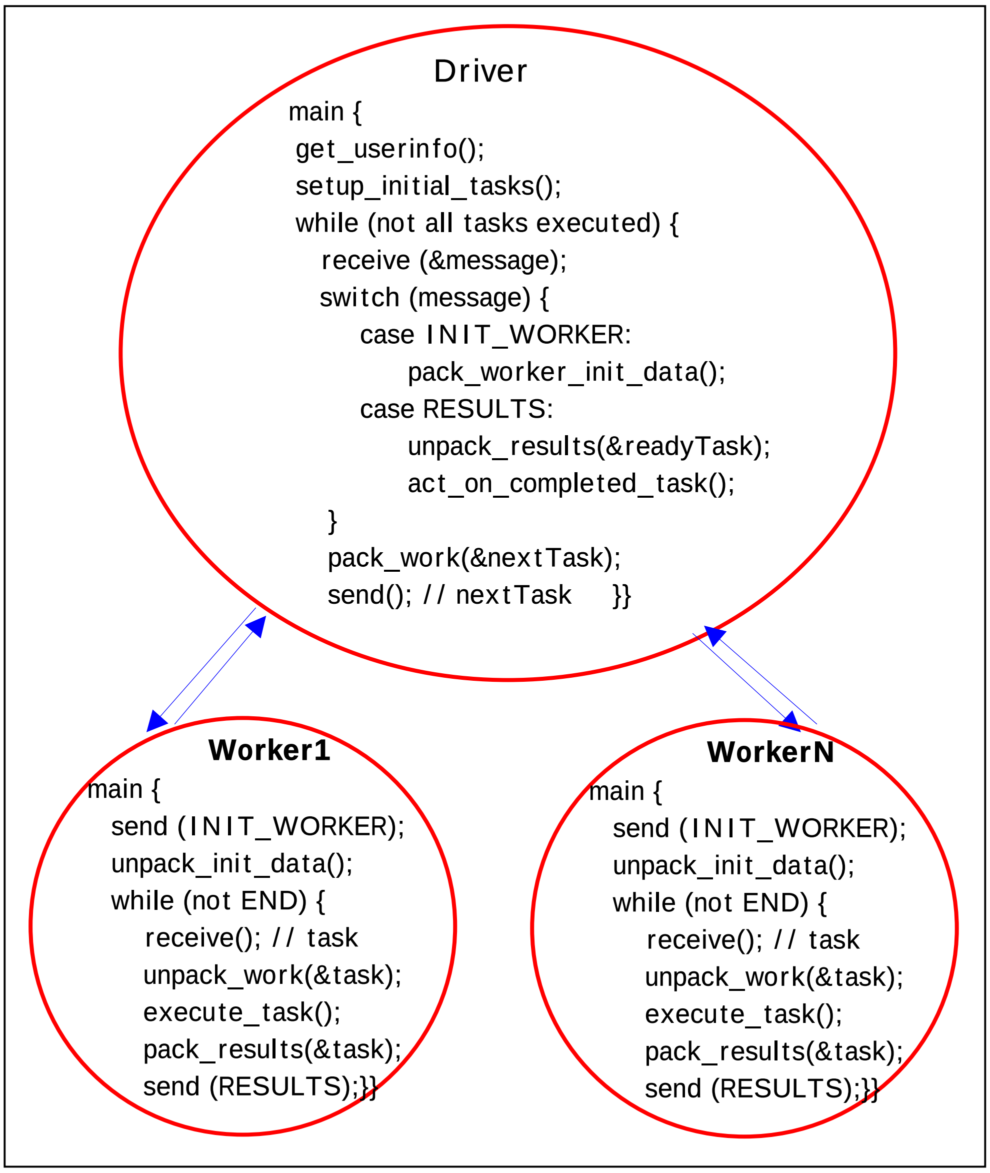
Task: This is the unit of work to be done. It contains the data describing the tasks (inputs) and the results (outputs) computed by the worker. The programmer needs to implement functions for sending and receiving this data between the master and the worker. The functions are the following:

- **pack_work():** Packs the work to be sent to the workers.
- **unpack_work():** Unpacks the work to be done.
- **pack_results():** Packs the results obtained.
- **unpack_results():** Unpacks the results received.

Worker: This executes the tasks sent to it by the master. The programmer needs to implement the following functions:

- **unpack_init_data():** Unpacks the initialization data passed in the Driver pack_worker_init_data() function.
- **execute_task():** Computes the results for a given task.

Figure 2-9 shows a simplified view of the MW Driver and MW Worker that includes the order in which the virtual functions completed by the user are executed. Appendix C shows an example of the implementation of the virtual function, corresponding to the Driver, Worker and Task for an image-thinning application.



```
main {
 get_userinfo();
 setup_initial_tasks();
 while (not all tasks executed) {
    receive (&message);
    switch (message) {
        case INIT_WORKER:
              pack_worker_init_data();
        case RESULTS:
              unpack_results(&readyTask);
              act_on_completed_task();
    }
    pack_work(&nextTask);
    send(); // nextTask    }}
```

```
main {
   send (INIT_WORKER);
   unpack_init_data();
   while (not END) {
      receive(); // task
      unpack_work(&task);
      execute_task();
      pack_results(&task);
      send (RESULTS);}}
```

```
main {
   send (INIT_WORKER);
   unpack_init_data();
   while (not END) {
      receive(); // task
      unpack_work(&task);
      execute_task();
      pack_results(&task);
      send (RESULTS);}}
```

Figure 2-9. MW virtual functions.

Applications running on top of MW are fault tolerant in the presence of the failures of machines that run worker processes. If a worker does not finish a task because it failed, the driver will re-send this task to other available worker. In order to make computations reliable with respect to driver failures, users can implement functions for writing and reading the state contained in the application master and tasks. This constitutes checkpointing and is performed in a user-defined frequency.

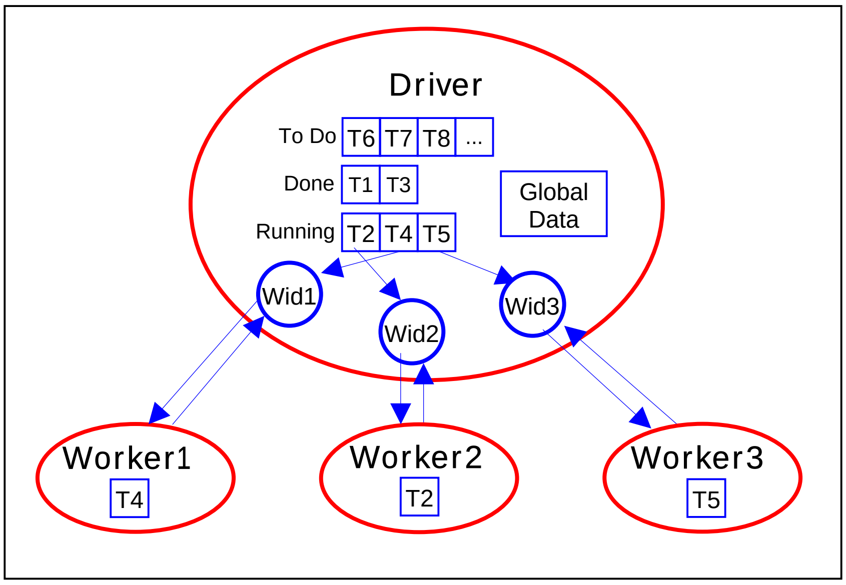


Figure 2-10. MW components.

With respect to task scheduling the Driver internally manages a list of workers and certain lists of tasks. Figure 2-10 depicts a Driver, 3 Workers and 8 Tasks (T1, T2, ..., T8). The *ToDo* list contains the tasks that are ready to be executed. The *Done* list contains the tasks that have already been executed, and the *Running* list contains the tasks that are currently being executed. Task scheduling is performed by assigning the first task in the *ToDo* list to the first idle worker in the worker list. In the Driver, the user can specify the way the task list will be ordered. By default, the worker list is ordered by using the KFLOPS information, provided in our case by Condor. The user can also specify a benchmark task to be executed on each worker when it joins the computation. If this is carried out, machines will be ordered by the benchmark factor.

2.4 Performance Model for Master-Worker Applications on Opportunistic Environments

In the case of master-worker applications, the overhead incurred in discovering and allocating new resources can be significantly alleviated by not releasing the resource once the task has been completed. Workers will be kept alive at the resource, waiting for a new task. However, by doing so, an undesirable scenario may arise in which some workers may be idle while other workers are busy. This situation will result in a poor utilization of the available resources in which all the allocated workers are not kept usefully busy and, therefore, application efficiency will be low. In this case, efficiency may be improved by restricting the number of allocated workers.

If we consider execution time, a different criterion will guide the allocation of workers as the more workers allocated for the application the lower its total execution time. The speedup of the application then directly depends on the allocation of as many workers as possible.

In general, the execution of a master-worker application implies a trade-off between the speedup and the efficiency achieved. On the one hand, our aim is to improve the speedup of the application as new workers are allocated. On the other hand, we also want to achieve high efficiency by keeping all the allocated workers usefully busy.

In this work we consider the problem of maximizing the speedup and efficiency of a master-worker application through both the allocation of the number of processors on which it runs and the scheduling of tasks to processors during runtime.

Scheduling strategies are evaluated by measuring the efficiency and total execution time of the application.

Resource efficiency (*E*) for *n* workers is defined as the ratio between the amount of time workers spent doing useful work and the amount of time workers were able to perform work.

$$E = \frac{\sum_{i=1}^{n} T_{work,i}}{\sum_{i=1}^{n} T_{up,i} - \sum_{i=1}^{n} T_{susp,i}}$$

n: Number of workers.

$\boldsymbol{T_{work,i}}$: Amount of time that worker *i* spent doing useful work.

$\boldsymbol{T_{up,i}}$: Time elapsed since worker *i* is alive until it ends.

$\boldsymbol{T_{susp,i}}$: Amount of time that worker *i* is suspended, that is, when it cannot do any work for the application.

Execution Time (ET_n) is defined as the time elapsed from when the application begins its execution until it finishes.

$$ET = T_{finish} - T_{begin}$$

$\mathbf{T_{finish,n}}$: Time of the ending of the application, measured as the time at which the master finishes.

$\mathbf{T_{begin,n}}$: Time of the beginning of the application, measured as the time at which the master begins.

In agreement with [EZL89], we view efficiency as an indication of benefit (the higher the efficiency, the higher the benefit), and execution time as an indication of cost (the higher the execution time, the higher the cost). The implied system objective is to achieve the efficient usage of each processor, while taking into account the cost to users. It is important to know, or at least to estimate, the number of processors that yield the point at which the ratio between execution time and efficiency is minimized, that is, optimized. This would represent the desired allocation of processors to each job.

More formally, we define the Execution-Efficiency Ratio as:

$$EER(i) = \frac{ExecutionTime(i)}{\mathrm{Re}\,sourceEfficiency(i)}$$

EER(i): Execution-Efficiency Ratio for *i* machines.

Execution Time (i): Application execution time when *i* machines are used.

Resource Efficiency (i): Efficiency achieved when *i* machines are used.

Releasing under-utilized processors could be beneficial both for the whole system and for the particular user. From the system perspective, released processors could be allocated to other users that, in turn, will improve overall cluster throughput. A particular user will also benefit because cluster job managers normally make use of priority and aging mechanisms in their allocation policies. Every user has a priority and the job manager uses that priority to directly decide how many resources are going to be allocated to that given user. The better the priority, the more resources the user will get. The aging mechanism assigns a lower priority to users when they have already been allocated resources for a long time. This mechanism will ensure that resources will be fairly allocated to all users through time. Therefore, user priority for allocating resources will be more negatively affected when their applications are running on a set of under-utilized resources.

2.5 Challenges of Master-Worker Applications Running on Opportunistic Environments

Up to now, we have seen both the programming model, the execution environment and the performance model that we have adopted in this work. Despite the conceptual simplicity of the master-worker paradigm, it presents some interesting challenges. Some challenges are related to functioning such as *master and worker fault tolerance*. If either the master or any worker fails the application will either never finish or will produce incorrect results. The failure of a worker can be detected and the task that it was executing can be re-executed on another worker, increasing the execution time of the whole application. Master failures can be handled by periodically checkpointing the whole master-worker application, which can be very hard to perform, depending on the exact state of the application that we wish to store. For example, it is easier to save the state of tasks already executed, and the pending task list, than to save the exact state of every component participating in the computation. Fault tolerance aspects are partially covered by MW.

There are many other challenges related to improving the performance of master-worker applications running on opportunistic environments:

- The role played by the master: This may or may not execute certain tasks; or in the case of a hierarchy, this could be a worker from another master.
- If messages are large in comparison to task computation time, the way tasks are sent to workers may depend on the network considered. Packs of tasks could be sent to workers located far away (belonging to a computational grid, for example), in order to minimize the network overhead. If workers are close to the master, a possibility is to send a task as soon as one is ready to be executed and as soon as there is a worker ready to execute them.
- Task scheduling: The way in which tasks are assigned to workers for their execution. This offers many possibilities, including random, FIFO, or policies that use a predictive model of task execution time. The amount of information used in this prediction differs throughout policies.
- Selecting the number of workers to participate in a computation. Some of the possibilities are:
  - One worker per task.
  - One worker per machine allocated by the master.
  - A variable number of workers, depending on the behavior of the tasks.

- The impact of preemption: In an opportunistic environment machines can appear (resource occupied by their owner can become available without any advance notice) and disappear (available resources can be reclaimed at any time). When users reclaim their machine, the worker executing a task there has to leave this machine immediately. This is a special case of worker failure (as it is not longer available), and the performance of the whole application will be affected.
- Dealing with heterogeneous machines, from the point of view of both reliability and performance. Even homogeneous machines could demonstrate different performance due to the load they have at a particular moment. Both the assignment of tasks to workers, and the number of worker that should be used, depends on the relative performance of the workers.

From all the challenges described above, we have studied in this work the task scheduling problem, the selection of the number of workers that will participate in the execution of the master-worker application and also the impact of machine reclaim. Specifically, our study is focused on giving answers to the following questions:

– How can tasks be assigned to workers? When the execution time incurred by the tasks of a single iteration is not the same, the total time incurred in completing a batch of tasks strongly depends on the order in which tasks are assigned to workers. Theoretical work has proved that simple scheduling strategies based on list-scheduling can achieve good performance [Hall97].

– How many workers should be allocated to the application? A simple approach would consist of allocating as many workers as tasks are generated by the application, at each iteration. However, this policy will generally result in poor resource utilization (low efficiency) because some workers may be idle if they are assigned a short task while other workers may be busy if they are assigned long tasks.

– How is the whole performance of an application affected by a machine leaving the computation, and what can be done to alleviate this effect?

Both homogeneous and heterogeneous environments were considered, and we also assumed that communications take much less time than computations.

Taking into account the layered structure of a Resource Management System (see Figure 2-7),

Figure 2-11 depicts the extended version of such a Resource Management System according to our goals. In italics we show the particular elements that this work is focused on, and its location within the layered structure of the Resource Management System, in particular scheduling, dynamically determining and adjusting the number of workers involved in a computation and finally the strategies to alleviate the impact of machine reclaim.

The *scheduling agent* determines the way resources obtained will be used by the application tasks. The *Adjust Component* determines the number of machines suitable for getting good resource usage and reasonable execution time. Finally, the *Replication Agent* provides the mechanisms for reducing the impact produced when a machine participating in a computation is lost.

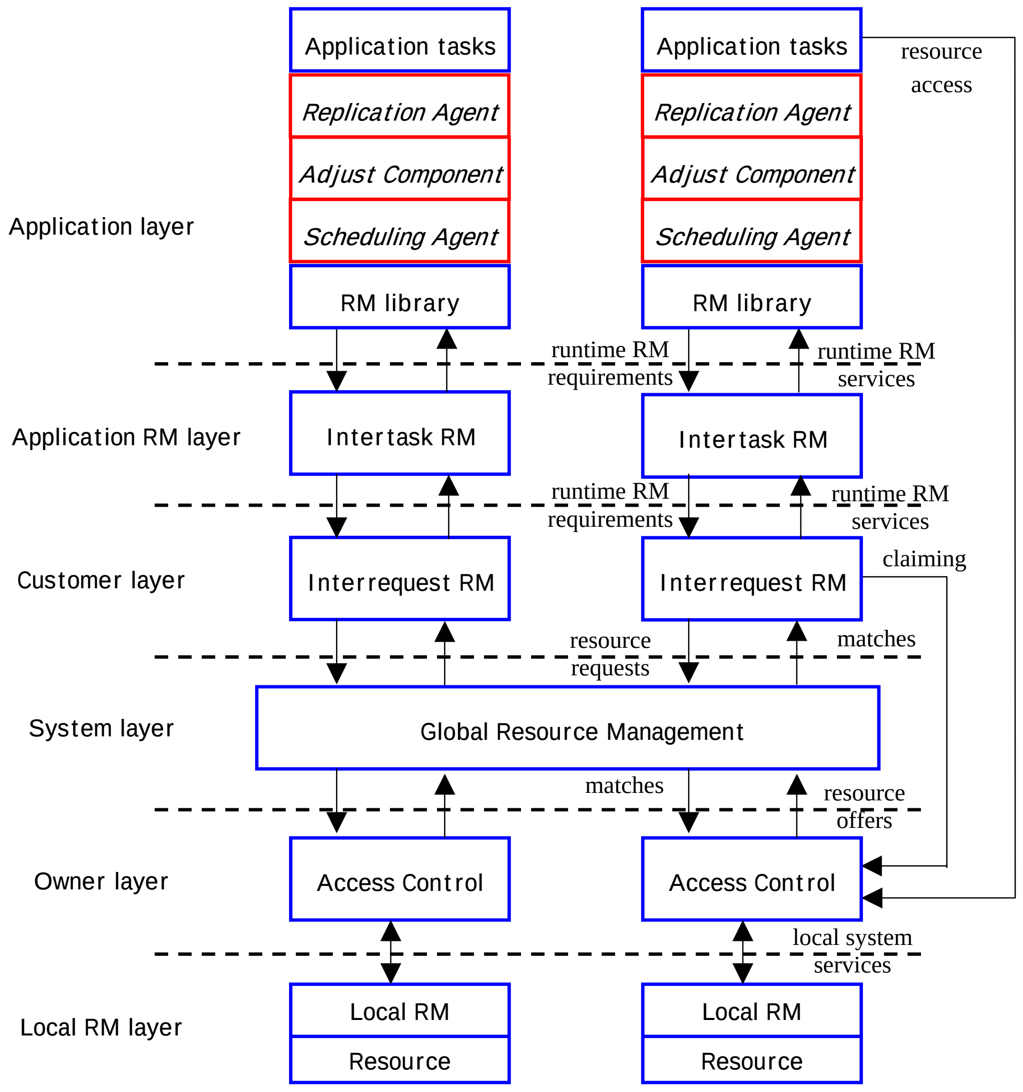


Figure 2-11. Extended layers of an RMS.

Master-Worker Programming Model and Problem Description

Chapter 3

Scheduling of Master-Worker Applications on Homogeneous and Dedicated Clusters

Abstract

This chapter first states the scheduling problem for master-worker applications on a cluster of dedicated homogenous machines. Then the Random & Average *scheduling strategy is introduced and studied by simulation through comparing it with other scheduling policies.*

3.1 Introduction

In chapter 2, we introduced the programming model, the execution environment and briefly described the problems to be studied. This chapter focuses on the scheduling of master-worker applications on dedicated and homogeneous clusters, which constitute our departure point. In the following chapters this formulation will be extended.

The scheduling problem in question is equivalent to the minimum makespan problem, for which multiple solutions exist in the literature. Most of these solutions have the main drawback of starting from very simple assumptions where, for example, the execution time of the tasks are known *a priori*, and do not have any variability. In our study, we will adopt assumptions, such as the variability of the task execution times, that bring the study closer to more realistic situations.

First, the *Random & Average* proposed scheduling policy is introduced. This scheduling strategy dynamically measures task execution times to control the assignment of tasks to workers. The effectiveness of the proposed strategy was assessed by means of simulation experiments in which several scheduling policies were compared. In the comparison, we considered a large set of different factors in modeling the behavior of master-worker applications. From these experiments, we have observed that the proposed strategy obtains similar results to other strategies that use a priori information about the application.

3.2 Problem Statement

The way in which tasks forming a parallel application are assigned to machines to be executed, and the number of machines that are to be used have a significant influence on both the execution time and the efficiency exhibited by the application. Let us suppose we have four tasks with different execution times, as shown in Figure 3-1:

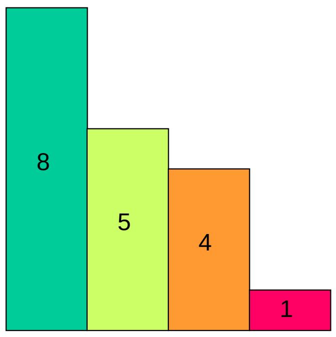


Figure 3-1. Task execution times.

If we decide to have as many machines (that is, workers) as tasks, we obtain an efficiency of 0.56 and an execution time of 8 units, as shown in Figure 3-2.

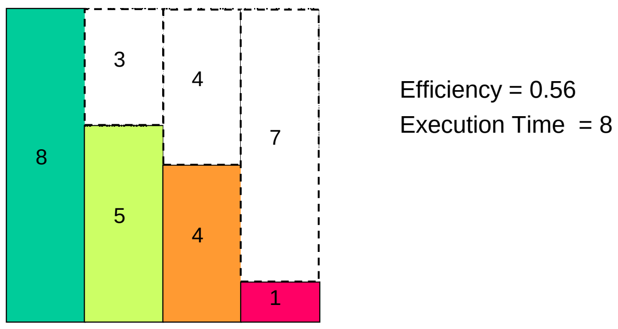


Figure 3-2. Assigning the tasks using 4 workers.

By choosing a different number of machines, two for example (see Figure 3-3), we get the maximal efficiency, namely 1, and an execution time of 9 units, that is, slightly larger than that obtained with as many machines as tasks.

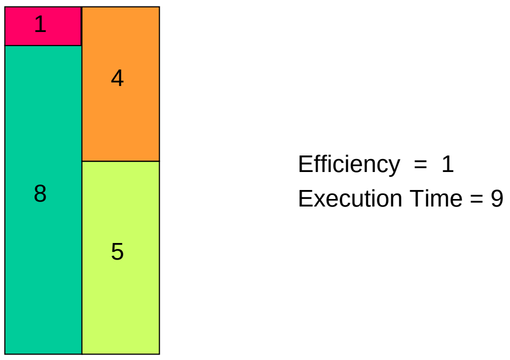


Figure 3-3. Assigning the tasks using 2 workers.

Now let us see an example of how the order in which tasks are assigned to machines (or workers) for execution also affect both execution time and efficiency. Consider the same four tasks for

Figure 3-1 and two machines. If tasks are assigned carelessly, as in Figure 3-4a, the first worker is idle 6 units of time, therefore efficiency is 0.75. In the second case, shown in Figure 3-4b, the amount of time workers spent doing useful work is equal to the amount of time workers had been able to execute work, therefore efficiency is 1, and the execution time is 9 units.

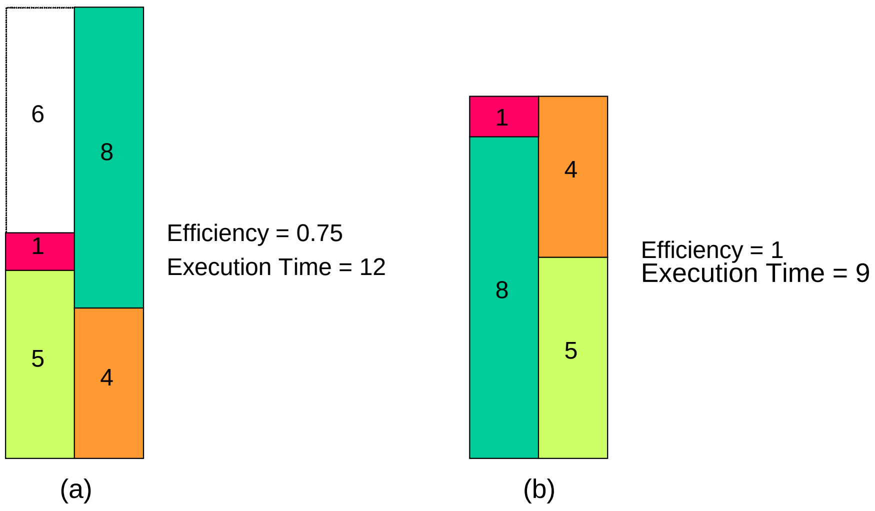


Figure 3-4. Assigning tasks to workers (a) carelessly (b) carefully.

In the master-worker model we consider, the exact execution time of the tasks is not known in advance. There is variability in task execution times from one cycle to another. Tasks will be executed for several cycles, until the completion of the problem. This end condition can be met after a fixed number of cycles, or until a convergence criterion is reached. There is one worker running per machine, so when we decide the number of workers to be considered for executing the application, we are deciding the number of machines. Both terms are used interchangeably. In this chapter, we consider clusters of homogeneous dedicated machines. In the next chapter, heterogeneous machines are considered, and in chapter 5, the assumption of dedicated machines is relaxed.

In next subsection, the scheduling problem is more formally introduced. The problem of determining the number of workers or machines is studied in chapter 4.

3.2.1 Scheduling Problem

The basic scheduling problem consists of assigning a set of *n* jobs J1, J2, ..., Jn to a set of *m* identical machines M1, M2, ...Mm [Gra66]. Each job Jj must be processed without interruption for a time *pj* > 0 on one of the *m* machines, each of which can process, at most, one job at a time. The objective is to reduce the total execution time, that is, minimizing makespan in an identical parallel machine environment. This well-known *minimum makespan* problem is NP-hard [Hoc97]. This means that there is no known efficient algorithm for solving the problem. Usually NP-hard problems are treated with heuristics.

Heuristics correspond to simple polynomial algorithms, which provide solutions quickly. The quality of the solution provided is bounded by using the concept of *approximation algorithms* [GGU72].

An approximation algorithm is polynomial, and is evaluated by the worst-case possible relative error over all possible instances of the problem. An algorithm is δ-approximation for a minimization problem if for every instance of that problem, it delivers a solution that is at most δ times the optimum. The closer δ is to 1, the better the solution.

Returning to the scheduling problem, different heuristics have been developed [Gra66]:

- The heuristic of assigning *any* job as soon as any machine becomes available, that is, assigning a *random* job, is a 2-approximation algorithm. This strategy is known as *List Scheduling.*
- LPTF (Largest Processing Time First): The heuristic that assigns the longest-remaining job to the first available machine is a 4/3-approximation algorithm.
- The complementary strategy, known as SPT assigns jobs from the shortest to largest processing times. This rule is optimal when minimizing the average for all job completion times.

3.2.2 Scheduling in Clusters of Distributed Machines

In load-sharing environments, where applications share resources, both applications and system components must be scheduled to achieve good performance. In [FK99] a classification of schedulers that contemplates different *performance goals* is presented:

- *High-Throughput Schedulers*: These are concerned with improving the performance of the system by optimizing the number of jobs that it executes in large amounts of time.
- *Resource Schedulers*: Their main goal is to coordinate access to a particular resource. This can be done by satisfying all the requirements to the resource (fairness criteria), or by optimizing the amount of resources used. In chapter 4, an algorithm for adjusting the number of workers to a particular master-worker application is presented. It is a

resource scheduler with the goal of achieving good efficiency, that is, good usage of the machine resource.

- *High-Performance Schedulers*: These promote the performance of individual applications by optimizing performance measures such as minimal execution time, speedup, etc. In section 3.3, a high-performance scheduling policy is presented, with the objective of minimizing application execution time on the available machines. The following subsection contains details about high-performance schedulers.

3.2.3 High Performance Schedulers

High-Performance schedulers determine an assignment of tasks, ordered in time, based on the rules of a scheduling policy, with the goal of optimizing application performance.

The first step for a high-performance scheduler for applications that are executed on load-sharing environments consists of selecting a set of resources on which to schedule application tasks. The number of machines that will be required of the *resource discover* is determined in the next chapter. The goal is to obtain a good efficiency without damaging execution time. Once the application has obtained machines, the scheduler then assigns application tasks to the computing resources.

3.3 Scheduling Policy

A considerable collection of scheduling algorithms has been proposed, with a practically infinite number of variants. Considering a homogeneous and dedicated environment, schedulers can be classified as follows:

- *With precise a-priori information*: Schedulers that use exact information on the execution time of the tasks or jobs before they are executed. These can be subdivided into:
    - *Adaptive*: The information used is obtained for each particular set of tasks. LPTF is an example of a strategy that uses information on task execution time to create a list of tasks sorted from the largest to the smallest. With this model, in our master-worker model with cycles, the execution time of the tasks should be known in advance before the execution of each cycle.
    - *Non-Adaptive*: In this case, the information is obtained once and used at each cycle. An example of this is when users supply a list with the order in which they want tasks to be executed.
- *Without precise a-priori information*: In this case, the scheduler does not have any a-priori information about task execution times. These are subdivided into:
    - *Adaptive*: These strategies use information collected and processed at execution time. They attempt to predict the behavior of tasks in the next iterations, taking into account the execution time obtained in previous iterations. The scheduling strategy that was simulated and compared with other policies falls into this category.
    - *Non-Adaptive*: The scheduler acts without considering the results obtained in previous executions. Random is an example of these strategies.

The kind of master-worker applications we are considering has a high degree of predictability, even though a wider set of cases was considered in the simulation study. It is possible to take advantage of this predictability in deciding the allocation of tasks to workers.

The proposed adaptive without precise a-priori information scheduling strategy employs a heuristic-based method that uses historical data on the behavior of the application. In particular, it dynamically collects statistics on the average execution time of each task and uses this information to determine the order in which tasks are assigned to processors. Tasks are sorted in decreasing order of their average execution time. They are then assigned to workers according to that order. At the beginning of the application execution, as no data is available regarding the average execution time of tasks, tasks are assigned randomly. This adaptive strategy has been called *Random & Average*, although the random assignment is only carried out once, when no historical data is yet available.

The next section presents the evaluation of the *Random & Average* policy by simulation, focusing on clusters of homogeneous machines that are available to the application the whole time.

3.4 Simulation Study

In this section, the performance of several scheduling strategies are evaluated with respect to the efficiency and execution time obtained when applied to scheduling master-worker applications on homogeneous systems. As we have stated in previous sections, we focus our study on a set of applications that are supposed to exhibit a highly regular and predictable behavior. We will test different scheduling strategies including both non-adaptive strategies that do not take into account any runtime information and adaptive strategies that try to learn from application behavior.

As a principal result from these simulation experiments, we aim to obtain information about how the studied strategies perform on average, and to ascertain certain bounds for the worst-case situations. For these reasons, it has been considered throughout the simulations that the number of processors is available over the whole application execution time (i.e. this would be the ideal case in which no machine suspensions occurs). This only implies a change in the expression used to evaluate the efficiency from that presented in chapter 2:

$$E = \frac{\sum_{i=1}^{n} T_{work,i}}{\sum_{i=1}^{n} T_{up,i}}$$

n being the number of workers,

$T_{work,i}$ being the amount of time that worker *i* spent doing useful work.

$T_{up,i}$ being the time elapsed since worker *i* is alive until it ends.

In a real scenario, if worker suspensions occurs, the efficiency of all policies will worsen, as will be shown in chapter 4.

3.4.1 Description of the Policies

The set of scheduling strategies used in the comparison were the following:

- **LPTF (Largest Processing Time First)**: This is a pseudo-optimal policy used for comparison purposed, easy to implement and fast in execution time. For each iteration, this policy first assigns the tasks with largest execution time. Before an iteration begins, tasks are sorted decreasingly by execution time. Then, each time a worker is ready to receive work, the master sends the next task on the list, that is, the task with the largest execution time. It is well known that *LPTF* is at least 4/3 of the optimum [Hall97]. This policy needs to know the exact execution time of the tasks in advance, which is not generally possible in a real situation, therefore it is only used as a sort of upper-bound in the performance achievable by the other strategies.
- **LPTF on Expectation**: This works in the same way as *LPTF*, but tasks are initially sorted decreasingly by the expected execution time. In each

iteration, tasks are assigned in that predefined order. If there is no variation in the execution time of the tasks, the behavior of this policy is the same as *LPTF*. This policy is non-adaptive, and represents the case in which the user has an approximately good knowledge of the application behavior, and wants to control the execution of the tasks in the order that he specifies. Obviously, it is possible for a user to have an accurate estimation of the distribution of times between the application tasks, but in practice, small variations will affect the overall efficiency because the order of assignment is fixed by the user at the beginning.

- **Random**: For each iteration, each time a worker is ready to get work, a random task is assigned. This strategy represents the case of a non-adaptive method that does not know anything about the application. In principle, it would obtain the worst performance of all the strategies presented here, therefore it will be used as a lower bound in the performance achievable by other strategies, as this strategy is a 2-approximation algorithm.

Table 3-1 shows a summary of the strategies considered and their main characteristics:

		Strategy Dynamics	
		Adaptive	Non-Adaptive
Information a-priori	Accurate	LPTF	LPTF on Expectation
	Non-accurate	Random & Average	Random

Table 3-1. Classification of the studied scheduling strategies

3.4.2 Simulation Framework

Figure 3-5 shows a simplified block diagram of the simulation framework implemented in this study. The simulator models a cluster of machines or processors (workers) with equal performance, a master-worker application and the scheduling algorithms previously mentioned.

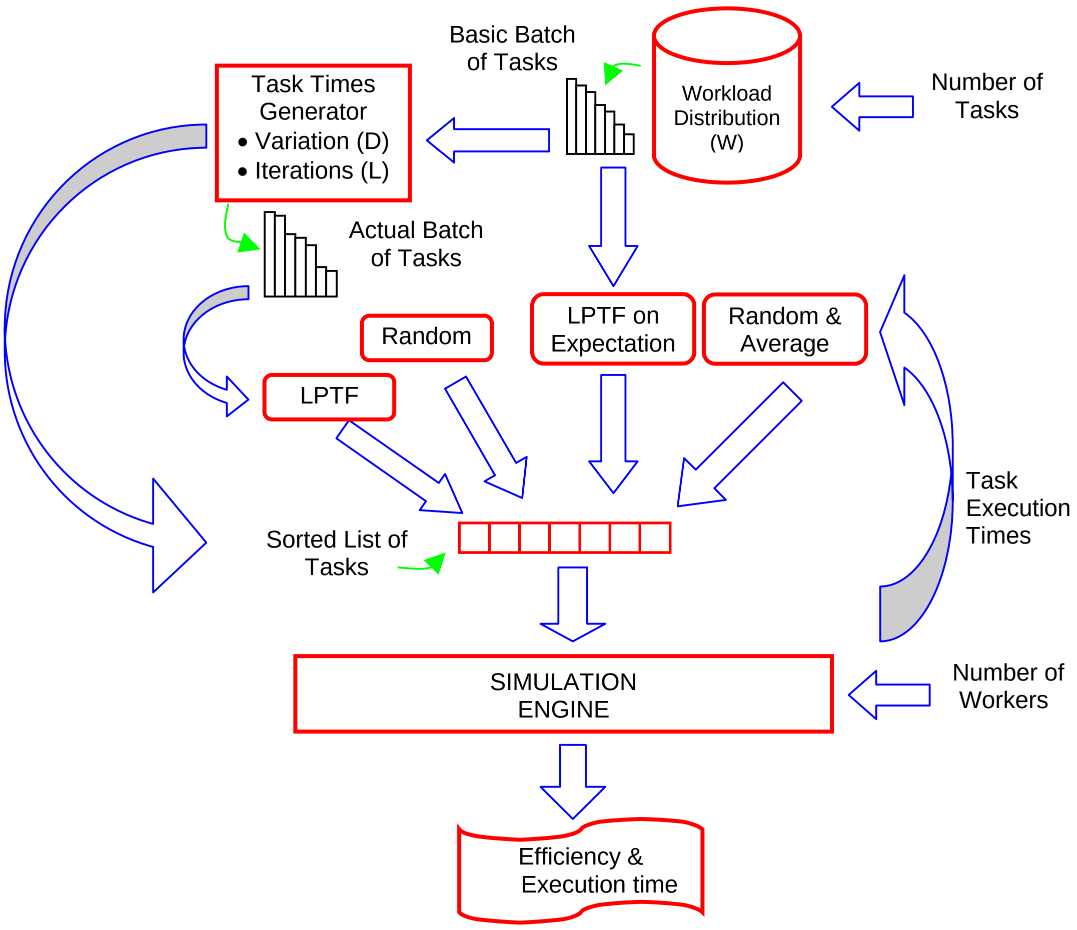


Figure 3-5. Simulation framework.

A master-worker application submitted to the cluster of workers consists of a *Basic Batch of Tasks* with a fixed distribution of times (or W*orkload*). We simulated the execution on *L* iterations of the master-worker application. For each iteration, an *Actual Batch of Tasks* was generated by applying a certain variation (D) of time to each basic batch task. Tasks were scheduled to workers according to the order of the *Sorted List* generated by each policy. The information used to generate the list was different in every case. The *LPTF* policy used the execution times derived from the *Actual Batch* (which corresponds to having a perfect knowledge of task execution times). The *LPTF on Expectation* strategy used the task times derived from the *Basic Batch*, i.e. it generated a single list that was used for the L iterations. *Random & Average* collected the task execution times once the execution of the *Actual Batch* was simulated, averaging task times and sorting the list according to these averages. The Random strategy generated the list in a random way for each *Actual Batch*. All tasks in the *Sorted List* were assigned to processors, as they become idle. Once a processor was assigned a task, it was marked as busy for a simulation time equal to the time of the assigned task. The execution of the *Actual Batch* was simulated for all scheduling policies before a new *Actual Batch* was generated. As an overall result from the simulation of a given master-worker application with a given scheduling policy, we obtained overall execution time and the efficiency for a given *Basic Batch*.

All described scheduling policies have been systematically simulated, to obtain efficiency and execution time, with all the possible number of workers ranging from 1 to as many workers as numbers of tasks, considering the following factors:

- ***Workload (W)*****:** The total amount of work (*TotalW*) is divided throughout the number of tasks that compose the batch, with the following scheme: 20% of the tasks contain *W%* of the total load, and the remaining 80% of the tasks contain (*TotalW-W*)*%* of the load. Workload values of 30%, 40%, 50%, 60%, 78% 80% and 90% were considered. A 30% workload would correspond to highly-balanced applications in which nearly all the tasks exhibit a similar execution time. On the contrary, a 90% workload would correspond to applications in which a small number of tasks are responsible for the largest amount of work. Moreover, the 20% tasks can have similar or different execution times. The same happens to the other 80% of tasks.

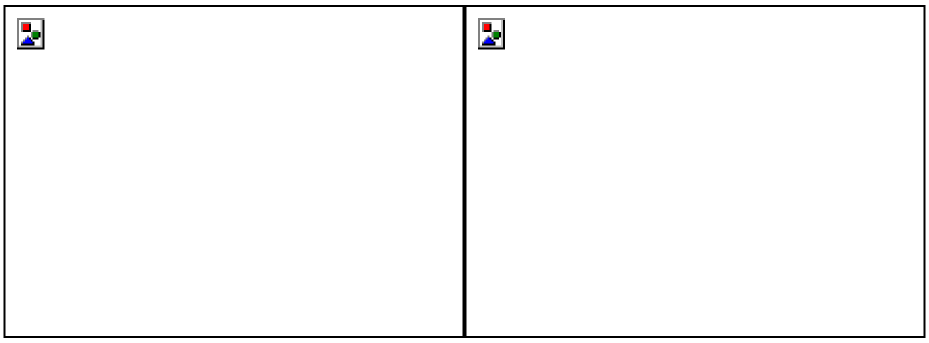
For each workload value, we have undertaken simulations with the four possibilities, shown in Table 3-2:

20% tasks	Remaining 80% of tasks	Notation in Figures (*i*-*i*)
Similar (0)	Similar (0)	0-0
Similar (0)	Different (1)	0-1
Different (1)	Similar (0)	1-0
Different (1)	Different (1)	1-1

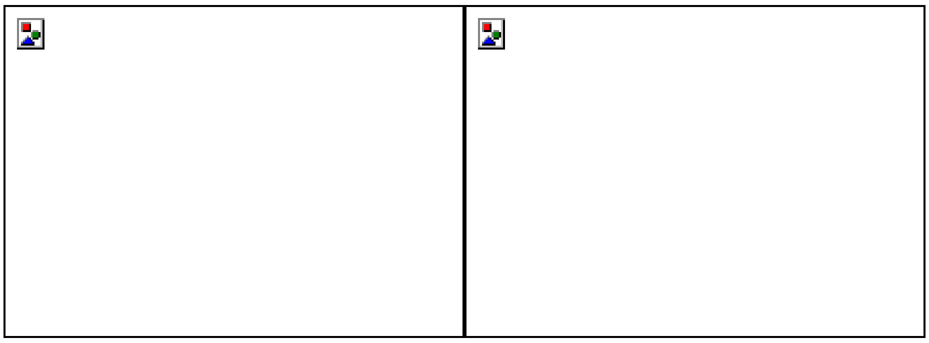
Table 3-2. Distribution of task times for a given workload.

Figure 3-6 shows the absolute execution times and cumulative execution times for 30 tasks with a workload of 60%, considering the 4 possibilities of Table 3-2. The absolute and cumulative execution times for a workload of 30% and 90% can be found in appendix A. The cumulative execution time for *n* machines shows the work percentage carried out when executing the *n* first tasks, sorted according the order in which they were generated.

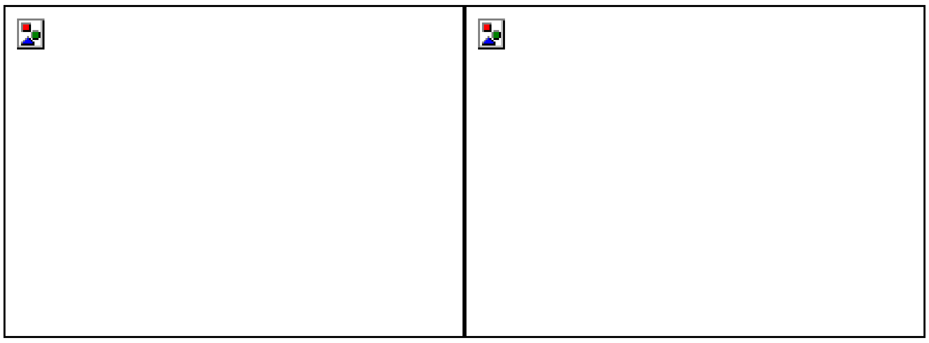
(a)



(b)



(c)



(d)

Figure 3-6. Workload distribution and workload cumulative percentages. (a) 60% 0-0, (b) 60% 0-1, (c) 60% 1-0, (d) 60% 1-1.

- ***Iterations (L)*****:** This represents the number of batches of tasks that are will be executed. The following values have been considered: 10, 35, 50 and 100.

- ***Variation (D)*:** From the workload factor, we determine the base execution times for the tasks. For each iteration a variation is then applied to each task's base execution time. Variations of 0%, 10%, 30%, 60% and 100% have been considered. When a 0% variation was used, the times of the tasks were constant over the different iterations. This case would correspond to very regular applications where task execution times are nearly the same in successive iterations. When a 100% variation was used, tasks exhibit significant changes in their execution time in successive iterations, corresponding to applications with highly irregular behavior. With a 100% variation, a task may double its execution time or become very small (but never have an execution time of 0, which would mean that this task disappears).

- ***Number of Tasks (T)*:** We have considered applications with 30, 100 and 300 tasks. These represent systems with a small, medium or large amount of tasks, respectively.

For each simulation scenario (fixing a certain value for *workload*, *iterations* and *variation*) efficiency and execution time have been obtained using all the possible values of workers from 1 to *Number of Tasks*.

3.4.3 Simulation Results

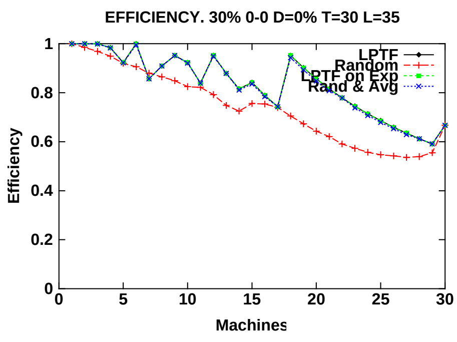
Although tests for all the commented values have been conducted, only those results that are the most relevant are presented in this section. The results for 30 tasks will be illustrated with figures, since they prove to be representative enough for the results obtained with a larger number of tasks. Moreover, those results with 30%, 60% and 100% deviation are emphasized, representing high, medium and low degrees of regularity. In real applications 100% deviation is not expected, but it nevertheless allows us to evaluate the strategies under the worst case scenario.

In the following subsections, some relevant result figures for both efficiency and execution time are presented. The X-axis always contains the number of workers. The Y-axis contains the efficiency and the execution time for efficiency figures and execution time figures, respectively. Five values (*W*, *i-i*, *D*, *T* and *L*) appear at the top of each graph. *W* stands for the workload, *i-i* describes the similarity of tasks according to Table 3-2, *D* stands for variation applied to task execution times at each iteration, *T* stands for the number of tasks and *L* for the number of iterations (cycles). We review the most relevant results obtained from the simulations. Appendix B presents the simulation results considering all the factors taken in account.

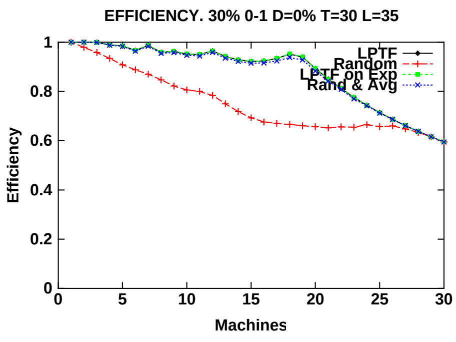
3.4.3.1 Effects on Efficiency

- ***Effect of workload (W) and Task Size (i-i)*****:** Figures 3-7 through 3-9 show the effect of varying the workload, considering 30% (Figure 3-7), 60% (Figure 3-8) and 90% (Figure 3-9) workload. In all cases, deviation was 0%, and the four possibilities for the execution times of all the largest tasks, as well as for all the smallest tasks, were considered (0-0, 0-1, 1-0 and 1-1). As expected, for large workloads, the number of workers that can usefully be busy is smaller than for small workloads. Moreover, when the workload is higher, efficiency declines faster. A large workload also implies a smoother curve in efficiency. It is important to point out that, in all cases, there is a point from which efficiency continuously declines. Before that point, small changes in the number of workers may imply significant and contradictory changes in efficiency, i.e. adding one worker may imply an improvement or a worsening in efficiency. In general, the *Random* policy tends to be insensitive to this change, as it decays in a constant way. There is an exception in the case of a workload of 90%, when all the largest and smallest tasks are of the same size (90% 0-0). For the other policies, this feature is stronger.

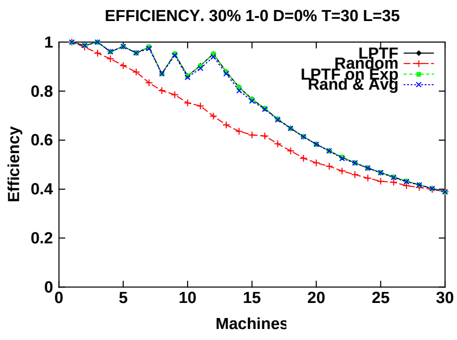
  With respect to the effect of task size (*i-i*), it is observed that the 20% of tasks executing the w% of the total work determine when the drop of efficiency begins. If they have the same execution time, the decay in efficiency is delayed. For example, in the case of 30% 0-1 (Figure 3-7b), efficiency begins to drop after 19 workers, while in the case of 30% 1-0 (Figure 3-7c), efficiency drops after 12 workers. The remaining 80% of the tasks have less influence, basically determining the smoothness of the efficiency curve. If the remaining 80% of the tasks have the same execution times, the efficiency curve has more peaks.



(a)



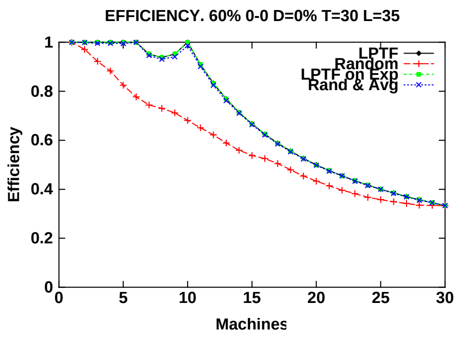
(b)



(c)

(d)

Figure 3-7. Effect of varying workload percentage and task size for a *W=30%.*
(a) 30% 0-0, (b) 30% 0-1, (c) 30% 1-0, (d) 30% 1-1.



(a)

(b)

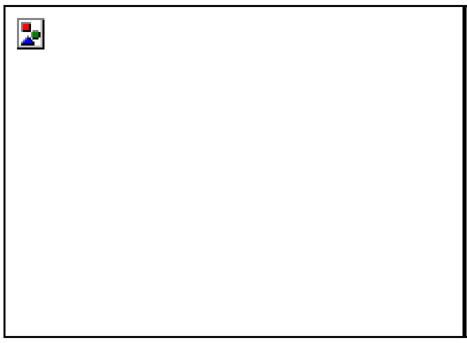
(c)

(d)

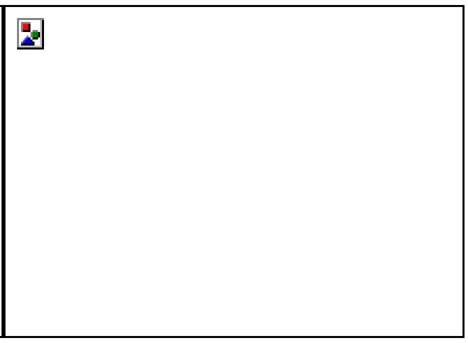
Figure 3-8. Effect of varying workload percentage and task size for a *W=60%.* (a) 60% 0-0, (b) 60% 0-1, (c) 60% 1-0, (d) 60% 1-1.

(a)

(b)



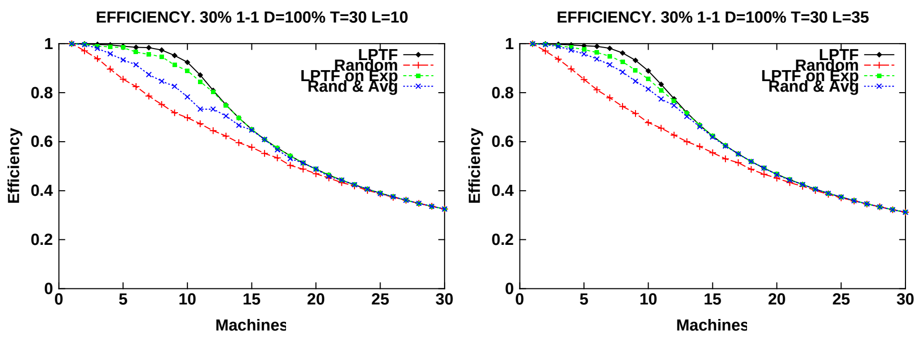
(c)



(d)

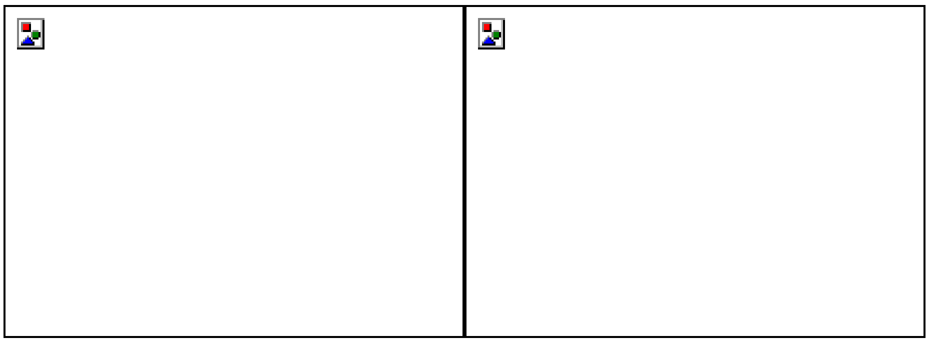
Figure 3-9. Effect of varying workload percentage and task size for a *W=90%*.
(a) 90% 0-0, (b) 90% 0-1, (c) 90% 1-0, (d) 90% 1-1.

- ***Effect of the number of iterations (L)*****:** The number of iterations (L) over which tasks are executed does not significantly affect efficiency for an adaptive strategy such as *Random & Average*. The other three policies are not affected at all by the number of iterations, as random does not use any information about previous iterations, and LPTF and LPTF on expectation use a-priori information that does not depends on the number of iterations. Figures 3-10 through 3-12 show the effect of varying the number of iterations, considering 30% (Figure 3-10), 60% (Figure 3-11) and 90% (Figure 3-12) workloads and 100% deviation. This is the case when the effect of the number of iterations is most significant. As can be seen when the number of iterations varies from 10 to 35, the gain in efficiency is less than 5%. When the number of iterations was greater than 35, no significant gain in efficiency was observed. The proposed strategy therefore achieves good efficiency without needing a large number of iterations to acquire precise knowledge of the application.



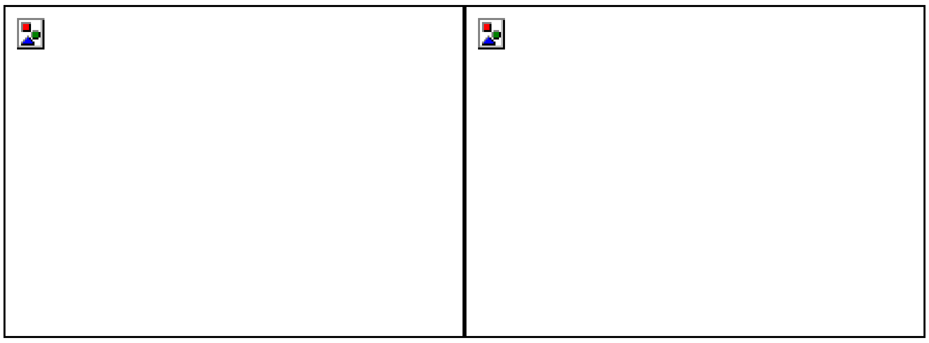
(a) (b)

Figure 3-10. Effect of varying the number of iterations for a *W*=30%. (a) L=10 (b) L=35



(a) (b)

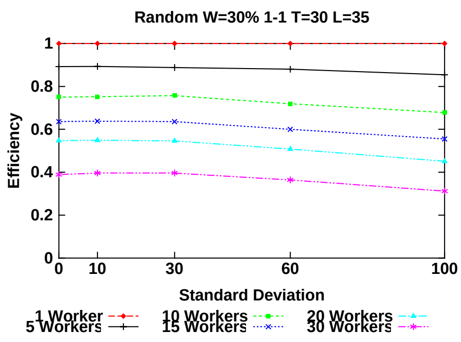
Figure 3-11. Effect of varying the number of iterations for a *W*=60%. (a) L=10 (b) L=35



(a) (b)

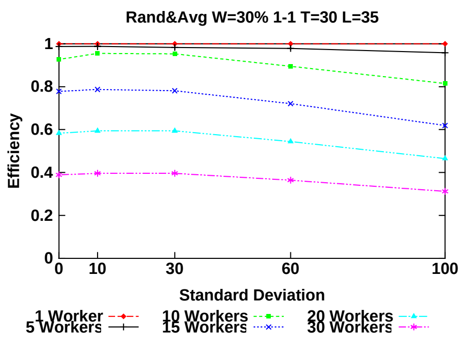
Figure 3-12. Effect of varying the number of iterations for a *W*=90%. (a) L=10 (b) L=35

- ***Effect of the variation (D):*** Figures 3-13 through 3-16 show the effect of varying the deviation for policies of *Random* (Figure 3-13), *Random & Average* (Figure 3-14), *LPTF* (Figure 3-15) and *LPTF on Expectation* (Figure 3-16), in the case of 30% and 90% workload. The presented graphs show the efficiency values for different deviation values. The X-axis contains deviation, with the Y-axis containing efficiency. Each curve shows what happens to efficiency by considering 1, 5, 10, 15, 30 and 30 workers, always having 30 tasks per iteration. This type of chart simplifies an understanding of the effect of variation, because a lower number of charts are needed. It is observed that when deviation is higher, efficiency declines more. But it is worth noting that it does not decline abruptly even when deviation is 100%. When deviation is increased, efficiency for the *Random & Average* policy declines less than that for the *Random* policy.



(a) (b)

Figure 3-13. Effect of varying deviation (D) for the *Random* policy. (a) *W*=30% (b) *W*=90%



(a)

(b)

Figure 3-14. Effect of varying deviation (D) for the *Random & Average* policy. (a) *W*=30% (b) *W*=90%

(a)

(b)

Figure 3-15. Effect of varying deviation (D) for the *LPTF* policy. (a) *W*=30% (b) *W*=90%

(a)

(b)

Figure 3-16. Effect of varying deviation (D) for the *LPTF on Expectation* policy. (a) *W*=30% (b) *W*=90%

3.4.3.2 Effects on Execution Time

In all the graphs, execution time is measured in terms of the relative differences with the execution time for the *LPTF* policy. The X-axis contains the number of machines or workers, with the Y-axis containing the percentage difference for execution time, with respect to the *LPTF* policy. For example, the point (X=9, Y=30), for the *Random* policy, means that with 9 workers, *Random* is 30% worse than *LPTF*.

- ***Effect of workload (W) and task size (i-i)*:** Figures 3-17 through 3-19 show the effect of varying the workload, considering 30% (Figure 3-17), 60% (Figure 3-18) and 90% (Figure 3-19) workload. In all cases, deviation was 0%, and we considered the four possibilities for the execution times, both for all the largest tasks as well as for all the smallest tasks (0-0, 0-1, 1-0 and 1-1). As can be seen, the *Random* policy always exhibits the worst execution time, particularly when an intermediate number of workers are used. *Random & Average* and *LPTF on Expectation* achieve an execution time comparable to the *LPTF* execution time.

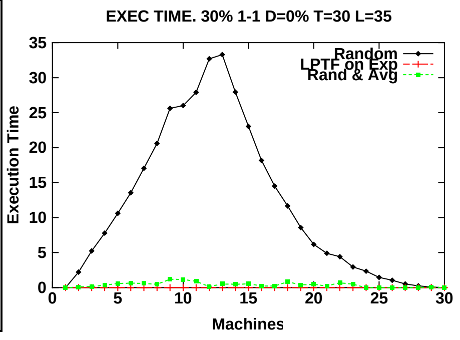
  In general, the pattern of the graphs is the following: when adding workers, the percentage difference with respect to the *LPTF* policy grows up to a certain point (where this difference is maximal), and then begins to decrease. The workload determines the number of workers associated with this point. With 90% 1-1 workload, this point corresponds to 5 workers, while with 30% 1-1, this point corresponds to 14 workers. The lower the workload, the higher the number of workers needed to reach this point.

  With respect to the effect of task size (*i-i*), it is observed that the remaining 80% of tasks determine the smoothness of the percentage difference curve. If they have the same execution time, then peaks are more notable. This effect is even stronger in the case of low workloads.

(a)

(b)

(c)



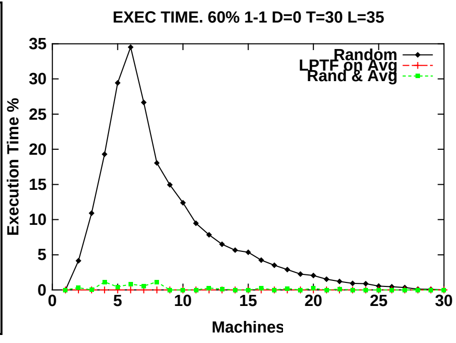
(d)

Figure 3-17. Effect of varying workload percentage and task size for a *W=30%.*
(a) 30% 0-0, (b) 30% 0-1, (c) 30% 1-0, (d) 30% 1-1

(a)

(b)

(c)



(d)

Figure 3-18. Effect of varying workload percentage and task size for a *W=60%.*
(a) 60% 0-0, (b) 60% 0-1, (c) 60% 1-0, (d) 60% 1-1

(a)

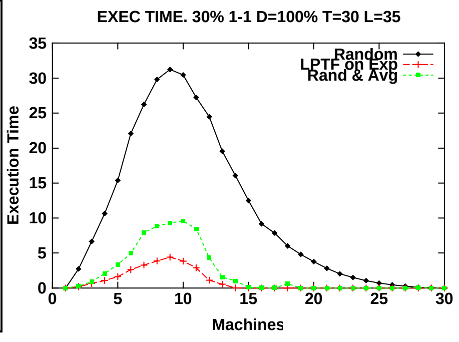
(b)

(c)

(d)

Figure 3-19. Effect of varying workload percentage and task size for a *W=90%.*
(a) 90% 0-0, (b) 90% 0-1, (c) 90% 1-0, (d) 90% 1-1

- ***Effect of the number of iterations (L)*:** As for efficiency, the number of iterations (L) over which tasks are executed does not significantly affect execution time for an adaptive strategy such as *Random & Average*. Figures 3-20 through 3-22 show the effect of varying the number of iterations, considering 30% (Figure 3-20), 60% (Figure 3-21) and 90% (Figure 3-22) workload and 100% deviation. This is the case when the effect of the number of iterations is the most significant. The execution time for the *Random & Average* strategy with respect to *LPTF* is slightly reduced when having a medium number of workers. The proposed strategy obtains a good execution time without needing a large number of iterations to acquire precise knowledge of the application.



(a) (b)

Figure 3-20. Effect of varying the number of iterations for a *W*=30%. (a) L=10 (b) L=35

(a) (b)

Figure 3-21. Effect of varying the number of iterations for a *W*=60%. (a) L=10 (b) L=35

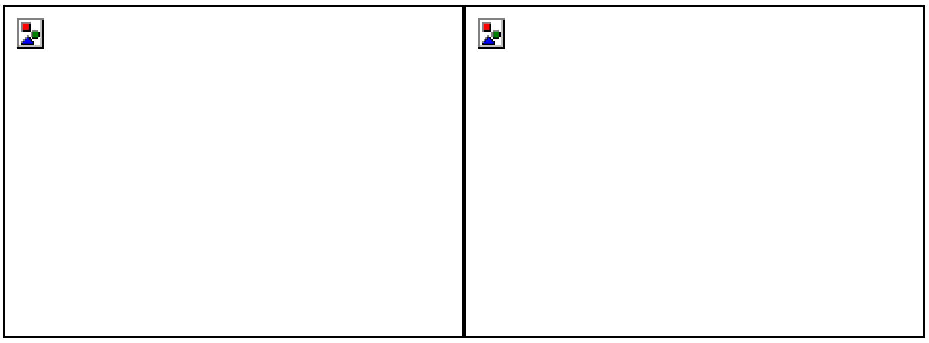

(a) (b)

Figure 3-22. Effect of varying the number of iterations for a *W*=90%. (a) L=10 (b) L=35

- ***Effect of the variation (D):*** Figures 3-23 through 3-26 show the effect of varying deviation for the *Random* (Figure 3-23), *Random & Average* (Figure 3-24), *LPTF* (Figure 3-25) and *LPTF on Expectation* (Figure 3-26) policies, in the case of 30% and 90% workload. The graphs presented show the execution time values for different variation values. The X-axis contains variation, whilst the Y-axis contains execution time. Each curve shows what occurs to execution time when considering 1, 5, 10, 15, 30 and 30 workers, always with 30 tasks per iteration. It is observed that when deviation is higher, execution time is greater, although *Random & Average* and *LPTF on Expectation* achieve an execution time comparable to the execution time of *LPTF*, even in the presence of a high variation in the execution time of the tasks.

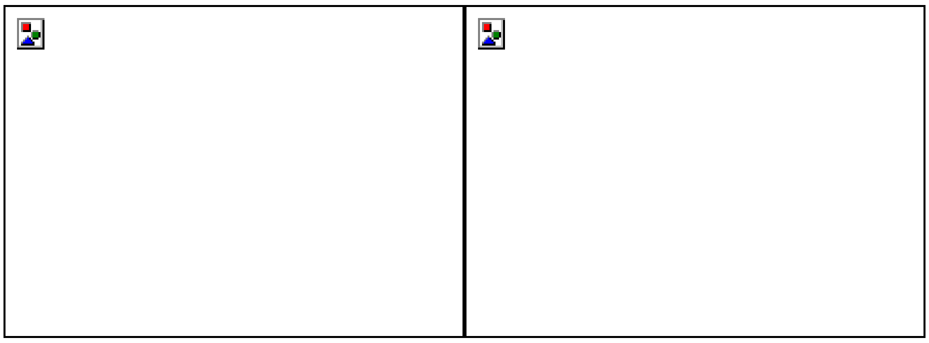
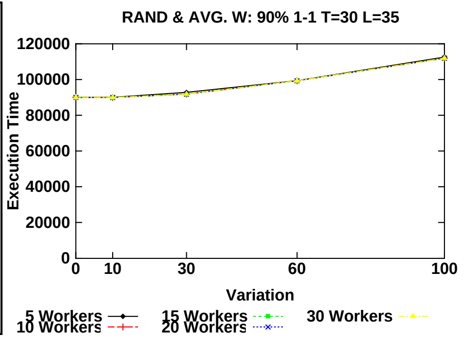

(a) (b)

Figure 3-23. Effect of varying deviation (D) for the *Random* policy. (a) *W*=30% (b) *W*=90%



(a) (b)

Figure 3-24. Effect of varying deviation (D) for the *Random & Average* policy. (a) *W*=30% (b) *W*=90%

(a) (b)

Figure 3-25. Effect of varying deviation (D) for the *LPTF* policy. (a) *W*=30% (b) *W*=90%

(a) (b)

Figure 3-26. Effect of varying deviation (D) for the *LPTF on Expectation* policy. (a) *W*=30% (b) *W*=90%

3.4.4 Discussion

We now summarize the main results that have been derived from all the simulations.

The number of iterations does not significantly affect either efficiency or execution time, although the *Random & Average* strategy needs certain iterations to *learn* about application behavior. The behavior of the policies was basically affected by the variation of execution times of the tasks in different iterations, by workload and by having significant differences among the execution times for the 20% of the tasks executing the w% of the total work.

Table 3-3 shows the efficiency bounds obtained for the previously described scheduling policies, always relative to *LPTF* policy. The first column contains the upper bound that is never surpassed in 95% of cases. The second column shows the upper bound for all cases, which always corresponded to 30% 0-0 workload with D=100%, that is, tasks without significant execution time differences and with high variance. As can be seen in Table 3-3, both *LPTF on Expectation* and *Random & Average* in most cases obtained an efficiency similar to that obtained by a policy such as *LPTF*, which uses perfect information about the application. Even in the worst case (scenarios in which all tasks have a similar execution time but a high deviation (100%)) loss of efficiency for both strategies was 17% approximately.

	Efficiency Bound in 95% of cases	**Worst Efficiency Bound**
Random	25,4 %	26,96 %
Random & Average	8,65 %	16,86 %
LPTF on Expectation	8,91 %	17,29 %

Table 3-3. Worst efficiency bounds for scheduling policies.

Similar results were obtained for execution time. *Random & Average* and *LPTF on Expectation* never performed worse than 7% in more than 95% of cases. Only in the presence of high variations were the differences increased to 16%, as is shown in Table 3-4.

	Efficiency Bound in 95% of cases	**Worst Efficiency Bound**
Random	29,46 %	35,7 %
Random & Average	7,16 %	16,24 %
LPTF on Expectation	6,85 %	12,33 %

Table 3-4. Worst execution time bounds for scheduling policies.

In 95% of cases, the execution time for the *Random* policy was always between 25% and 30% worse than *LPTF*.

In the light of the simulations carried out, we can conclude that a simple adaptive strategy such as *Random & Average* will perform very well, in terms of efficiency and execution time, in most cases. Even in the presence of highly irregular applications, overall performance will not significantly worsen. Similar results have been obtained for the *LPTF on Expectation* policy, but the use of this policy requires the user to have a good knowledge of the application.

In general, efficiency is high with a low number of machines but so, too, is execution time. Having a large number of machines results in a low execution time, but also in low efficiency. As this is an execution time-efficiency trade-off, we should therefore determine an adequate number of machines in order to obtain reasonable values for both execution time and efficiency.

In the next chapter, the *Random & Average* scheduling policy is applied in practice by using *Condor* and *MW*. It also contains a strategy, derived from the simulations, for determining the number of workers that must be allocated in order to obtain good efficiency as well as good execution time. Both homogeneous and heterogeneous environments are considered.

Scheduling of Master-Worker Applications on Homogeneous & Dedicated Clusters

Chapter 4

Self-Adjusting Scheduling for Master-Worker Applications

Abstract

This chapter presents a self-adjusting algorithm for dynamically determining the suitable number of workers for running a master-worker application. An implementation for homogeneous environments is first evaluated and its drawbacks are discussed. A second version that overcomes these drawbacks is later presented and evaluated, using an image thinning application. The chapter ends with a study of the necessary changes that need to be introduced in the algorithm in order to allow it to also work on heterogeneous systems, and its corresponding evaluation with the thinning application.

4.1 Introduction

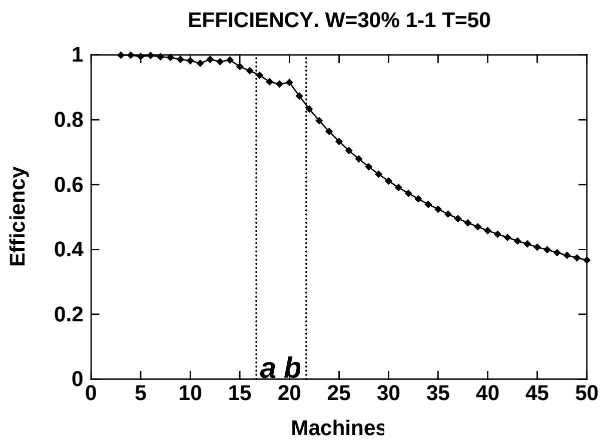
The design of the self-adjusting scheduling strategy is based on the following observation illustrated by Figures 4-1a and 4-1b, which shows the effect on efficiency and execution time, respectively, of an LPTF (Largest Processing Time First) policy. In general, the results in the previous chapter showed that for any given workload distribution, a similar scenario to the one depicted in Figure 4-1 is found. The Y-axis contains efficiency and execution time, respectively, and the X-axis contains the number of workers, T, (the maximum in this example is Max = 50). Three main intervals can be observed in this figure:

1. Interval [0,a] corresponds to the situation in which the application is running with a shortage of workers. Consequently, efficiency tends to be close to 1, but speedup is low. It is also important to point out that, in this interval, small changes in the number of workers may imply significant and contradictory changes in efficiency. The particular distribution of task times exhibited by the application may fit properly with a given number of workers, but adding more workers puts the task schedule “out of gear”, and, in some processors, results in more idle times.

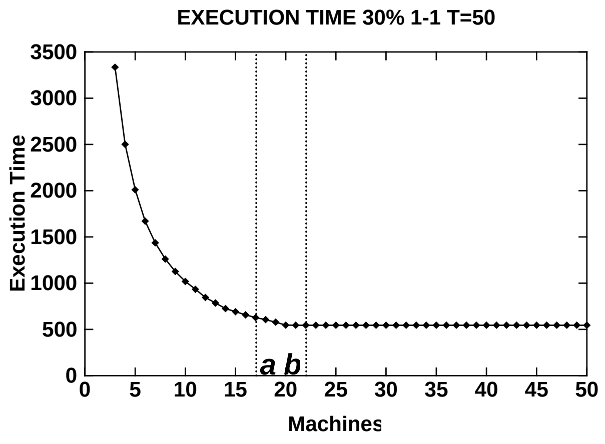
2. Interval [a, b] corresponds to the situation in which the application is using an *ideal* number of workers. Efficiency is high and speedup is also high. All the workers are doing useful work, and the application is close to its maximum parallelism utilization.

3. Interval [b, Max] corresponds to the scenario in which the application uses an excess of workers. At this interval, efficiency decays continuously as new workers are added to the application. Moreover, speedup is only slightly improved with new workers, because the global

execution time is dominated by the execution time of the largest task in each batch.



(a)



(b)

Figure 4-1. (a) Efficiency and (b) Execution time curves.

More formally, as was introduced in chapter 2, for any given workload distribution there is an optimal number of machines, denoted as *perfect number*, which should be allocated in order to obtain the best ratio between execution time and resource efficiency (denoted as EER, Execution-Efficiency Ratio).

$$EER(i) = \frac{ExecutionTime(i)}{\mathrm{Re}\,sourceEfficiency(i)}$$

EER(i): Execution-Efficiency Ratio for *i* machines.

Execution Time (i): Application execution time when *i* machines are used.

Resource Efficiency (i): Efficiency achieved when *i* machines are used.

The *perfect number* of machines exhibits the minimum value of that ratio (*perfect number* = min (*EER(i)*), $\forall$*i*).

Moreover, if the number of machines is close to the *perfect number*, the application still exhibits a good ratio between execution time and efficiency. The *ideal interval* [a,b] is the set of machines that exhibits this good ratio. Specifically, it is defined as the set of machines that have less than 1.1 the ratio presented by the *perfect number* of machines. Figure 4-2 shows the EER curve corresponding to the efficiency and execution time curves of Figure 4-1.

$$IdealInterval = \{\, i, \ \mid 1 \le i \le n \quad EER(i) \le ERR(PerfectNumber) * 1.1 \}$$

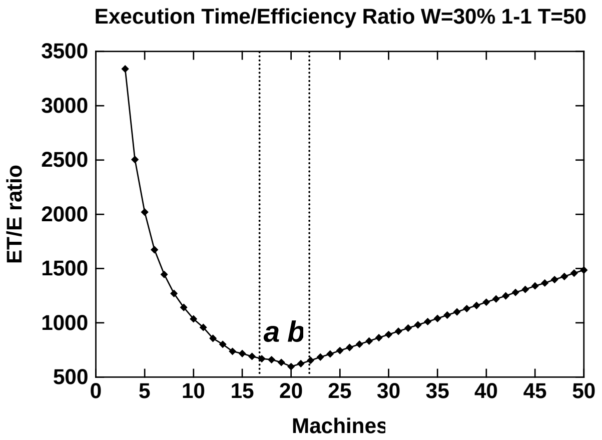


Figure 4-2. EER curve for a 30% workload.

Such a definition for the ideal interval has been adopted because, according to the experiments, it guarantees that efficiency is above 0.8 and that execution time is lower than 1.1 the time of executing tasks with as many machines as tasks. Additionally, most users of a system would consider both values acceptable for their applications. Obviously, the starting value and width of the *ideal interval* strongly depends upon workload distribution. Figure 4-2 illustrates the meaning of the *perfect number* and the *ideal interval* for a workload distribution in which almost all tasks exhibit a similar execution time (*W*=30%), considering 50 tasks (*T*=50). The perfect number of machines is 20, which corresponds to the minimum Execution-Efficiency ratio. At this point, efficiency is 0.9 and execution time is 1.01 the execution time obtained when each task is executed on a different machine, that is, with as many machines as tasks. The ideal interval is [18, 22], which corresponds to the machines that exhibit a ratio not greater than 10% the ratio exhibited by 50 machines.

Table 4-1 shows the number of machines corresponding to the perfect number, and the ideal interval for the different workloads considered in this work when there are 50 tasks and both the largest and smallest tasks have different values (case 1-1).

Workload	**Perfect Number**	**Ideal Interval**
30%	20	18-22
40%	14	13-15
50%	11	10-12
60%	9	8-10
70%	8	7-9
80%	7	6-8
90%	6	6-7

Table 4-1. Perfect number and ideal interval for different workloads.

The above-mentioned characteristics will be used to guide the design of a self-adjusting strategy. In the next sections, we will describe the design and implementation of such a strategy and its evaluation both in homogeneous and heterogeneous environments. The algorithm works in a dynamic way by adjusting the number of workers to a number belonging to the ideal interval. We will also describe the changes included into MW to support the self-adjusting algorithm and the corresponding experimentation that was carried out with the algorithm on both homogeneous and heterogeneous platforms.

4.2 Self-adjusting algorithm for homogeneous environments

The control of an application in an opportunistic environment implies, first, deciding in which order tasks will be executed, and, second, determining the number of machines that will participate in the computation. We will use the *Random & Average* scheduling policy described in the previous chapter to solve the first issue. The simulation results presented in the previous chapter justify both the need for measuring and recording execution times from previous iterations, and the need to build and maintain an average execution time list. In the next section, we present a strategy aimed at solving the problem of adjusting the number of machines at runtime.

4.2.1 Self-Adjusting Algorithm with Static Tables

The proposed strategy is based on the use of a table derived empirically from the simulation results presented in the previous chapter. It is also based on a characteristic common to opportunistic systems: machine allocations are more time-consuming than machine releases. In an opportunistic environment, the time cost incurred in the allocation of a machine is not negligible, because a negotiation protocol is usually carried out until a suitable machine is found and then allocated to a given application. This implies that requests for machine allocation will not be served immediately, or not even completely served at all if insufficient suitable machines are found. On the other hand, releasing machines back to the system is undertaken immediately, and essentially with no time cost.

Therefore, at the beginning of the execution of the master-worker application, as many workers as tasks per iteration (*N*) are created by the strategy for the application, that is, the maximum number of workers. Once the maximum number of machines at the start of an application is requested, machines are released in the case of the application running on an excess of machines, rather than obtaining a lower number of machines and then asking for more.

Then, at the end of each iteration, the adequate number of workers for the application is determined in a two-step approach. The first step quickly reduces the number of workers in an attempt to lower the number of workers to a value of number of machines that belongs to the ideal interval. The second step carries out a fine correction of that number. If the application exhibits a regular behavior, the number of workers obtained by the first step in the initial iterations will not change, and only small corrections will be undertaken by the second step.

The first step determines the number of workers according to the workload exhibited by the application. Table 4-2 is an experimental table that has been obtained from the simulations described in chapter 3. Workload (w) exhibited by the application is the factor that most influences the number of workers. Furthermore, the question of whether the 20% of the tasks executing the w% of the total work are of similar or different size is relevant when determining the number of workers. The difference in size of the remaining 80% of the tasks is not influent, nor are the other factors considered in the previous chapter.

For each workload distribution, Table 4-2 shows the number of workers needed in order to have a number of machines pertaining to the *ideal interval*, that is, to obtain efficiency greater than 80% and execution time less than 1.1 the execution time, when using *N* workers. As we have said in the previous chapter, these values would correspond to a situation in which resources are busy most of the time, while the execution time is not significantly degraded.

The first column contains the *workload*, as defined in chapter 3. The second and third columns contain the worker percentage with respect to the number of tasks for a given workload (w), in the cases where the 20% of the tasks that executes the w% of the total work to be done have similar or different executions times, respectively.

For example, if 20% of the tasks are carrying out 40% of the total work, then the number of workers to be allocated will either be *N**0,48 or *N**0,28. The former value will be used if the 20% of the tasks are similar; otherwise the later value is applied.

Although having a table like this implies a super-simplification of reality, this is needed in order to make the algorithm work with manageable parameters.

Workload	**%workers needed (20% tasks of similar size)**	**%workers needed (20% tasks of different size)**
30%	60%	40%
40%	48%	28%
50%	40%	22%
60%	34%	18%
70%	30%	16%
80%	26%	14%
90%	22%	12%

Table 4-2. Percentage of workers with respect to the number of tasks.

The fine correction step is carried out at the end of each iteration when both the workloads between iterations remain constant, i.e., the application remains in a same table entry, and the ratio between the last iteration execution time and the execution time with the current number of workers given by Table 4-2 is less than 1.1. This last value corresponds to a threshold of 10% over the execution time obtained with the number of machines according to the table. This means that the strategy allows variations in the execution time, when they never surpass 10% of the execution time obtained with the number of machines given by Table 4-2. The correction consists of diminishing the number of workers by one, if efficiency is less than 0.8, and observing the effects on execution time. If it becomes worse, a worker is added; however, never surpassing the value given by Table 4-2.

In the experimental system considered, the number of machines is handled cumulatively. This means that when *Nworkers* machines are requested and the application has already allocated *CurrentNworkers* machines, if (*Nworkers* > *CurrentNworkers*), only *Nworkers - CurrentNworkers* machines will be added to the application. Otherwise, *CurrentNworkers - Nworkers* machines will be released.

The complete algorithm to determine the number of workers that should be used is shown in Figure 4-3.

```
1.  Nworkers = Ntasks  /* We are in the first iteration */
    Workload = ∞, Efficiency = ∞, Execution Time = ∞
/* Next steps are executed at the end of each iteration i */
2.  Compute Efficiency, Execution Time, Workload and the Differences of the
    execution times of the 20% largest tasks.
3.  If (Workload of  iteration i != Workload of iteration i-1)
                    Set Nworkers = NinitWorkers according to Workload
                    and Differences of Table 4-2.
    else
                    if (Execution Time of it. i DIV
                            Execution Time using NinitWorkers)  <= 1.1)
                            if (Efficiency of iteration i < 0.8)
                                    Nworkers = Nworkers – 1
                    else
                            Nworkers = Nworkers + 1
```

Figure 4-3. Algorithm to determine *Nworkers*.

4.2.2 Experimental Evaluation with a Fibonacci Application

This section first describes the changes performed to MW in order to support both the generalized master-worker paradigm and the self-adjusting strategy, and then explains the experimentation performed with the aim of testing the self-adjusting strategy. After that, the results obtained when testing the effectiveness of the proposed self-adjusting scheduling strategy on a homogeneous environment are reported. We executed a synthetic master-worker application that performed the computation of Fibonacci series, which could serve as representative example of the generalized master-worker paradigm. The application was run on a platform first composed only of homogeneous machines.

4.2.2.1 Extended Version of MW

In its original implementation, MW supported one master controlling only one set of tasks. Therefore, the MW API has been extended to support:

- The proposed master-worker programming model with cycles.
- Scheduling policies. Both *Random & Average* and *Random* policies were included.
- The self-adjusting strategy. Useful information is also collected to allow dynamic adjusting the number of workers.

Figure 4-4 shows the changes introduced to MW. Components inside green circles were added. The *ToDo* queue contains the tasks corresponding to the current iteration (tasks behind these correspond to the tasks to be performed in subsequent iterations). This queue is sorted according to the desired scheduling policy (*Sched* circle), which in our case implies sorting the tasks according to the average execution time from previous iterations. The *Adjust* circle represents the computation of the number of workers requested. In order to carry out the adjusting and the scheduling, several statistics have to be collected (*Stats* circle). The blue *Widx* circles represent the information the master has on the workers participating in the computation. For a particular iteration, the *Running* and *Done* queues contains the tasks that are currently being executed and the tasks that have already been executed, respectively.

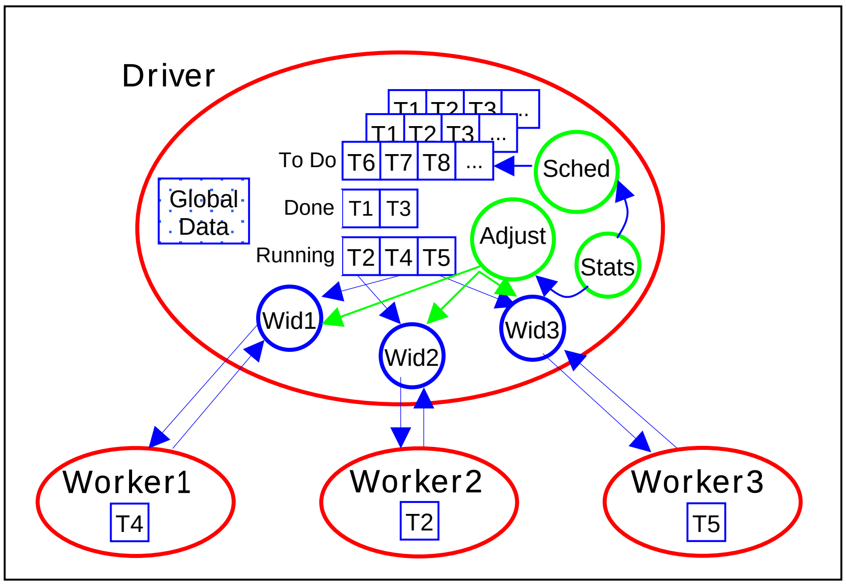


Figure 4-4. Extensions to MW

To create an MW application, the user has to implement certain virtual functions. In the Extended MW version, when creating the master process, the user needs to implement another pure virtual function: **global_task_setup**. There are also some changes in the functionality of certain others pure virtual functions:

- **global_task_setup()**: This initializes the data structures needed to keep the intermediate tasks results generated at the end of each iteration. This is called once, before the execution of the first iteration.
- **setup_initial_tasks (iterationNumber)**: The set of tasks created depends on the iteration number. Therefore, there are new tasks for each iteration, and these tasks could depend on values returned by the execution of

previous tasks. This function is called before each iteration begins, and creates the tasks to be executed in the *iterationNumber* iteration.

- **get_userinfo()**: The functionality of this function remains the same, but the user needs to call the following initialization functions:
  - **set_iteration_number (n)**: This is used to set the number of times tasks will be created and executed, that is, the number of iterations. If **INFINITY** is used to set the iterations number, tasks will then be created and executed until an end condition (or convergence condition) is achieved. This condition needs to be set in the function **end_condition()**, which is called before creating tasks for a new iteration.
  - **set_Ntasks (n)**: This is used to set the maximum number of tasks to be executed per iteration.
  - **set_task_retrive_mode (mode)**: This function allows the user to select the scheduling policy. It can be FIFO (**GET_FROM_BEGIN)**, based on a user key (**GET_FROM_KEY**), random (**GET_RANDOM**) or random and average (**GET_RAND_AVG**).
- **printresults (iterationNumber)**: This allows the results of the *iterationNumber* iteration to be printed.

In addition to the above changes, the *MWDriver* collects statistics on tasks execution times, workers' state (when they are alive, working and suspended), and on iteration beginning and ending. The new functions introduced are:

- **master_task_begin()** and **master_task_end()**: These are executed by the master every time a task begins and ends execution respectively, in order to get tasks execution time with the master clock.

- **begin_loop_iteration()** and **end_loop_iteration()**: These are executed at the beginning and end of each iteration, respectively, in order to measure the time it took to complete an iteration. *Begin_loop_iteration* also prepares all the data structures used to record all the time in which tasks are executing, suspended and alive. After calling *end_loop_iteration*, statistics are performed on the values recorded.

At the end of each iteration, function **UpdateWorkersNumber()** is called to dynamically adjust the number of workers accordingly, with regard to the self-adjusting algorithm explained in the previous section.

4.2.2.2 Experimentation Framework

Experiments were conducted using a platform composed of a dedicated Linux cluster running Condor, and a Condor pool of workstations at the University of Wisconsin. The total number of available machines was around 700 although the experiments were restricted to machines with Linux architecture (both from the dedicated cluster and the Condor pool). The execution of the applications was carried out using the services provided by Condor for resource requesting and detecting, determining information about resources and fault detecting. The execution of the applications was first carried out with a set of processors that do not exhibit significant differences in performance, so that the platform could be considered to be homogeneous. Figure 4-5 depicts the Condor configuration file part, showing the requirements imposed on the machines and the basic features of some machines obtained for an execution of the Fibonacci example. *W* stands for the worker, *Mem* for the memory and *LoadAvg* for the load average. This information was provided by *Condor*.

```
Universe = PVM

Executable     = master-fib

Requirements = ((Arch == "INTEL") && (OpSys == "LINUX")
             && (KFlops > 88000) && (KFlops < 93000))


Machine_count = 1..1

Queue
```

(a)

```
W: c20.cs.wisc.edu, KFlops = 89632, Mips = 609,
                    Mem = 511, LoadAvg = 0.92
W: c12.cs.wisc.edu, KFlops = 89182, Mips = 608,
                    Mem = 920, LoadAvg = 0.75
W: c03.cs.wisc.edu, KFlops = 89573, Mips = 609,
                    Mem = 920, LoadAvg = 0.83
W: c23.cs.wisc.edu, KFlops = 89935, Mips = 607,
                    Mem = 511, LoadAvg = 1.00
W: c25.cs.wisc.edu, KFlops = 89883, Mips = 609,
                    Mem = 511, LoadAvg = 1.01
W: c09.cs.wisc.edu, KFlops = 89205, Mips = 605,
                    Mem = 920, LoadAvg = 0.99
W: c24.cs.wisc.edu, KFlops = 89808, Mips = 608,
                    Mem = 511, LoadAvg = 1.00
W: c08.cs.wisc.edu, KFlops = 90421, Mips = 607,
                    Mem = 920, LoadAvg = 0.85
W: c14.cs.wisc.edu, KFlops = 90702, Mips = 607,
                    Mem = 920, LoadAvg = 0.92
W: c21.cs.wisc.edu, KFlops = 89680, Mips = 608,
                    Mem = 511, LoadAvg = 0.83
W: c18.cs.wisc.edu, KFlops = 90743, Mips = 606,
                    Mem = 511, LoadAvg = 0.94
W: c02.cs.wisc.edu, KFlops = 90224, Mips = 606,
                    Mem = 920, LoadAvg = 0.96
W: c20.cs.wisc.edu, KFlops = 89632, Mips = 609,
                    Mem = 511, LoadAvg = 1.00
W: c36.cs.wisc.edu, KFlops = 91557, Mips = 605,
                    Mem = 511, LoadAvg = 0.92
```

```
W: c32.cs.wisc.edu, KFlops = 89431, Mips = 609,
                    Mem = 511, LoadAvg = 0.99
W: c41.cs.wisc.edu, KFlops = 90062, Mips = 608,
                    Mem = 511, LoadAvg = 0.99
W: c51.cs.wisc.edu, KFlops = 89898, Mips = 608,
                    Mem = 511, LoadAvg = 1.00
W: c33.cs.wisc.edu, KFlops = 89847, Mips = 609,
                    Mem = 511, LoadAvg = 1.00
W: c54.cs.wisc.edu, KFlops = 89927, Mips = 609,
                    Mem = 511, LoadAvg = 1.00
W: c52.cs.wisc.edu, KFlops = 89426, Mips = 609,
                    Mem = 511, LoadAvg = 1.00
```

(b)

Figure 4-5. (a) Condor configuration file, (b) Machines obtained in a sample execution.

The application used in the experiments conducted in order to evaluate the self-adjusting strategy with static tables was composed of 28 synthetic tasks at each iteration. The number of iterations was fixed at 35, so that the application was running in a steady state most of the time. Each synthetic task performed the computation of a Fibonacci series. The length of the series computed by each task was randomly fixed at each iteration in such a way that the variation in execution time of a given task in successive iterations was 30%. Experiments were carried out with two synthetic applications that exhibited a workload distribution of 30% and 50% approximately. In the former case, all large tasks exhibited a similar execution time. In the latter case, the execution time of larger tasks exhibited significant differences. These two synthetic programs can be representative examples for master-worker applications with a highly-balanced and medium-balanced distribution of workload between tasks, respectively. Figure 4-6 shows, for instance, the average and deviation time for each of the 28 tasks in the master-worker with a 50% workload.

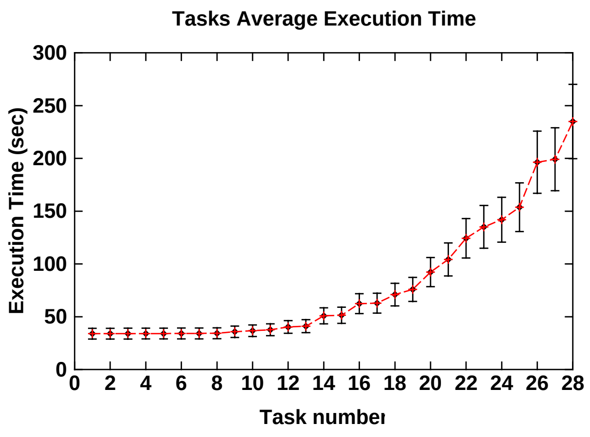


Figure 4-6. Tasks execution times.

Different runs on the same programs generally produced slightly different final execution times and efficiency results, due to the changing conditions in the opportunistic environment. Hence, average-case results are reported for sets of three runs.

4.2.2.3 Evaluation of the Self-Adjusting Strategy with Static Tables

The goal of this initial experimentation was to validate the effectiveness of the self-adjusting strategy on a homogeneous environment.

Tables 4-3 and 4-4 show the efficiency, execution time (in seconds) and speedup obtained by the execution of the master-worker application with 50% and 30% workload, respectively. The results obtained by the self-adjusting scheduling strategy are shown in bold in both tables. In addition to these results, the results obtained when a fixed number of processors were used during the whole execution of the application are shown. In particular, a fixed number of processors of n=28, n=25, n=20, n=15, n=10, n=5 and n=1 were tested. In all cases, the order of execution was carried out according to the sorted list of average execution time (as described in chapter 3 for the

Random & Average policy). The execution time for n=1 was used to compute the speedup of the other cases.

#Workers	1	5	**8**	10	15	20	25	28
Efficiency	1	0,94	**0,80**	0,65	0,43	0,33	0,28	0,22
Exec. Time	80192	16669	**12351**	12365	13025	12003	12300	12701
Speedup	1	4,81	**6,49**	6,49	6,16	6,68	6,52	6,31

Table 4-3. Experimental results in the execution of a master-worker application with 50% workload using the *Random and Average* policy.

#Workers	1	5	10	15	**18**	20	25	28
Efficiency	1	0,88	0,78	0,79	**0,74**	0,65	0,65	0,65
Exec. Time	31836	7030	3960	2860	**2753**	2598	2309	2218
Speedup	1	4,52	8,03	11,13	**11,56**	12,25	13,78	14,35

Table 4-4. Experimental results in the execution of a master-worker application with 30% workload using the *Random & Average* policy.

The first results shown in Tables 4-3 and 4-4 are quite significant, as they prove that the self-adjusting scheduling strategy was able, in general, to achieve a high efficiency in the use of resources, while speedup was not significantly degraded. Improvement in efficiency can be explained because the self-adjusting strategy tends to use a small number of resources with the aim of avoiding idle time in workers that compute short tasks. In general, the larger the number of processors, the larger the idle times incurred by workers in each iteration. This situation is also more remarkable when the application workload is more unevenly distributed among tasks. Therefore, for a given number of processors, the largest loss of efficiency was normally obtained in the application with a 50% workload. All these results were expected, in accordance to the simulation study described in the previous chapter; we can therefore affirm that, even with all the simplifications incurred, the simulation model represents what actually happens in a real environment, when executing a real application.

It can also be observed in both tables that the self-adjusting scheduling strategy generally obtained an execution time that was similar or even better than that obtained with a larger number of processors. This result basically reflects the opportunistic nature of the resources that were used in the experiments. The larger the number of processors allocated, the larger the number of task suspensions and reallocations incurred at run time. The need to terminate a task prematurely, when the user reclaimed the processor, normally prevented the benefits in execution time obtained by the use of additional processors. Therefore, a conclusion from the results is that, reducing in the number of processors allocated to an application running in an opportunistic environment is good, not only because it improves overall efficiency, but also because it avoids side effects on the execution time caused by suspensions and reallocations of tasks. The cost of losing a machine during execution time will be evaluated in the next chapter.

As is perhaps to be expected, the best performance was normally obtained when the largest number of machines were used, although better machine efficiencies were obtained with a smaller number of machines. These results may seem to be obvious, but it should be noted that self-adjusting scheduler strategy only used statistical information collected at runtime, and the execution of the application is influenced by the effects of resource obtaining, local suspension of tasks, task reassume and dynamic redistribution of load.

4.2.2.4 Comparison of *Random* and *Random & Average* scheduling policies.

An additional set of experiments was carried out in order to evaluate the influence on the order of task assignment. The results obtained when master-worker applications with 50% and 30% workload were scheduled using a *Random* policy. In this policy, when a worker becomes idle, a random task from the list of those pending is chosen and assigned to it. As can be seen in Table 4-5 and 4-6, the order in which tasks are assigned has a significant impact when a small number of workers is used. For less than 15 processors, the *Random & Average* policy performs significantly better than the *Random* policy, both in efficiency and in execution time. When 15 or more processors are used, differences between both policies were almost negligible. This fact is because, when the *Random* policy has a large number of available processors, the probability of assigning a large task at the beginning is also large. Therefore, in these situations, the assignments carried out by both polices are likely to follow a similar order.

Random							
#Workers	1	5	10	15	20	25	28
Efficiency	1	0,80	0,56	0,40	0,34	0,26	0,26
Exec. Time	80192	20055	14121	13273	12153	12109	12716
Speedup	1	4,00	5,68	6,04	6,59	6,62	6,31
Random & Average							
#Workers	1	5	10	15	20	25	28
Efficiency	1	0,94	0,65	0,43	0,33	0,28	0,22
Exec. Time	80192	16669	12365	13025	12003	12300	12701
Speedup	1	4,81	6,49	6,16	6,68	6,52	6,31

Table 4-5. Experimental results for *Random* and *Random & Average* scheduling with a master-worker application with 50% workload.

Random							
#Workers	1	5	10	15	20	25	28
Efficiency	1	0,87	0,82	0,81	0,7	0,66	0,63
Exec. Time	32241	7477	4807	2932	2679	2291	2105
Speedup	1	4,31	6,7	10,99	12,03	14,07	15,31
Random & Average							
#Workers	1	5	10	15	20	25	28
Efficiency	1	0,88	0,84	0,82	0,65	0,65	0,65
Exec. Time	31836	7030	3960	2860	2598	2309	2218
Speedup	1	4,52	8,03	11,13	12,25	13,78	14,35

Table 4-6. Experimental results for *Random* and *Random & Average* scheduling with a master-worker application with 30% workload.

With a number of workers close to the number of tasks, both strategies present a similar behavior, with differences in execution time and efficiency values that are less than 3%, produced because of the underlying system.

4.2.3 Self-Adjusting Strategy with Dynamic Information

The self-adjusting strategy considered up to now has certain drawbacks:

- First, the computation cost incurred at runtime in evaluating the workload distribution exhibited by an application, as a means of determining the appropriate table entry. This implies sorting a list of task execution times at the end of each iteration.
- Secondly, the sensitivity of the method to small variations in task execution times in successive iterations. This problem resulted in scenarios in which some machines were released and immediately reclaimed, because the application workload was oscillating between two table entries.

A variation of this strategy, which is described in this section, tries to overcome the problems related to the allocation of workers by, on the one hand, being more conservative in releasing machines and, on the other hand, trying to approach the *ideal number* of machines in a more gentle way, once the application runs with a number of machines close to the upper limit of the ideal interval (*b* point in Figure 4-1). The assignment of tasks to workers has not changed from the previous version of the self-adjusting strategy.

At the end of each iteration the Self-Adjusting algorithm (shown in Figure 4-7) computes the number of workers (*Nworkers*) that should be allocated to the application using two main criteria. Table 4-7 shows the meaning of the principal variables used:

Variable	**Meaning**
Nworkers	Number of workers to be required in the next iteration.
asp	Application achievable speedup.
ItExecutionTime	Execution time obtained in the previous iteration.
ItEfficiency	Efficiency obtained in the previous iteration.
ItMinTaskExecTime	Smallest task execution time in the previous iteration.
ItMaxTaskExecTime	Largest task execution time in the previous iteration.

Table 4-7. Meaning of the variables used in the self-adjusting algorithm.

1. First, the *AdjustBySpeedup* function computes *Nworkers* by evaluating *asp* (achievable speedup), defined as the ratio between the execution time of the whole application (by adding all the time tasks) and the execution time of the largest task (*ItMaxTaskExecTime*), obtained in the last iteration.

   $$asp = \frac{\sum_{i=1}^{MaxTasks} ETi}{ETj} \quad j, \mid \forall i \; 1 \le i \le MaxTasks \, , ET_j > ET_i$$

   The upper limit of the ideal number of machines corresponds to *asp*. Therefore, the number of workers (*Nworkers*) is set to $\lceil asp \rceil + 1$. We ask for one machine more than asp, so as to avoid situations in which machines are released at one iteration and claimed back at the next. If this represents an excess of machines, the next step in the algorithm will correct it. This procedure is always used when the application has not allocated all the machines requested in a previous iteration. It is possible that the requirement of workers has changed from one iteration to the next. Therefore, *asp* is recomputed to check whether the previous requirement of workers is still valid or not.

2. When the application is running with the number of workers previously computed in *Nworkers*, the adjusting criterion to update *Nworkers* is based on two metrics: execution time (*ItExecutionTime*) and efficiency (*ItEfficiency*) obtained in the last iteration. If the execution time is greater than the execution time of the largest task plus a given threshold, then one more worker is allocated. The threshold has been fixed as the maximum between the time of the smallest tasks (*ItMinTaskExecTime*) and 15% of the largest tasks. This threshold was empirically fixed as it proved able to detect most of the situations in which the application due to lack of workers is not exploiting all its parallelism, and it does not yield unstable situations in which workers are claimed and released too frequently. When the second metric is applied, a machine is released when efficiency is smaller than 0.8.

```
1.  In the first iteration Nworkers = Ntasks
For next iterations (While convergence condition is not met) {
2.  Compute ItEfficiency, ItExecutionTime, ItMinTaskExecTime,
    ItMaxTaskExecTime, CurrentNworkers.
3.  if (CurrentNworkers <  Nworkers )  // We have not got the number of
                                                 workers needed
       Nworkers = AdjustBySpeedup()
    else
       if (ItExecutionTime >  (ItMaxTaskExecTime +
                   MAX(ItMinTaskExecTime, 15%(ItMaxTaskExecTime))))
            Nworkers = Nworkers + 1
       else
            if (ItEfficiency < 0.8)
                      Nworkers = Nworkers – 1
  }
```

Figure 4-7. Algorithm to determine *Nworkers*

It is important to point out that the criteria described in point 2, above, are applied only when the application runs during a whole iteration with a stable number of machines. In this way, metrics obtained under unstable situations are not considered. This means that when a new machine that was previously requested is allocated in the middle of an iteration, and used for executing pending tasks, temporarily contradictory results in efficiency or execution time metrics may be produced. This refinement is not shown in Figure 4-7, for the sake of simplicity.

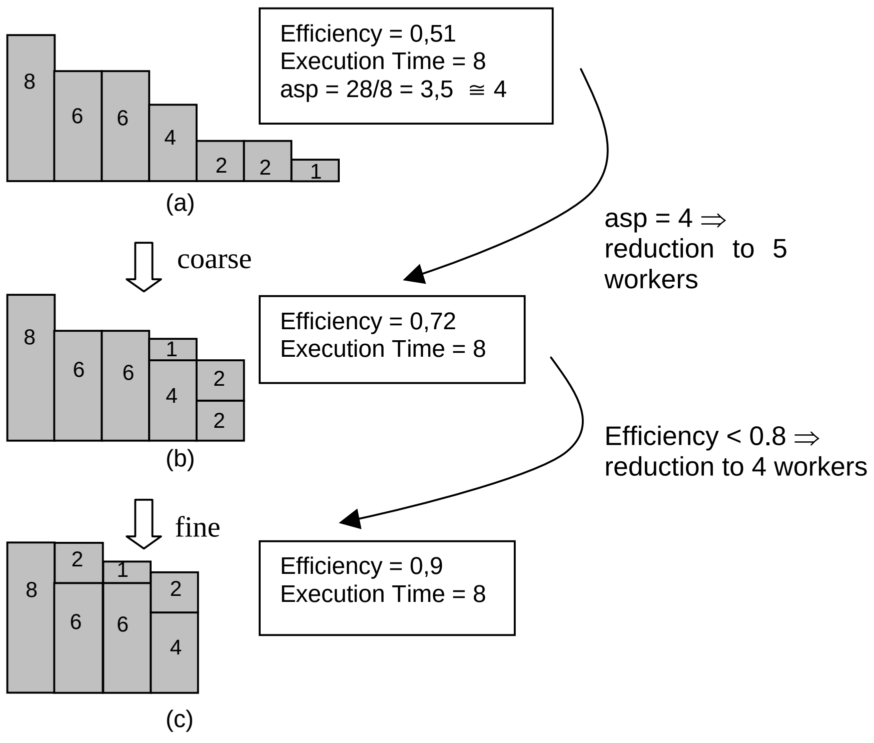


Figure 4-8. Worker allocation. (a) As many workers as tasks, (b) coarse adjust, (c) fine adjust.

Figure 4-8 shows a simple example considering 7 tasks. The number inside each box indicates the corresponding task execution time. Initially, as many workers as tasks per iteration (*N*) are allocated for the application, as is exemplified in Figure 4-8a. Later, at the end of the first iteration, that is, after obtaining the execution times of the tasks in the previous iteration, the number of workers is adjusted to the achievable speedup corresponding to the application (Figure 4-8b). This step is also repeated at the end of every iteration that finishes without the requested number of machines. When the application has obtained the requested number of machines, machines are added if the execution time is high, or released if efficiency is low (Figure 4-8c)

The self-adjusting algorithm is based on two main assumptions:

1. Application parallelism will not exhibit drastic increases over time. This means that the application will need the largest number of processors at the initial stages of the execution and, later, that the number will be kept nearly constant, or will gradually decay. It is assumed that the application will not exhibit scenarios in which parallelism alternates phases with significant decreases followed by phases with drastic increases. This assumption is relevant, given the cost for allocating machines in many environments such as the opportunistic one used in these experiments: requests for new machines are not serviced immediately by the system, while released machines are immediately lost. The negative effect of parallelism fluctuations could eventually be alleviated by releasing processors in a more conservative way than that presented in the current algorithm, in order to anticipate later increases in parallelism.

2. The value of a*sp* obtained after the first iteration will not change significantly in the near future. This assumption can be violated if the variation of task times is significantly large, although it may not imply a change in *asp*. In practice, the swift reduction carried out initially by our algorithm may be too drastic. A simple alternative would be to reduce the initial number of processors in the interval [*asp*, N], by using a more gradual technique such as a binary search or a golden sections method [NVZ96].

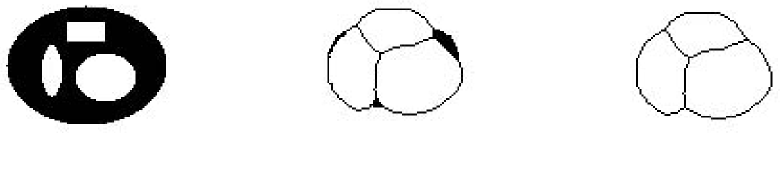
The first assumption was never violated in the experiments performed, and it was also found that the time value of *asp* was stable before the system allocated all the initially-requested machines. Therefore, none of the extensions mentioned above to the basic algorithm were implemented, although they could be included without difficulty.

Another extension that may easily be included with our strategy could overlap the scheduling phase computed in the master machine with the execution of workers. This would compensate time incurred in the computation of *Nworkers* when the number of workers is relatively large. However, in the experiments carried out, this time was negligible and it was not necessary to include this extension.

4.2.4 Experimental Evaluation with an Image Thinning Application

In this section, the results obtained with the aim of testing the effectiveness of the improved self-adjusting scheduling strategy on a homogeneous environment are reported. The environment considered is the same as that used for the Fibonacci application, described in section 4.2.2.2.

Thinning is the operation performed to an image in order to obtain its skeleton, that is, the basic lines and possibly an idea of the width of the lines in the original image. The thinning algorithm for binary images utilized was adapted from the AFP3 (Fully Parallel Algorithm) described in [GH92]. Figure 4-9 shows an example containing both the original and the resulting images.



(a) (b) (c)

Figure 4-9. Thinning (a) Original image, (b) 10 iterations later, (c) 36 iterations later.

The application works as shown in Figure 4-10. Initially, the master divides the image into *M* horizontal parts. Each part contains the pixels of a piece of the image, plus border pixels from neighboring parts. One task is created to compute the thinning operation of one part, which basically consists of deleting pixels. When tasks are assigned to workers, the thinning operation of the corresponding part is carried out. At the end of each iteration, workers send the image back to the master, which updates the border pixels. If there are no more pixels to delete, the part achieves the local convergence criterion, and finishes. When all the parts have finished, the global convergence criterion is then met, the skeleton image is reconstructed by combining the parts in order, and the application finishes.

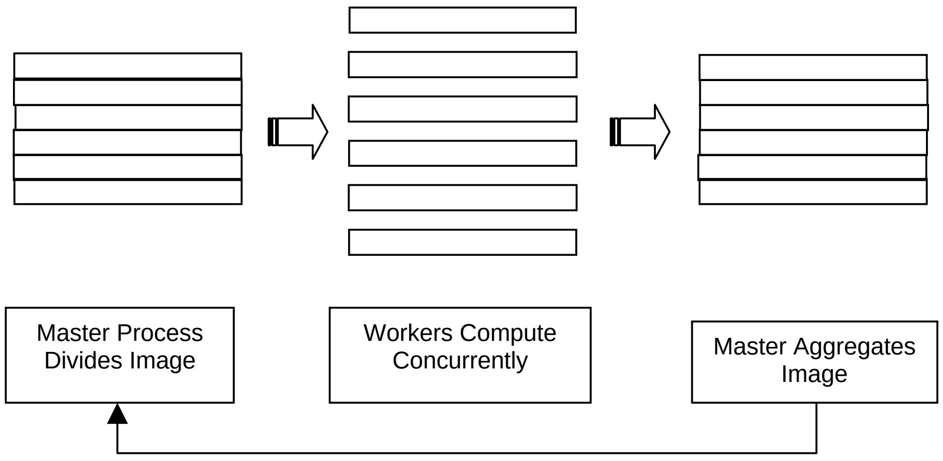


Figure 4-10. Image thinning as a master-worker application

This application exhibits two characteristics that make its use attractive for evaluating a self-adjusting strategy. First, tasks corresponding to different parts of the application usually exhibit different execution times. Tasks that are assigned complex parts of the image spend more time than tasks that deal with simple parts of the image. Therefore, a self-adjusting strategy should be able to schedule short tasks together to the same worker, and relinquish spare workers. Secondly, the execution time of each task gradually decreases as the image thinning approaches convergence. Again, the self-adjusting strategy should also be able to reduce the number of workers as the execution time of converging tasks is close to zero.

The thinning application was run with 3 images: *Figures*, *Letters* and *Boy&Ball* (shown in Figure 4-11a, 4-11b and 1-11c). Images were initially divided into 8, 16 and 32 parts, which corresponded to the initial set of tasks created at the initial iteration. The number of iterations until thinning convergence was 92, 97 and 105 for the *Figures*, *Boy&Ball* and *Letters* images, respectively. The size of the images was enlarged so that the execution time of the largest task was initially in the range of 50 seconds when images were divided into 8 parts. Communication time was negligible, with respect to computation time.

Execution times obtained when executing the same program are not the same because of the opportunistic environment on which they are run. Therefore average-case results are reported for sets of three runs.

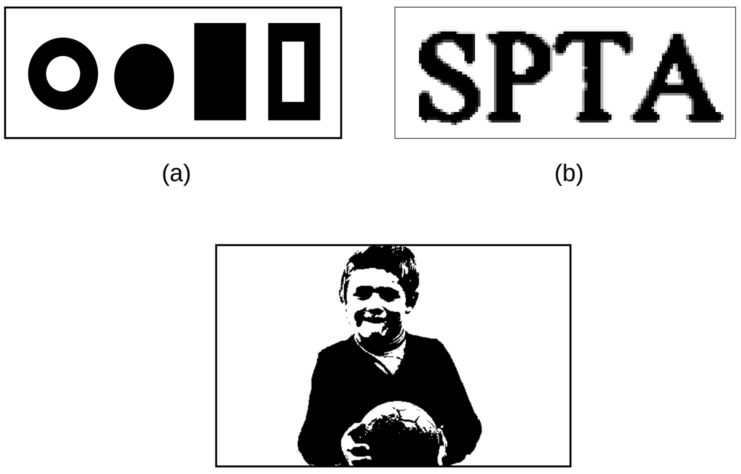


Figure 4-11. Reference images used in the experimentation. (a) *Figures* (b)*Letters* (c) *Boy&Ball*

The machines used had the featured outlined in Table 4-8. Despite the differences exhibited, the machines used can be considered homogeneous. The processor speed (*Kflops*, *MIPS*) had fairly similar values, and the applications did not require high amounts of memory, therefore, differences at this point did not affect the execution time. Usually, load average highly affects the performance, but when using *Condor*, if the *Load Average* surpasses a threshold, the machine will not be considered *idle* and will therefore not be assigned to any computation.

Feature	**Minimum Value**	**Maximum Value**
KFlops	88793	90735
MIPS	605	612
Memory	442 Mb	859 Mb
Load Average	0.04	1

Table 4-8. Characteristics of the machines used in the experimentation.

Results of efficiency and execution time (in seconds) are shown in Table 4-9 when the thinning application was run both with and without using the self-adjusting strategy (*self-adjusting* column and *No self-adjusting* column, respectively). When no adaptive scheduling was used, the initial number of requested workers was equal to the initial number of tasks. Once a task met the convergence criterion, the corresponding worker was released. In contrast, in the self-adjusting case, workers were released only in accordance with the self-adjusting strategy, and no workers were released automatically on task completion. Tasks were assigned to workers in decreasing order of average execution time, in both self-adjusting and non self-adjusting cases. Therefore, the results mainly reflect the effectiveness of the strategy to dynamically adjust the number of resources.

In addition to the results obtained for both strategies using an initial number of tasks of 8, 16 and 32, we also include the execution time of a sequential thinning application (column *InitialTasks* = 1) for comparative purposes. In the *NworkersAvg* rows, the average number of workers used are shown.

			Non Self-Adjusting			Self-Adjusting		
InitialTasks		**1**	**8**	**16**	**32**	**8**	**16**	**32**
Nworkers Avg.	Figures	1	5,5	9,12	12,85	2,45	4,17	7,37
	Letters	1	5,3	10,36	21,11	3,89	6,55	9,41
	Boy&Ball	1	5,57	7,02	11,34	2,85	4,01	8,92
Efficiency	Figures	1	0,41	0,41	0,48	0,88	0,89	0,86
	Letters	1	0,64	0,59	0,399	0,8	0,82	0,83
	Boy&Ball	1	0,59	0,64	0,7	0,88	0,86	0,87
Exec. Time (in seconds)	Figures	12746	4141	2634	1533	4473	2648	1703
	Letters	12803	3179	1562	1204	3230	1833	1399
	Boy&Ball	10080	2948	1678	1001	3094	1732	1002
ExecTime/ Efficiency Ratio	Figures	12746	10100	6424,39	3193,75	5082,95	2975,28	1980,23
	Letters	12803	4967,18	2647,45	3010	4037,5	2234,36	1685,54
	Boy&Ball	10080	4996,61	2621,87	1430	3515,9	2013,95	1151,72

Table 4-9. Results of the master-worker thinning application

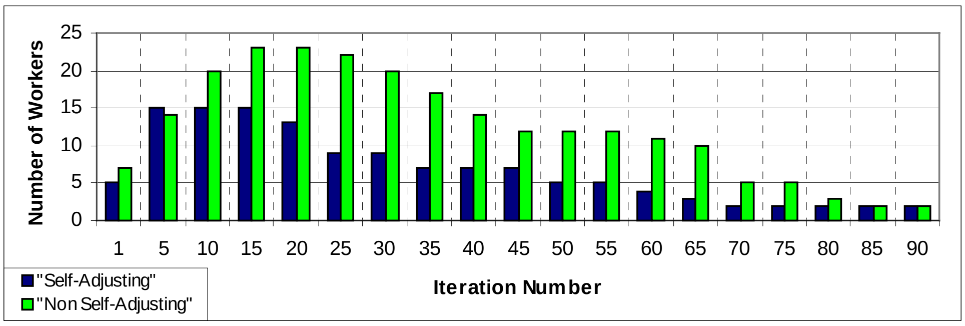
Although 8, 16 and 32 workers were initially claimed by both strategies, a smaller number of workers were effectively allocated throughout the computation. The non self-Adjusting strategy simply relinquished workers as tasks were completed. However, the self-adjusting strategy further reduced the number of allocated workers, as can be seen in the *Nworkers Avg* row of Table 4-9, which contains the average number of workers used from the beginning to the end of the computation. In general, the strategy saved between 20% to 55% of workers, compared to the Non Self-Adjusting case.

As can be seen in Table 4-9, self-adjusting obtains efficiency values above 0.8 in all cases, while no self-adjusting obtains efficiency values that are significantly smaller (between 0.4 and 0.65 in most cases). Execution time results indicate that the self-adapting strategy leads to a penalty that in most cases is less than 15% compared to the non self-adjusting case. Only for the *Letters* example with 16 and 32 tasks was the difference in execution time 17% and 19%, respectively. In general, the execution time of the application does not decrease linearly as the image is decomposed into more parts, because maximum parallelism is only achievable at the initial iterations of the algorithm. Later, as different parts of the image converge, parallelism decays and consists only of the tasks that compute the most complex parts of the images.

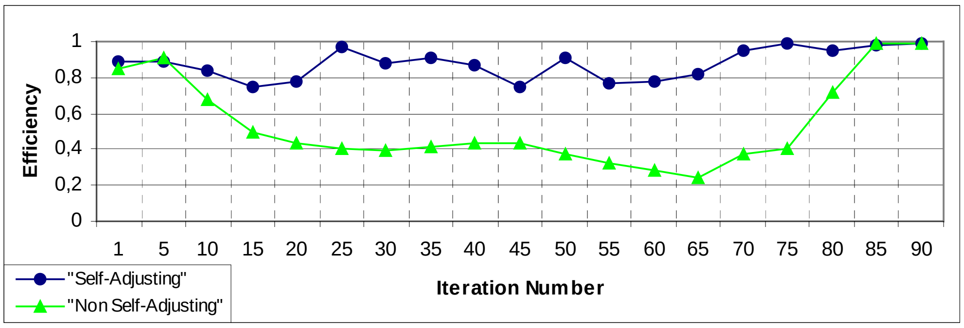
As a global index of performance, the last three rows of Table 4-9 show the index between execution time and efficiency, corresponding to both strategies (EER). The lower the index, the better the use of resources achieved by a given strategy. This means that the self-adjusting scheduling strategy achieves a better trade-off between efficiency and execution time.

Figure 4-12 shows a detailed example of one execution of the thinning application applied to the *Figures* image, initially divided into 32 parts. This example is a representative illustration of the general behavior and performance achieved by both the Self-Adjusting and the Non Self-Adjusting algorithms. The information shown is related to number of workers, efficiency and execution time after iterations 1, 5, 10, 15, and so on. Execution times are shown in a logarithmic scale.

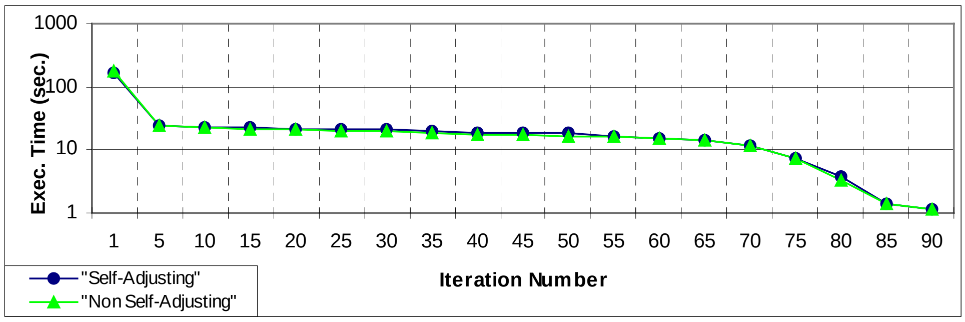
As can be seen, the allocation of resources is not serviced immediately after request. This implies, for instance, that the Non Self-Adjusting algorithm achieves a maximum number of 23 workers in iteration 15. At this time, some of the tasks have already finished (those corresponding to image borders) and, therefore, the application does not need the whole set of 32 workers requested at the beginning. In general, the Self-Adjusting algorithm is able to tune the number of workers from the initial iterations, fixing the maximum number of workers to 15 after iteration 10. Significant differences in the number of workers (and, consequently, in efficiency) are mainly observed at the central iterations of the computation (from iteration 15 to 75). In these stages, the execution time for each iteration is slightly better for the Non Self-Adjusting algorithm, at the expense of sometimes using twice the number of workers than those used by the Self-Adjusting strategy. Later, the application is close to the end and the number of workers is very small in both cases, so efficiency and execution time are very similar for both strategies.



(a)



(b)



(c)

Figure 4-12. (a) Number of workers, (b) efficiency and (c) execution time obtained with the *Figures* image divided into 32 parts.

4.3 Self-Adjusting Algorithm for Heterogeneous Environments

This section first describes the problems that heterogeneous environments introduce and how the self-adjusting strategy is modified accordingly. Subsequently, it shows the experimentation performed with the master-worker image thinning application on heterogeneous environments.

4.3.1 Modifications to the Self-Adjusting Strategy

Results obtained in a homogeneous environment are valid as long as tasks can be sorted, according to their relative importance, in terms of expected execution time. Basically, the scheduling mechanism is based on assigning larger tasks at the beginning and shorter tasks at the end. However, in a heterogeneous environment, it is not simple to derive actual task importance form measured wall clock execution time. Wall clock time will reflect both task importance (in terms of algorithm complexity) and resource performance. Therefore, a scheduling strategy needs some normalization factor to be applied to the measured wall clock times when it tries to compare task execution time averages.

Consider the example of Figure 4-13. Figure 4-13a shows the execution time of tasks when they are executed on similar machines. Figure 4-13b shows when these tasks are executed on a heterogeneous set of machines. In a heterogeneous environment, if only the wall clock times obtained in the previous iteration are considered, in the next iteration, task 4 will be considered the largest task and will be the first task executed (as in Figure 4-13c). But that is erroneous, because there are other larger tasks that have been executed on faster machines, and therefore their wall execution times are smaller.

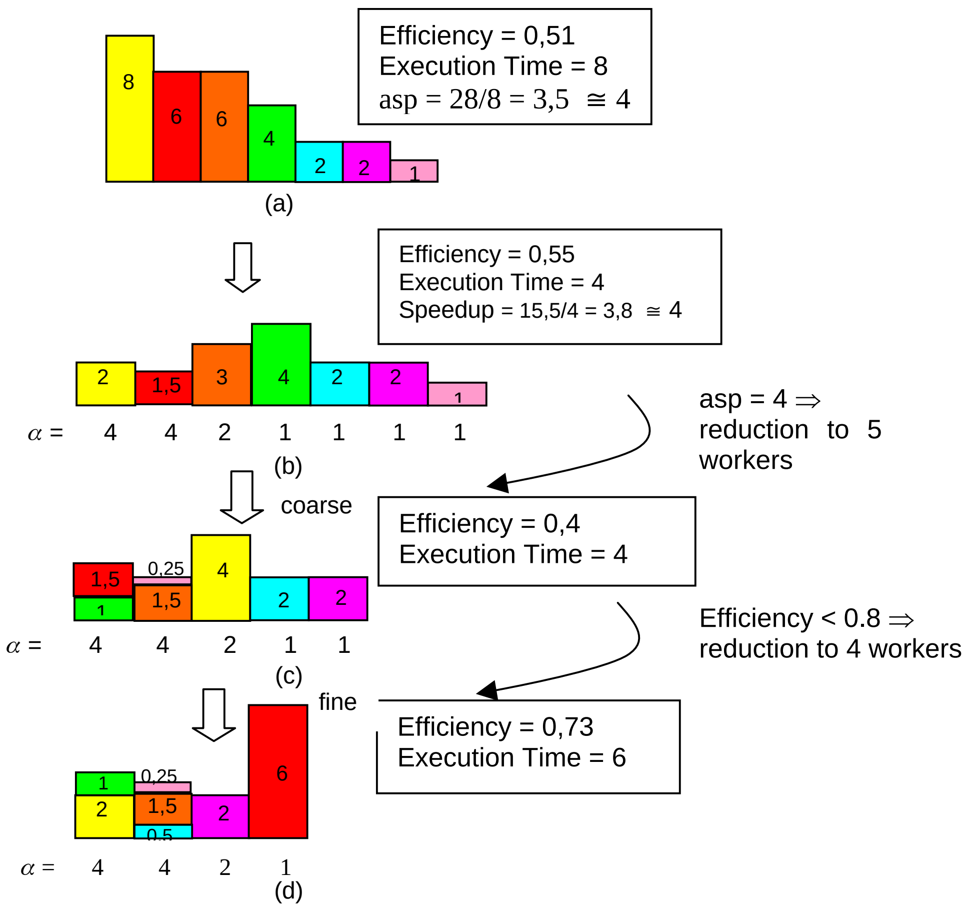


Figure 4-13. Worker allocation on heterogeneous machines. (a) Homogeneous environment, (b) As many workers as tasks, (c) Coarse adjust, (d) Fine adjust.

To prevent this algorithm from operating incorrectly, we introduced the normalization factor, called α, that directly depends on the architectural characteristics of each resource. The normalization factor of a given resource is computed by simply running a short benchmark on it when it joins the computation. The benchmark is also run in the master machine. A value α=1 is assigned to the master machine. Each worker machine *j* has the following α_j:

$$\alpha_j = BM_{master}/BM_{worker}j$$

BM_{master} and BM_{worker}*j* being the times obtained from the execution of the benchmark task in the master and worker *j*, respectively.

The factor α is used to normalize the execution times of the tasks. The absolute execution time of task *i* is defined as:

$$AET_i = ET_i * \alpha(M(i))$$

ET_i being the wall clock execution time of task *i*, and *M(i)* being a mapping function that returns the machine to where task *i* was executed. α(M(i)) corresponds to the normalization factor of the machine where task *i* was executed.

Although the way in which the α normalization factor is computed might cause inconsistent results when running the same benchmark on the same machine, producing execution times with significant differences between them, in practice it has worked quite well. In order to obtain a more precise α factor we could use the dynamic information provided by *Condor* to compute it. *Condor* periodically runs the well-known benchmarks *Linpack* and *Dhrystone*, to determine the Kflops (floating point performance) and Mips (integer performance), respectively. *Lindpack* [Don01] is a collection of subroutines that analyze and solve linear equations and linear least-squares problems. *Dhrystone* [Wei84] is a short synthetic benchmark program intended to be representative for system (integer) programming, based on published statistics on the use of programming language features. Despite the simplicity of our benchmark for computing the α factor, it was a valid approximation in practice, as our experiments confirm that machines with high α corresponded to machines for which high values for MIPS and Kflops were returned by *Condor*. The machine Load Average is also measured by periodically executing operating system calls that return this value (for example *uptime* and *top* on *Unix*).

At the end of each iteration, the wall clock execution times for the tasks are normalized by using the α factor. After this, tasks are ordered from largest to smallest by considering the average of the absolute execution times obtained. Tasks will be assigned to workers in the next iteration according to this order. Application efficiency is measured by using the wall clock times.

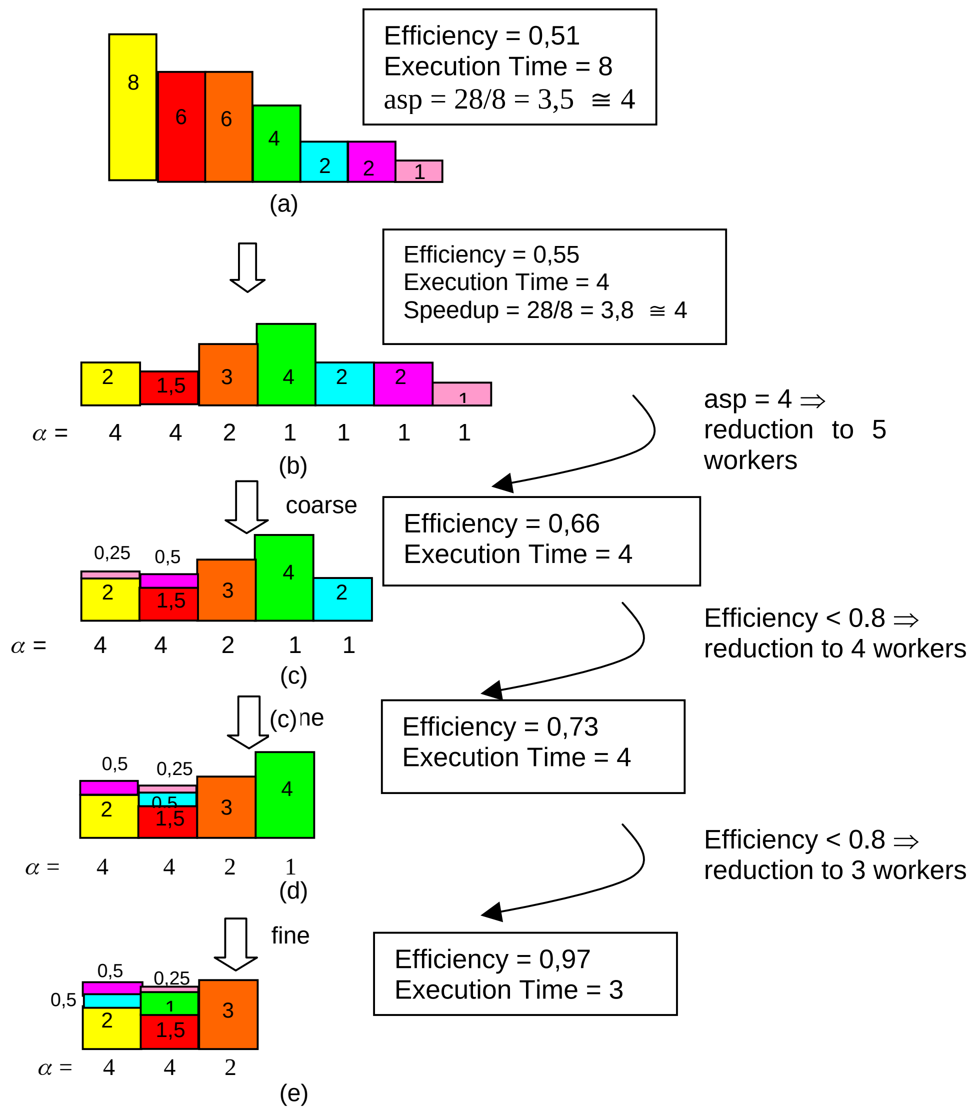


Figure 4-14. Worker allocation on heterogeneous machines. (a) Homogeneous environment, (b) As many workers as tasks, (c) coarse adjust, (d,e) fine adjust.

Figure 4-14 shows the corrected example when using normalized task execution times to determine the next task to be executed.

Once in a heterogeneous environment, the two components of the strategy are handled in the following way:

- ***Tasks Scheduling:*** At the end of each iteration, the wall clock execution times of the tasks are normalized by using the α factor. After that, tasks are ordered from largest to smallest by considering the average of the absolute execution times obtained. Tasks will be assigned to workers in the next iteration according to this order. Machines are sorted by the α factor, which means that large tasks will be scheduled onto better machines and short tasks onto slower machines.
- ***Worker Allocation:*** Application efficiency is measured by using wall clock times. When releasing a machine, the slowest one participating in the computation is released. Figure 4-14c shows an example of the coarse adjust, while Figure 4-14d and 4-14e show the fine adjust. In the implementation, a more conservative policy was taken when releasing machines, because if a machine is released but later needed, it is then possible that a slower machine will be obtained instead, and the global performance of the application is therefore worsened. This means that, as with the example of Figure 4-14, the self-adjusting algorithm will stop when efficiency is above 0.7 (Figure 4-14d).

4.3.2 Experimental Evaluation with an Image Thinning Application

The same master-worker thinning application executed on a homogeneous environment was executed on a heterogeneous environment. The available machines from the Condor pool now differed in the KFlops, MIPS, Memory and Load Average. Consequently, the execution of the application was carried out in a completely heterogeneous platform with a set of processors that exhibit significant differences in performance. The minimum and maximum feature values of the machines obtained can be seen in Table 4-10.

Feature	Minimum Value	Maximum Value
KFlops	40156	212561
MIPS	220	1049
Memory	220 Mb	1025 Mb
Load Average	0	1

Table 4-10. Characteristics of the machines used in the heterogeneous experimentation.

We will now validate the need to introduce the α factor into the algorithm initially designed for homogeneous environments.

4.3.2.1 Evaluation of the Self-Adjusting Algorithm with and without Applying the Normalization Factor

In the heterogeneous environment, we executed both versions of the self-adjusting scheduling algorithm: that adapted to the heterogeneous environment using the α normalization factor, and that which was originally implemented for homogeneous environments and did not use the normalization factor. When releasing machines, both versions release the slowest available, because in both cases the workers list is sorted according to the α factor. Both versions differ in the way in which task execution times are used. The homogeneous version uses task wall clock times directly, while the heterogeneous version uses wall clock times corrected by the α factor as previously explained.

In this case, the thinning application was run with the *Figures* and *Letters* images (Figure 4-11a and 4-11b) as examples of applications with tasks of different and similar sizes respectively. Images were initially divided into 32 parts, which corresponded to the first set of tasks created at the beginning of the execution. *Figures* was used as an example of an uneven distribution of work amongst tasks, while *Letters* is an example in which most of the tasks are assigned a similar amount of work (except tasks corresponding to the border of the images, which are assigned less work).

Results for the executions are shown in Tables 4-11 through 4-14 when the thinning application was run both using the homogeneous and heterogeneous self-adjusting scheduling strategy on a heterogeneous environment.

In addition to efficiency and wall clock time, we also measure:

- *Pool Performance*: An index that characterizes the quality of the machines used in a particular execution of the application [GK+00]. This is defined as

$$PP = \frac{\sum_{i=1}^{NMach} up_i * \alpha_{ii}}{\sum_{i=1}^{NMach} up_i}$$

up_i being the time that machine *i* was alive, and α_i being the correction factor for machine *i*. Higher values of *PP* correspond to a better pool of machines.

- *Average Number of Workers*: This contains the average number of workers used per cycle.
- *Average Number of Suspensions*: This corresponds to the average number of suspensions per cycle.
- *Number of cycles before having a stable number of machines*: This corresponds to the number of cycles it takes to obtain the maximum number of machines required.

Figures	**Homogeneous Self-Adjusting on Heterogeneous Env.**			
	Execution 1	**Execution 2**	**Execution 3**	**Execution 4**
Efficiency	0,78	0,82	0,82	0,79
Wall Clock Time	3416,2	3229,5	3678,2	3898,2
Pool Performance	1,51	1,15	0,86	0,87
Average N Workers	7	6,19	6,57	5,91
Average Suspensions	0,26	0,29	0,14	0,2
N cycles before stable	3	1	1	3

Table 4-11. Performance Metrics for *Figures* with the homogeneous version of self-adjusting (without normalization factor).

Figures	Heterogeneous Self-Adjusting on Heterogeneous Env.			
	Execution 1	Execution 2	Execution 3	Execution 4
Efficiency	0,80	0,82	0,80	0,82
Wall Clock Time	2492,5	2630,2	2624,6	2839,2
Pool Performance	1,53	2,18	1,13	1,72
Average N workers	8,22	7,64	7,84	7,13
Average Suspensions	0,15	0,24	0,4	0,55
N cycles before stable	1	3	1	2

Table 4-12. Performance metrics for *Figures* with the heterogeneous version of self-adjusting (with normalization factor).

Letters	Homogeneous Self-Adjusting on Heterogeneous Env.			
	Execution 1	Execution 2	Execution 3	Execution 4
Efficiency	0,77	0,79	0,79	0,79
Wall Clock Time	1420,0	1264,5	1657,9	1360,7
Pool Performance	0,9	1,67	1,5	1,22
Average N workers	8,46	8,76	8,13	9,28
Average Suspensions	0,03	0,08	0,7	0
N cycles before stable	4	1	3	1

Table 4-13. Performance metrics for *Letters* with the homogeneous version of self-adjusting (without normalization factor).

Letters	Heterogeneous Self-Adjusting on Heterogeneous Env.			
	Execution 1	Execution 2	Execution 3	Execution 4
Efficiency	0,75	0,77	0,76	0,79
Wall Clock Time	1105,1	1096,4	1138,6	1221,3
Pool Performance	1,59	1,67	1,55	1,57
Average N workers	10,11	10,15	10,5	9,06
Average Suspensions	0	0,48	1,1	0
N cycles before stable	1	1	1	3

Table 4-14. Performance metrics for *Letters* with the heterogeneous version of self-adjusting (with normalization factor).

The heterogeneity of the resources used in each run of the application makes application performance with the homogenous and the heterogeneous version of the self-adjusting strategy difficult to assess. Application execution time (measured as a pure wall clock time) depends on several factors in addition to whether the normalization factor is used or not. However, from Tables 4-11, 4-12, 4-13 and 4-14, it is observed that both versions of the self-adjusting strategy always obtain efficiency values above 0.75 (as both algorithms work by reducing the number of machines until reasonable efficiency is achieved). The average number of workers used is consistent in the executions. More workers were used when all the tasks were similar (*Letters*), while with the *Figures* example, there were many short tasks and, in consequence, the number of workers was less in both versions.

The experiments showed that overall execution time mainly depends on the latest worker that completes its work, once the application runs in a steady state with a stable number of workers. This time basically depends on both the relative importance of tasks assigned to the worker and the relative performance of the worker. The pure heterogeneous version identifies the relative importance of each task, thanks to the α normalization factor. It therefore achieves a stable situation in which the same worker (to which the same task(s) is assigned in all iterations) is the last one to finish at each iteration, and the number of workers is reduced in a coherent way, until this situation is reached.

On the other hand, the homogeneous version does not correctly identify the relative importance of each task. As a result, tasks are not always assigned to the same worker to whom they were previously assigned. Additionally, the last worker to finish is different from one iteration to another. Therefore, the homogeneous version of the adjusting algorithm reduces the number of workers using erroneous execution time reference; this leads to using a number of workers that is slightly less than the number used in the heterogeneous version of the algorithm.

By using fewer workers than the heterogeneous version, the homogenous version always achieved a worse overall execution time. The worst case of the heterogeneous strategy was always better than the best case of the homogeneous strategy. The consequences of incorrectly identifying the relative importance of each task are more noticeable in those cases in which tasks are not similar in terms of execution time. That is why, in the executions performed with the *Figures* image, where task execution times are within a wider range, there are more differences in the final execution time (wall clock times): because the errors made by the homogeneous version without normalization factor were more significant. When tasks are similar (as in the case of the *Letters* image), the errors produced by not taking the normalization factor into account are hidden, because there are many equivalent large tasks.

Wall clock time was not only reduced by using a larger number of workers. There are other factors that contribute to this reduction: pool performance, suspensions, the number of cycles elapsed before allocating the desired number of machines, and the order of worker allocation, although these have only a minor influence.

In general, for a given set of workers, wall clock time tends to be reduced if these machines have a better pool performance. It was also observed that the better the pool performance and the more balanced the relative performance of individual workers, the better the overall execution time. For the same number of workers, a higher pool performance decreases execution time, while a lower pool performance increases it. With a similar pool performance, a higher number of machines reduces wall clock time, while a lower number of machines increases it. This rule is not always fulfilled, as it can be seen in executions 1 and 2, in Table 4-11. This is explained by the effect of other factors such as the average number of suspensions and the number of cycles elapsed before allocating the desired number of machines.

A higher number of average suspensions increases execution time, even if both the number of workers and pool performance are high, because it implies that some of the iterations take a longer time to finish. The other factors that influence execution time, in addition to pool performance, include the number of cycles needed to allocate all the workers requested at the beginning of the execution and the order in which workers were allocated (best workers might be allocated at the beginning of the execution or in a later iteration). Unfortunately, all these factors depend on the execution environment and are not under the control of the self-adjusting strategy. The next chapter will discuss some easy ways in which, in one way or another, to alleviate the effect of losing machines.

4.3.2.2 Comparison of Self-Adjusting and Non Self-Adjusting Strategies on Heterogeneous Environments

Finally, we will compare the results obtained when executing the image thinning application, with and without using the heterogeneous self-adjusting strategy. The non self-adjusting strategy works in the same way as explained for the non self-adjusting strategy in a homogeneous environment (section 4.2.4); however, normalized execution times were considered, and also machines were sorted by the α factor. These experiments were performed with the aim of evaluating the ability to improve the efficiency produced by the self-adjusting strategy. Tables 4-15 through 4-18 show the results of the thinning application being executed with the *Figures* and *Letters* images.

Figures	**Non Self-Adjusting**			
	Execution 1	**Execution 2**	**Execution 3**	**Execution 4**
Efficiency	0,47	0,51	0,48	0,54
Wall Clock Time	2947,5	2744,3	2746,01	2920,6
Pool Performance	4,07	4,44	4,36	3,73
Average N Workers	13,72	14,03	14,33	13,82
Average Suspensions	0	0,4	0	0,23
N cycles before stable	3	6	3	3

Table 4-15. Performance Metrics for *Figures* with Non Self-Adjusting Scheduling on a Heterogeneous Environment.

Figures	Self-Adjusting			
	Execution 1	Execution 2	Execution 3	Execution 4
Efficiency	0,83	0,82	0,83	0,73
Wall Clock Time	2834,09	2777,4	2999,5	2996,3
Pool Performance	4,93	4,28	4,22	4,82
Average N workers	7,23	8,05	7,57	9,7
Average Suspensions	0	0	0,15	1,02
N cycles before stable	1	1	1	1

Table 4-16. Performance Metrics for *Figures* with Self-Adjusting Scheduling on a Heterogeneous Environment.

Letters	Non Self-Adjusting			
	Execution 1	Execution 2	Execution 3	Execution 4
Efficiency	0,52	0,46	0,48	0,54
Wall Clock Time	1125,3	1103,9	1037,2	1088,2
Pool Performance	4,58	3,70	5,07	3,72
Average N workers	19,86	18,4	18,4	18,35
Average Suspensions	0,45	0	0,71	0,39
N cycles before stable	10	2	5	7

Table 4-17. Performance Metrics for *Letters* with Non Self-Adjusting Scheduling on a Heterogeneous Environment.

Letters	Self-Adjusting			
	Execution 1	Execution 2	Execution 3	Execution 4
Efficiency	0,74	0,72	0,7	0,79
Wall Clock Time	1214,6	1278,8	1265,2	1200,2
Pool Performance	4,34	3,9	3,92	4,87
Average N workers	10,39	10,78	11,11	10,65
Average Suspensions	0,51	0	0	0,56
N cycles before stable	1	1	1	1

Table 4-18. Performance Metrics for *Letters* with Self-Adjusting Scheduling on a Heterogeneous Environment.

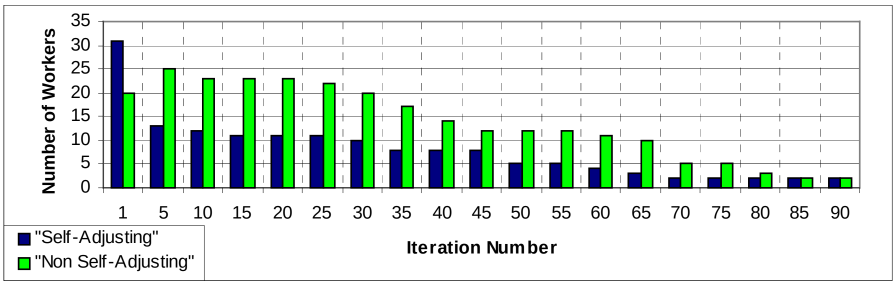
In Tables 4-15 through 4-18, it is observed that pool performance values are significantly higher than those obtained in the experiments in section 4.3.2.1. The reason for this difference is that the results of Tables 4-15 through 4-18 were obtained by using a poorer master processor, in terms of its relative performance. Workers machines therefore had a higher normalization factor, which also implies a higher pool performance.

As the main conclusion from Tables 4-15, 4-16, 4-17 and 4-18, we observe that the self-adjusting strategy always obtains efficiency values above 0.7 in all cases, while no self-adjusting obtains efficiency values that are significantly smaller (between 0.45 and 0.55 in most cases). With the *Figures* example, slightly better results in efficiency were achieved by the strategy, as it was able to schedule a higher number of small tasks so that workers were kept busier. In all examples, the self-adjusting strategy was able to save more than 40% of the workers in most cases. Moreover, execution-time results indicate that the self-adjusting strategy has a moderate penalty that, in most cases, is less than 15%, compared to the non self-adjusting case.

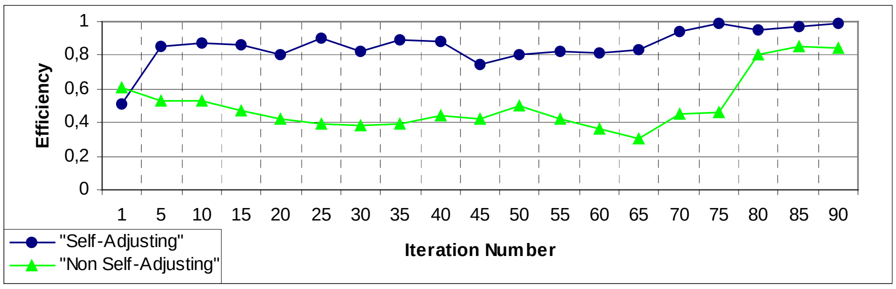
It is also worth pointing out that the effect of the average suspensions over execution time is more significant in the self-adjusting case than in the non self-adjusting case, because, in the latter, the suspension can be produced on a machine that was idle, waiting for another to finish work. In such a case, efficiency will be improved, since, if a machine is suspended, it is not able to execute useful work.

In the non self-adjusting case, there are some high values in the number of cycles elapsed before allocating the maximum number of desired machines. In many cases, this means that the application was running with 1 or 2 machines less than maximum; therefore, this factor is not very relevant in the final execution time incurred by the application. The lack of these machines mainly affected certain medium or short tasks that were scheduled to other available workers, once they had completed their previously assigned task. This scheduling was mostly carried out without incurring any penalty on the overall execution time.

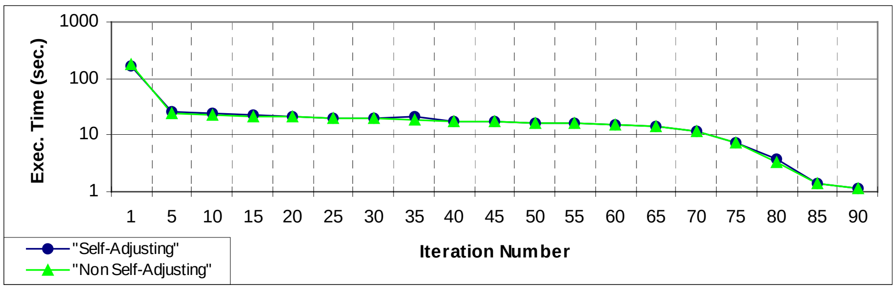
Figure 4-15 shows a detailed example of one execution of the thinning application applied to the *Figures* image, initially divided into 32 parts. This example is a representative illustration of the general behavior and performance achieved by both the self-adjusting and non self-adjusting algorithms. We show the information related to number of workers, efficiency and execution time after iterations 1, 5, 10, 15, and so on. Execution times are shown in a logarithmic scale.



(a)



(b)



(c)

Figure 4-15. (a) Number of workers, (b) efficiency and (c) execution time obtained with the *Figures* image divided into 32 parts.

Figure 4-15 is nearly equivalent to Figure 4-12 (see section 4.2.4), which was obtained when the same comparison between a self-adjusting and non self-adjusting strategy was carried out on a homogeneous environment. As a result, all the comments made for Figure 4-12 are also applicable to Figure 4-15; namely, that resource allocation is not serviced immediately after request; that the self-adjusting algorithm is able to tune the number of workers from the initial iterations, even in the presence of heterogeneous workers (in the example, using 13 machines in iteration 5); and that the most significant differences in the number of workers (and, consequently, in efficiency) are mainly observed at the central iterations of the computation (in this case, from iteration 5 to 75). In these stages, the execution time of each iteration is slightly better for the non self-adjusting algorithm, at the expense of sometimes using twice the number of workers than those used by the self-adjusting strategy.

Chapter 5

Scheduling in the Presence of Machine Loss

Abstract

This chapter first evaluates the impact of machine reclaim in an opportunistic environment. Secondly, strategies to alleviate this effect are proposed and evaluated by simulation. The scalability of the proposed strategies is then discussed. Finally, the results of the implementation of these strategies are commented on.

5.1 Introduction

In the previous chapters, it was assumed that all the machines participating in the computation were available throughout the entire application execution time, although temporal machine suspension and loss were experienced in many of our experiments in a real environment.

In this chapter, the same model of master-worker applications as in the previous chapters is considered. It is also assumed that such applications are executed on a non-dedicated cluster, and that they will use the services of a cluster middleware, which will offer several services for discovering idle CPUs and allocating them to the application. Allocated CPUs will be used to complete several batches of tasks. In general, we assume that allocated resources are not relinquished by the application until the last batch of tasks is completed. By keeping the resources allocated in this way, the application will alleviate the overhead incurred by the cluster middleware in the resource discovery and allocation phases.

We will investigate the scheduling problem that arises in parallel applications executing on a network of machines by using a mode of cycle-stealing. In this mode of execution, a parallel application executes its tasks in several machines whenever they are idle. When the user reclaims the machine, tasks must relinquish control immediately, which means that it must be stopped and vacated. In this case, the parallel application has the risk of losing work currently in progress on reclaimed machines and, therefore, the total execution time of the parallel application will be affected by the need to reschedule the pre-empted task.

We intend to provide insight into how machine-owner interference may degrade the performance of a parallel application running on a non-dedicated environment. Moreover, we will evaluate the effectiveness of simple strategies for alleviating the impact of machine loss. One strategy is based on the use of extra machines, which are added to the common pool of machines used by the application. The others are based on the use of extra machines that are used for executing certain replicas of large tasks. The following questions now arise:

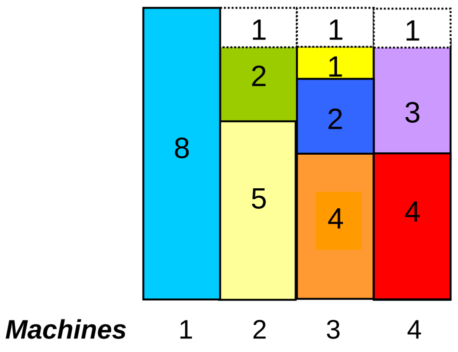
- How many extra machines should be allocated to an application, running on an opportunistic environment, in order to achieve a performance equivalent to that achieved in a non-opportunistic environment?
- Should the allocated extra machines be used for running more pending tasks or for task replication?

First the impact on the performance of an application is evaluated when it runs on two different scenarios: a set of N dedicated machines, and a set of N non-dedicated machines (in which pre-emption may occur). This study shows that losing machines may have a considerable impact on the application execution time, and therefore, three simple strategies to alleviate this problem are proposed and evaluated. All strategies are based on the use of additional machines, but differ in the way that these extra machines are used. In the first strategy, additional machines are added to the common pool of machines used by the application. The other two are based on task replication, in which the additional machines are used to execute certain tasks that are already running on other machines.

These strategies have been implemented using Condor, PVM and MW, the same environment used in previous chapters. In this chapter, we consider the same model of master-worker applications as in the previous chapters. It is also assumed that such applications are executed on a non-dedicated cluster, and that they will use the services of a cluster middleware that will offer several services to discover idle CPUs and allocate them to the application. The allocated CPUs will be used to complete several batches of tasks. In general, we assume that the allocated resources are not relinquished by the application until the last batch of tasks is completed. By keeping the resources allocated in this way, the application will alleviate the overhead incurred by the cluster middleware in the resource discovery and allocation phases.

5.2 Background to the Problem

In a non-dedicated cluster, as in any opportunistic environment, there is no guarantee that all the machines needed by an application will be available throughout the whole execution time of the application. We would like to use an example to illustrate the effect of losing machines, by taking into consideration a single batch of tasks and assuming a pseudo-optimal scheduling policy such as LPTF (Largest Processing Time First). At this point, we are only concerned with the effect of losing machines. Figure 5-1 shows 8 tasks executed on 4 machines, these being the 4 machines available during the whole computation. Colored boxes represent tasks. The number inside each task represents its execution time. Dotted boxes containing a number represent the amount of time that machines are idle but operational, and with which there is therefore a waste of machine resources. In agreement with the equation for efficiency introduced in chapter 2, an efficiency of 0.9 is obtained (machines executed useful work 29 units of time, and 32 units of time were able to execute work); and the execution time is 8 units (the batch of tasks is completed in 8 units of time).



$$E = \frac{8+7+7+7}{\sum_{i=1}^{4} 8} = \frac{29}{32} = 0.9$$

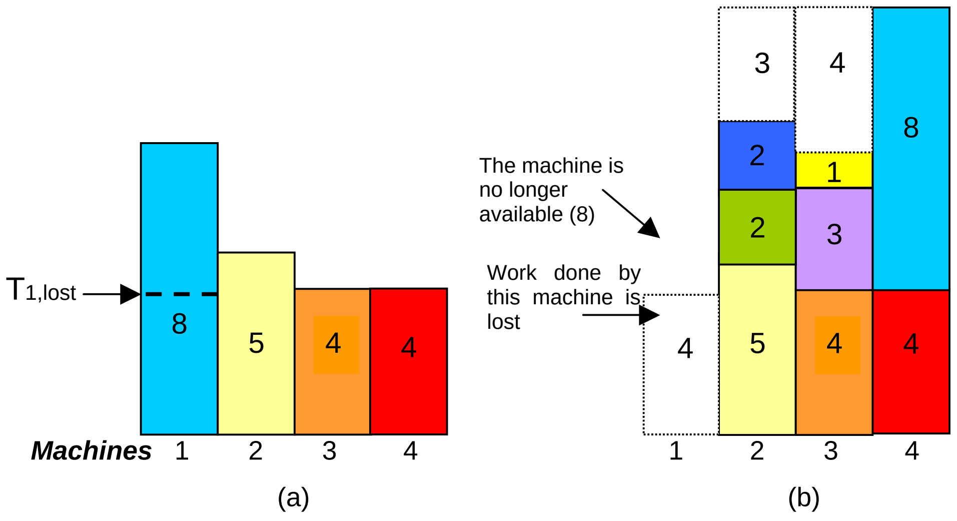
$$ET = 8$$

Figure 5-1. Example of efficiency and execution time taking into consideration a fixed number of machines.

During the execution of these tasks, in a random time $T_{i,lost}$, a random machine *i* is lost. Figure 5-2 shows the effect of losing the machine where the largest task (*t)* is running at $T_{1,lost}$=4. This task is rescheduled to machine 4 on which *t* runs from the beginning again. As task *t* is the largest of the pending tasks, it will be rescheduled as soon as a machine becomes idle. In this case, an efficiency of 0.72 is obtained (machines were 29 units of time executing useful work, and (12*4)-8=40 units of time able to execute work); and an execution time is 12 units. Clearly, both efficiency and execution time are worsened.

$$E = \frac{0+9+8+12}{4+12+12+12} = \frac{29}{40} = 0.72$$

$$ET = 12$$



T1,lost
8
5
4
4
Machines 1 2 3 4
(a)
The machine is no longer available (8)
Work done by this machine is lost
4
5
2
2
3
4
3
1
4
8
4
1 2 3 4
(b)

Figure 5-2. Example of efficiency and execution time taking into consideration the loss of one machine during execution time, at time = 4. a) Before loss occurs. b) After loss occurs.

First, we only consider situations in which one machine is lost at every cycle and the cycle is completed in a degraded way. Cycle length is not long enough to recover the machine before the cycle is completed. A new machine will again be available at the beginning of the next cycle, when the scenario in which a machine is lost will be repeated.

This formulation is a plausible situation that occurs in real situations, as has been concluded from an empirical study. Certain experiments using a Condor [LLM88] pool at the University of Wisconsin have been conducted. The master-worker application for the computation of Fibonacci series described in chapter 3 has been executed, with the following characteristics:

- The application was composed of 50 batches.
- 50 and 75 machines were used.
- Execution time incurred in the completion of one cycle (batch of tasks) was on average 12 minutes.
- Total execution time for the application was on average approximately 10 hours.
- Application workload distribution was 30% approximately.

The application was run at different times of day on different days and it was determined that, approximately, no machine losses were observed in 55% of the batches, with one machine loss being observed in 40% of the batches. Losing more than one machine in a given batch only occurred in 5% of the cases. Some factors that affect the number of machine losses produced include the priority of the user, the number of machines available, and the demand on machines belonging to the pool. Figure 5-3 shows the number of workers during each cycle for a particular execution. In general, machine losses are distributed in time, that is, they are not grouped around particular cycles.

Figure 5-3. Number of machines per cycle for a sample execution.

Situations in which more than one machine is lost per cycle are therefore not considered, because in empirical studies, such situations are less likely to occur. Only when the completion time of a batch is long enough would more than one machine be lost. Moreover, the loss of more than one machine is also accompanied in these situations with the discovery of new machines that can be allocated to the application before the whole batch is completed. This scenario therefore gives rise to a more complex analysis, and one which is beyond the scope of this current work.

5.3 Impact of Machine Reclaim

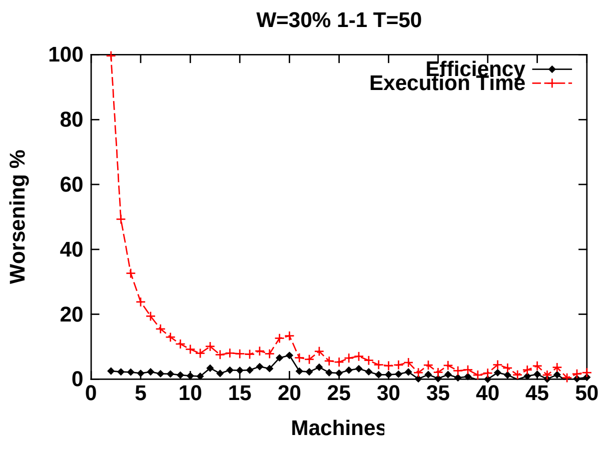
We now evaluate the impact of losing one machine per cycle with respect to efficiency and execution time. The results of this evaluation will be used later in this chapter to compare the effectiveness of different solutions that try to reduce the negative effects of machine loss. The impact of machine loss has been quantified by simulation according to the following framework, which models application and system characteristics.

A system composed of N machines is assumed. During a cycle, a random machine *m* is lost at a random time. All machines have the same probability of being lost. The time at which a machine is lost is uniformly distributed throughout the total duration of the largest task per cycle. If a task *t* was running on *m*, *t* must be executed from scratch again on another machine.

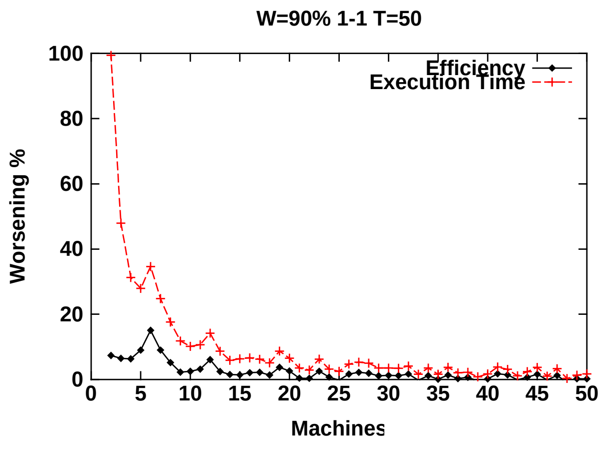
In the specification of the application and scenarios, the following parameters have been considered:

- **Number of Task *(T)*:** This is the number of tasks that composes a batch and that must be executed during the cycle. The number of tasks was 50 for all the simulations in this section. In section 5.4 a higher number of machines is considered.
- **Workload distribution *(W% i-i)*:** This is the same as explained in chapter 2: the total amount of work (*TotalW*) is divided throughout *T* tasks with the following scheme: 20% of the tasks contain *W%* of the total load, and the remaining 80% of tasks contain the *TotalW-W% load*. Workload values (*W%*) of 30%, 40%, 50%, 60%, 70%, 80% and 90% have been considered. This section only reports the results obtained in cases where all 20% of the tasks and the remaining 80% of the tasks exhibited different execution times (depicted by a label 1-1 in the figures), because those results were quite similar to the rest of cases.

To quantify the effect of losing one machine, we have compared the results obtained having *N* fixed machines and those obtained when having *N* machines and losing one. The worsening percentages have been measured systematically for efficiency and execution time, using a number of machines ranging from 2 to *Number of Tasks per batch* (50). For each simulation scenario (with a particular workload distribution and a given number of machines), 100 simulations were carried out, so different machines were lost at different time instants. Values shown in the graphs are average. The worsening percentage for both efficiency and execution time are shown in Figure 5-4 for the cases of a workload of 30% and 90%, respectively. We only show these cases because they are representative examples of the extreme situations that can be exhibited by a given workload. The X-axis contains the number of machines. The Y-axis contains the percentage values. A positive value *v%* means that when losing one machine, efficiency or execution time gets *v%* worse.



(a)



(b)

Figure 5-4. Worsening percentage when one machine is lost per cycle.
a) 30% workload and b) 90% workload.

As Figure 5-4 shows, when the application uses a small number of machines, that is, below the ideal interval ([18,22] for 30% and [5,7] for 90%), the effect of losing one machine is significant on execution time because the completion time of the cycle may be significantly affected. With 3 machines, the execution time may be 50% worse in both cases. With less than 5 machines, worsening of execution time is greater than 25%. The worsening index declines to 15% for a workload of 30% before entering the ideal interval. For a workload of 90%, the worsening percentage in execution time is close to 35% for 6 machines. In general, efficiency using a small number of machines tends to be high because all machines are busy and no idle periods are detected. So, when one machine is lost, the remaining machines are still very busy, and therefore the impact on efficiency is less significant than the effect on execution time.

With a high number of machines (larger than the ideal interval), the effect of losing one is not very significant. In this case there are many idle machines and therefore the execution time achieved by a dedicated system is close to the optimal, and efficiency is quite low (as Figure 5-4 (a) shows, efficiency for more than 23 machines falls below 0.8). This situation can be observed for more than 23 machines with a 30% workload and more than 8 machines for a 90% workload. By losing one machine, computation power will not be reduced, and efficiency is low even without the loss, because the application is running in a scenario in which not all machines are doing useful work.

Our main point of interest is focused on the *perfect number* of machines and on the surrounding interval of machines at this point (*ideal interval*), because, as stated in the previous chapters, the self-adjusting strategy dynamically adapts the number of machines reaching, a number of machines belonging to the ideal interval, in an ideal environment without machine loss. Therefore, we think that this is the desired scenario for the execution of applications. In the above examples, the *perfect number* corresponds to 20 and 6 machines for the 30% and 90% cases, respectively. At this point, the worsening effect in both execution time and efficiency is not negligible. Execution time is 13% and a 35% worse for a 30% workload and a 90% workload, respectively. This means that, particularly in the case of unbalanced workloads, the application will suffer from a significant delay due to machine losses at every cycle. In the next section, alternatives for reducing such an impact are evaluated.

5.4 Strategies for Reducing the Impact of Machine Loss

In this work, we have restricted ourselves to solutions that do not use task migration. This restriction has been adopted because the main interest concerns solutions that could easily be applied to any opportunistic environment. In current middleware systems, migration is not always supported and, when supported, it is not always available. Sometimes, the system imposes restrictions on the application that preclude the use of migration. Furthermore, migration may sometimes fail due to pre-emption deadlines or to limits on resource consumption during pre-emption. Furthermore, a model of parallel applications that uses a moderate amount of time in completing each batch of tasks has been considered. This means that in some cases, the cost of checkpointing and migrating one task could be as costly as the time needed to reschedule the task from the beginning, on a different machine.

As a consequence, our focus is on solutions that overcome the impact of losing a machine, which are conceptually simple, and whose implementation is not based on any special requirement in the underlying middleware.

As was stated in the introduction to this chapter, a simple and intuitive solution for reducing the impact of machine loss would consist of using extra machines (this solution is also referred to as using machines in advance). With this approach, we look for a solution whereby the use of a cluster of N+X non-dedicated machines may achieve the performance of a dedicated cluster with N machines. The use of additional machines for executing replicas is a solution adopted in fault-tolerant systems with deadlines for terminating tasks [LC88, OS91, GMM97]. In our case, we adapt this idea to compensate for the loss of a machine, from the point of view of the application's global execution.

However, the additional machines can be used in different ways. They can be used indistinctly from the others or they can be used to run replicas of certain tasks. In the latter case, the largest tasks are the obvious candidates for replication. There are also some hybrid solutions in which part of the extra machines may be used for replication and the other part will be used as ordinary machines. In this work, only three different cases have been considered, which nevertheless give sufficient insight into the benefits of each strategy. The strategies are denoted as:

- **NR (No Replication):** extra machines are simply added to the common pool of machines. The effect is that the application runs on a larger pool.
- **SR (Single Replication):** only the largest task is replicated in one of the extra machines.
- **CR (Complete Replication):** all the extra machines are used to run a copy of one of the largest application tasks. The largest *K* tasks are replicated if *K* extra machines are added.

In the following subsections, we evaluate the above policies when a *perfect number* of machines is used; the study is then generalized to any number of machines.

5.4.1. Impact of Machine Loss with the *Perfect Number* of Machines

For each workload distribution (*W%*), Table 5-1 shows the worsening percentage of Execution Time (uppermost number in cell) and Efficiency (number in lower part of cell), taking into consideration the *perfect number* of machines. The first column denotes the workload; the second column contains the number of machines that constitute the perfect number for each workload distribution; the third column represents the worsening percentage of having *N* machines with 1 loss with respect to having *N* fixed (this column corresponds to the situation analyzed in section 3). The remaining columns show the worsening percentage with respect to N fixed machines when 1, 2 and 3 machines are used in advance and the strategy is either NR, SR or CR. Notice that SR and CR represent the same strategy when only one machine is used in advance.

As Table 5-1 shows, there is no best strategy in every case for reducing the impact of machine loss. The ability of each strategy to reduce such impact depends on workload distribution. However, certain conclusions can be reached based on the above table. First, it is clear that the use of extra machines reduces the impact on execution time of machine loss. The worsening percentages in execution time in the third column (0 in advance column, when no extra machines are used) are always larger than the percentages obtained by any strategy using machines in advance. On the other hand, efficiency tends to be worse as more machines are used. The use of one machine in advance obtained a slight reduction in efficiency-worsening only with a workload of 30%, in comparison with the case of no extra machines (6.96% and 7.32%, respectively). In general, for any strategy, every time that one additional machine is used, efficiency experiences a negative impact that ranging from between 3% and 8% (depending on the workload distribution). This effect is caused by the addition of more machines into the pool, thereby increasing the system's capability to perform more work. However, total effective work does not increase because the extra resources are used to execute a replicated task (not contributing as effective work), or to repeat a task that has been pre-empted (in this case, overall efficiency may be worse because more machines will be idle, after completing their tasks, waiting for the completion of the whole cycle).

We also observe that complete task replication is better than single task replication in nearly all cases. Complete task replication obtains better execution times than single task replication, while the cost in efficiency is similar for both strategies. In particular, using three machines in advance, the CR strategy obtains an execution time that is never worse than 4% compared to the execution time achieved with N dedicated machines.

			Machines in Advance								
			0 in Advance	1 in Advance		2 in Advance			3 in Advance		
Workload	Perfect Number		NR	NR	SR/CR	NR	SR	CR	NR	SR	CR
30%	20	ET	13.35%	6.42%	8.49%	5.93%	5.5%	6.27%	8.42%	6.24%	3.91%
		E	7.32%	6.96%	9.1%	11.02%	10.99%	12%	16.11%	14.94%	14.21%
40%	14	ET	13.06%	8.91%	11.59%	9.57%	5.6%	5.42%	8.64%	5.56%	3.34%
		E	6.64%	9.26%	11.85%	15.49%	13.34%	13.73%	20.21%	18.42%	17.6%
50%	11	ET	16.43%	15.95%	11.83%	9.14%	4.48%	4.02%	6.24%	3.82%	2.5%
		E	7.24%	13.8%	12.61%	16.84%	14.42%	15.08%	21.76%	20.34%	20.17%
60%	9	ET	18.23%	9.96%	11.46%	10.06%	5.07%	4.26%	13.58%	6.93%	1.2%
		E	7.6%	11.05%	13.47%	19.34%	16.93%	17.31%	28.27%	25.01%	22.42%
70%	8	ET	23.33%	12.85%	11.18%	11.74%	8.02%	5.34%	12.13%	6.81%	2.27%
		E	9.59%	13.23%	13.62%	22.23%	20.43%	19.43%	29.88%	27.42%	25.36%
80%	7	ET	28.53%	18.38%	16.29%	12.31%	7.15%	7.98%	10.82%	7.71%	2.14%
		E	12.23%	17.12%	17.26%	24.04%	21.61%	22.65%	31.81%	30.34%	27.67%
90%	6	ET	34.63%	22.43%	15.86%	15.04%	10.94%	7.55%	9.63%	4.86%	3.08%
		E	15.06%	20.47%	17.94%	27.47%	25.5%	24.58%	33.56%	31.47%	31.05%

Table 5-1. Worsening percentage of execution time and efficiency taking into consideration the perfect number of machines.

The positive performance achieved by the complete replication strategy may be explained in terms of this strategy being the most effective when the system loses one of the machines executing one of the biggest tasks. In these cases, the task lost is also running in another machine and, therefore, the whole cycle is completed as if no machine loss was experienced (i.e. we achieve the same execution time as in the dedicated system). If the system loses a machine with a task that is not one of the largest, then this task should be rescheduled to another machine. However, the length of the task will be short or moderate and, on average, its impact will not be very significant. When only the largest task is replicated or there is no replicated task at all, the benefit of replication is exclusively restricted to the scenario in which the largest tasks is lost. Unfortunately, losing any other largest task has a significant and negative effect on the overall execution time, since that task needs to be executed again from the beginning.

Figure 5-5 shows a graphical example that illustrates the behavior of CR, SR and NR in accordance with the argument presented above. This example is based on the example used in section 4.2, but uses 2 extra machines. Figure 5-5a shows a scenario in which CR is applied and the largest task is lost at time 4. In this case, total execution time (ET) remains equal to execution with a dedicated system. Figure 5-5b shows the scenario in which SR is applied and the second largest task is lost. Consequently, there is needed to reschedule it, producing a final execution time of ET=9 (one unit more than in a dedicated system). Finally, Figure 5-5c shows the scenario in which NR is used and the largest task is lost again at time 4. Here, the total batch is completed at time ET = 12 (4 units more than in the dedicated scenario).

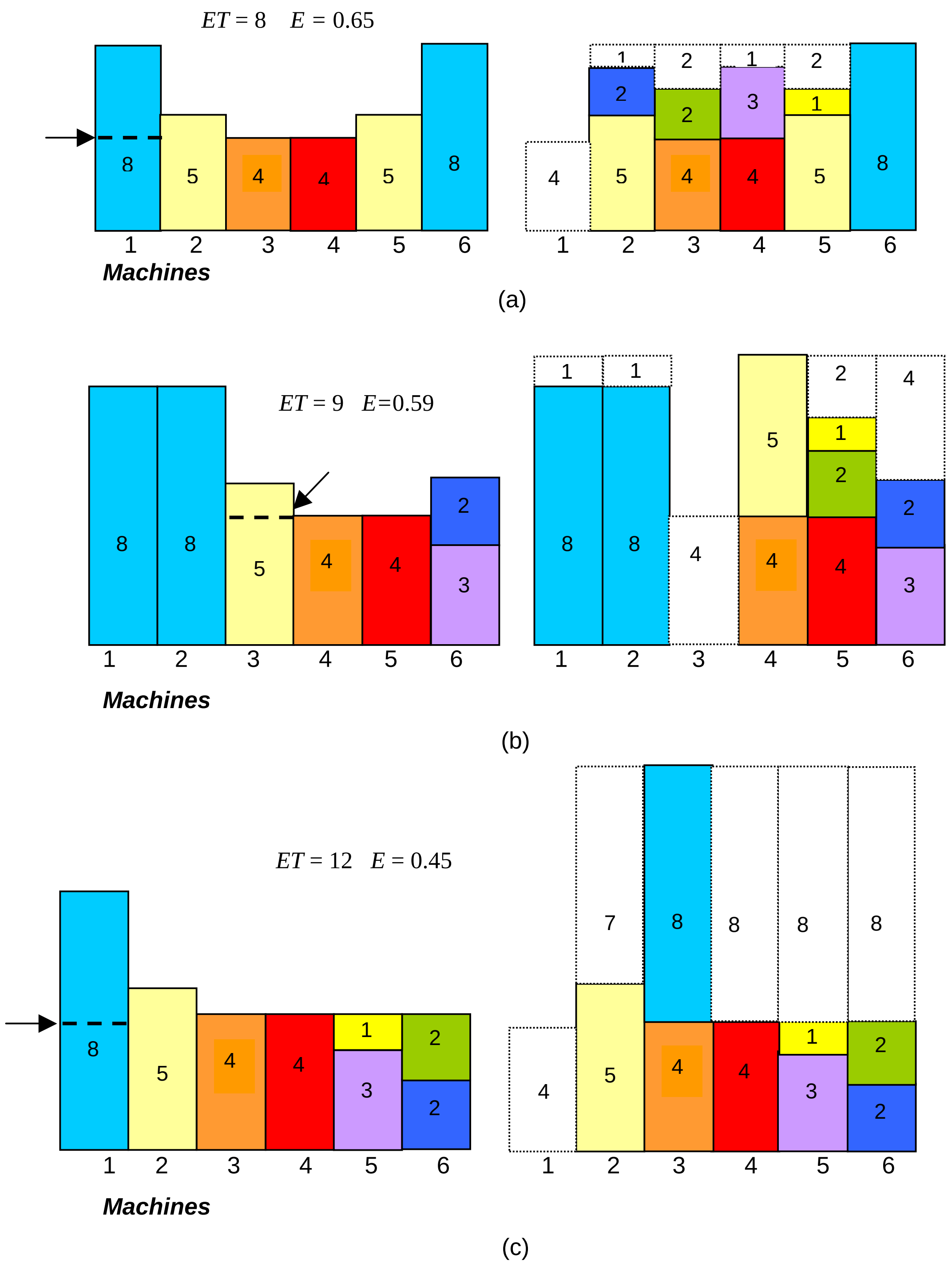


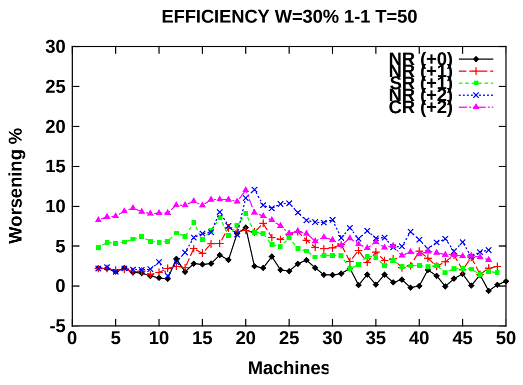
Figure 5-5. Examples of Execution Time taking into consideration 2 machines in advance and a) CR with $T_{1,lost}$=4, b) SR with $T_{3,lost}$=4, and c) NR with $T_{1,lost}$=4.

Finally, it is worth pointing out that the use of extra machines without replication appears to be the most effective strategy only for 30% and 40% workloads with one extra machine. These workloads exhibit many tasks with similar execution times, and therefore the benefits of replication are less visible when only one additional machine is used. When the batch of tasks contains significantly large tasks or the number of machines used is further increased, then replication strategies obtain better results, because they avoid the negative effects related to the loss of the largest tasks.

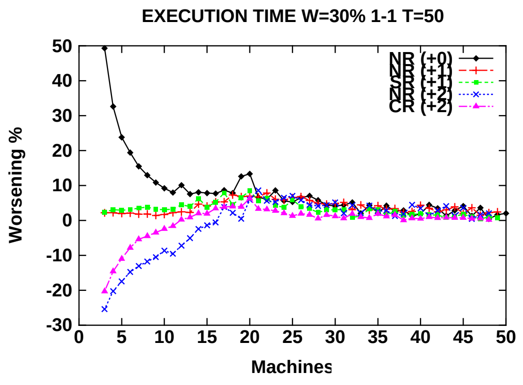
5.4.2 Impact of Machine Loss with any Number of Machines

The behavior of the above-mentioned strategies using a wider range of machines has also been evaluated. As was stated above, an application is expected to run using the *perfect number* of machines or any number of machines close to perfect (what was referred to as the *ideal interval*). However, applications may not always be able to use this desirable number of machines. For instance, the system may not find idle machines and therefore the application must run with a number of machines that is smaller than the desired. Alternatively, users may be concerned only with execution time and may not care about resource efficiency. Therefore, their application may be running with an excess of machines (above the *ideal interval*). The idea is now to obtain insight with respect to how many extra machines should be used when the application runs with a non-ideal number of machines.

Figures 5-6 and 5-7 show the worsening ratio in efficiency and execution time achieved by the NR and CR strategies, using one and two machines in advance for 30% and 90% workloads, respectively. They also show the worsening when no extra machines are used (N curve). All curves start at X = 3, i.e., at this point the execution and efficiency of a dedicated system composed of 3 machines is being compared with non-dedicated systems with 3 (N curve), 4 (N+1 curves) and 5 (N+2 curves), respectively. The case of N+3 machines has not been included in order to maintain clarity in the figures. The main conclusions can be derived from the curves shown.

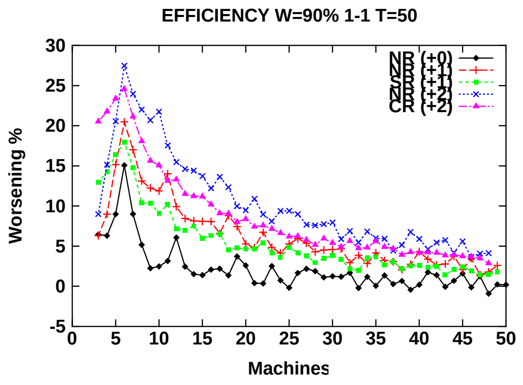


(a)

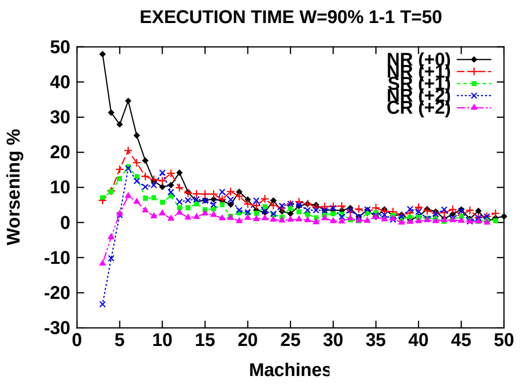


(b)

Figure 5-6. (a) Efficiency and (b) execution time worsening for a 30% workload distribution.



(a)



(b)

Figure 5-7. (a) Efficiency and (b) execution time worsening for a 90% workload distribution.

As Figures 5-6 and 5-7 show, when the number of machines used by the application is small (less than the ideal interval), using as many machines in advance as possible is the best solution for all workload distributions and strategies. This means that using two machines in advance is simply better than using one machine in advance, independently of the particular strategy adopted to manage the extra machines and the workload distribution. This effect is because the application is running with an insufficient number of processors and, therefore, any additional processor will improve execution time even in the presence of machine loss. However, for a fixed number of extra machines, NR is slightly better than CR both in terms of execution time and efficiency. Finally, it is worth mentioning that, when two extra machines are used, both NR and CR improve the execution time of the application (the worsening in execution time is negative in both cases).

When the application runs with a large number of machines, task replication strategies tend to perform slightly better than no replication strategies, although the percentage difference between the results achieved by both strategies is always less than 5% for a fixed number of machines in advance. In this case, applications execute with an excess of machines, i.e., a similar execution time might be achieved using fewer machines. Therefore, if machines are added in advance, and if they are used to replicate the largest tasks, the negative effects of losing these tasks are avoided. In this case, there is a scenario in which the loss of one of the largest tasks obtains the same execution time as in a dedicated system (as Figure 5-5a illustrates).

5.5 On the Scalability of Strategies for Reducing the Impact of Machine Loss

The following question now arises: Are the same results obtained in the previous sections valid when a larger number of machines are considered?, i.e., how do those results scale?. In order to get insight into this question the same experiments were done taking 150 machines into account.

For each workload distribution (*W%*), Table 5-2 shows the worsening percentage of Execution Time (uppermost number in cell) and Efficiency (number in lower part of cell), for 150 tasks per batch, taking into consideration the *perfect number* of machines. The first column denotes the workload; the second column contains the number of machines that constitutes the perfect number for each workload distribution; the third column represents the worsening percentage of having *N* machines with 1 loss with respect to having *N* fixed. The remaining columns show the worsening percentage with respect to N fixed machines when 1, 2 and 3 machines are used in advance, and the strategy is NR, SR or CR.

As Table 5-2 shows, the effect of losing a machine is less relevant in the case of 150 tasks than in the case of 50 tasks. This is because even though the relative difference between largest tasks and smallest tasks is the same in both cases, the absolute differences is greater in the case of 50 tasks.

The use of extra machines reducing the impact in execution time of machine losses is still valid, but not so much as in the case of having fewer machines. The worsening percentages in the NR-0 in advance column are larger than the percentages obtained by any strategy using machines in advance, except in the case of 30% workload and one extra machine.

			Machines in Advance								
			0 in Advance	1 in Advance		2 in Advance			3 in Advance		
Workload	Perfect Number		NR	NR	SR/CR	NR	SR	CR	NR	SR	CR
30%	57	ET	7.18%	8.55%	7.87%	5.8%	5.31%	5.63%	6.23%	5.26%	4.79%
		E	4.46%	6.96%	6.69%	6.53%	6.39%	6.55%	8.46%	8.09%	8.02%
40%	40	ET	9.15%	7.47%	5.95%	5.46%	4.27%	4.59%	6.08%	5.76%	4.2%
		E	5.32%	6.63%	5.78%	7.42%	6.77%	7.18%	10.01%	9.78%	9.11%
50%	31	ET	12.4%	7.54%	6.68%	7.44%	6.28%	5.62%	6.4%	5.85%	4.9%
		E	7.2%	6.66%	6.72%	9.56%	9.28%	8.05%	11.48%	11.21%	10.67%
60%	26	ET	15.77%	14.82%	11.74%	9.71%	7.7%	8.1%	11.12%	10.24%	7.14%
		E	9.15%	10.78%	10.1%	10.4%	10.28%	11.2%	15%	15.76%	13.75%
70%	22	ET	16.44%	14.88%	13.74%	14.18%	13.74%	9.47%	13.48%	12.74%	7.81%
		E	9.54%	11.27%	11.19%	14.82%	14.75%	12.15%	17.47%	16.76%	13.93%
80%	19	ET	18.76%	16.05%	16.22%	15.46%	13.25%	12.54%	12.88%	10.72%	9.42%
		E	10.72%	12.92%	13.99%	16.66%	15.5%	15.76%	19%	17.88%	17.87%
90%	17	ET	24.43%	19.35%	20.39%	17.39%	14.04%	15.48%	16.63%	15.01%	9.61%
		E	14.27%	15.69%	16.95%	18.72%	17.3%	18.67%	22.39%	21.84%	19.35%

Table 5-2. Worsening percentage for execution time and efficiency, considering 150 tasks.

Improvements achieved in execution time are smoothed in the case of 150 tasks because the ratio between extra machines and the perfect number of machines is lower than the ratio exhibited in the case of 50 tasks. Similarly, efficiency also worsens in a smooth way for the same reason. However, for a given workload and equivalent ratios between the number of extra machines and the perfect number, similar results can normally be observed in the cases of both 150 and the 50 tasks.

For instance, the negative impact of execution time is reduced in relative terms in a similar way in the case of 50 tasks with a workload of 30% with 1 machine in advance and in the case of 150 tasks with a workload of 30% with 3 machines in advance. In these cases, the ratios between the number of extra machines and the perfect number are, respectively, 1/20 and 3/57, which can be considered equivalent. And the worsening percentage is reduced from 13.35% to 8.49% (36.4%) and from 7.18% to 4.79% (33.3%), respectively.

Task replication is the strategy that achieves a better alleviation of machine loss, but its benefits are mainly noticeable when the number of extra machines is 3. This fact can again be explained by the relative weight of extra machines over the total number of machines. As more machines are used in the case of 150 machines, the number of large tasks that have to be replicated must be larger than the number used in the case of 50 tasks, in order to achieve an equivalent reduction in the worsening percentage of execution time.

Taking into account the results of Tables 5-1 and 5-2, it can be concluded that the reduction of the negative impact in execution time due to machine losses can only be achieved at the expense of worsening the overall efficiency exhibited by the application. In this case, it would depend on how much each particular user is willing to pay in terms of efficiency in order to improve application execution time. The more concerned the user is about execution time, the larger the number of machines that should be used for task replication. However, according to our simulation results, adding a number of extra machines between 15% or 20% more than the perfect number has proved that reasonable improvement in application execution time can be attained with a moderate worsening in overall efficiency.

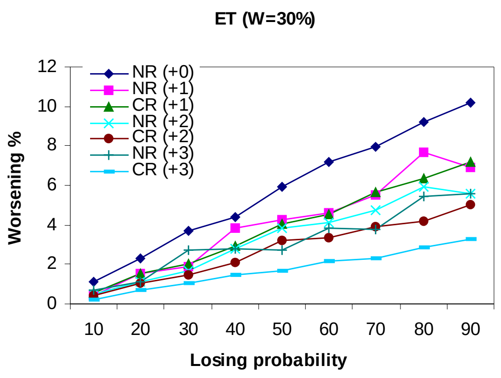
5.6 Study of Strategies Considering Different Probabilities of Losing Machines

In the previous sections, it was assumed that there was a machine lost at each cycle. In the experimental evaluation, we saw that in real opportunistic environments machine losses are less frequent. This means that the probability of losing a machine during a cycle will have a value of between 0 and 1. Obviously, the occurrence or not of a machine loss in a given cycle cannot be predicted beforehand. This fact implies that the use of extra machines (either with or without task replication) cannot simply be used as a policy that is turned on and off on an iteration basis. Extra machines and eventually task replication must be used in a sort of speculative way. The master-worker application will benefit from these extra machines only in the cycles in which a machine is lost, otherwise, the application will be penalized in its efficiency in cycles in which no machines loss occur. Intuitively, one would expect that the use of extra machines is worthwhile for “high” frequencies of machine loss, i.e., when nearly all cycles result in one machine being lost. For low occurrences of machine losses, the use of extra machines would not pay off.

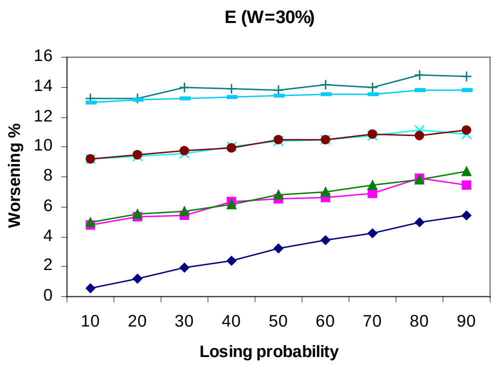
In this section, we present the results of a set of simulations that were carried out in order to determine which frequencies of machine loss it is worth using extra machines for, and for which frequencies it is not. These results will also highlight some empirical rules that may be applied in practice in order to decide when the use of extra machines should be turned on or off.

That is, in this section we evaluate the results obtained when performing the same simulations, but including a probability of losing a machine in a cycle. This probability took the following values: 10%, 20%, 30%, 40%, 50%, 60%, 70, 80% and 90%.

The main conclusions of this study can be derived from Figures 5-8 and 5-9 which show the simulation results for worsening percentages of execution time (ET) and efficiency (E) for different losing probabilities, considering 30% and 90% workloads, and always assuming the *perfect number* of machines. *(NR+0)* represents the worsening percentage incurred when losing one machine without having any extra machine, with respect to the case in which no machines are lost. The other series show the worsening percentage for the NR and CR strategies when 1, 2 and 3 extra machines are used. The complete set of values for all workloads and losing probabilities of 10%, 40%, 50%, 60%, 80% and 90% can be seen in Appendix E.

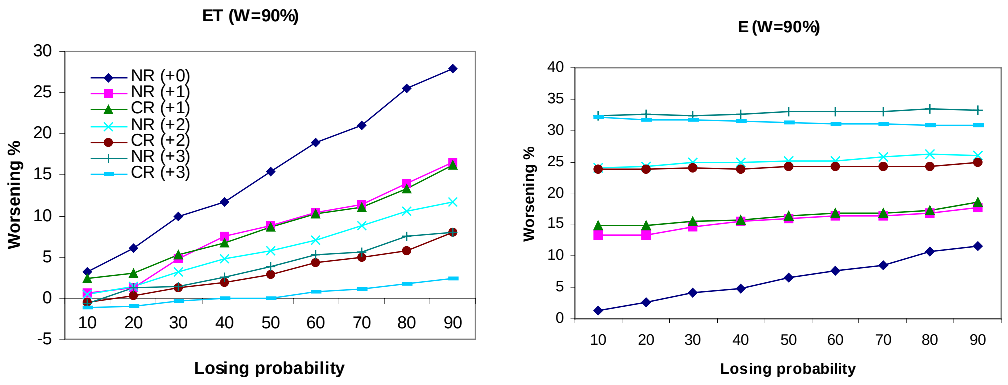


(a)



(b)

Figure 5-8. Worsening percentage for a 30% workload. (a) Execution time (ET), (b) Efficiency (E).



(a) (b)

Figure 5-9. Worsening percentage for a 90% workload. (a) Execution time (ET), (b) Efficiency (E).

As can be seen in Figures 5-8 and 5-9 and also in Tables E-1 through E-6 of Appendix E, the probability of losing a machine has a linear effect on the worsening of execution time. The higher the probability of losing a machine, the higher the worsening percentage. In all cases, the use of additional machines reduces the worsening of execution time and it is normally observed that for the same number of extra machines, complete task replication performs slightly better than no replication. These results agree with those depicted in previous sections in which the probability of losing a machine was considered to be 100%. In some cases, Complete Replication with 2 extra machines is even able to outperform No Replication with 3 extra machines. It is also worth noting that, for low or even moderate probabilities of machine loss, the negative impact on application execution time is low or moderate. For a workload of 30%, the worsening percentage is always less than 10% for all probabilities. For larger workloads, worsening percentages above 10% are obtained only for probabilities of loss of 50% or greater.

From point of view of efficiency, every extra machine added to the application increases the worsening of efficiency by a percentage that remains nearly constant, independently of the probability of losing machines and the use made of those machines (either for task replication or not). The cost of each extra machine ranges, approximately, from 3% with a 30% workload to 8% for a workload of 90%.

If we take into account the worsening percentages both of execution time and of efficiency, it seems clear that the use of extra machines (and task replication) is mainly worthwhile in those cases in which the probability of losses is moderate or high. The cost to be paid in terms of efficiency is too high when the probability of losses is less than a 50% in all workload cases. Obviously, in view of the comments made in the previous section, the use of task replication might be adopted independently of the probability of loss in those cases in which reducing the overall execution time is the most important criterion for a given user, and where overall efficiency is not relevant.

A final observation is related to the fact that task replication exhibits significant improvement, especially in scenarios with high probabilities of machine losses. From a practical point of view, this would constitute an empirical rule by which to automatically trigger the task replication mechanism. Task replication would be triggered whenever machine suspension or loss was detected at least once every two cycles. Otherwise, the application would be executed without using any extra machine.

5.7 Experimental Evaluation on a Real Platform

In this section, we will present a prototype implementation of the CR strategy, as the results obtained in previous sections have shown that this strategy proved to be the one that provides the best alleviation for machine losses in most cases. The section is completed with a brief experimentation in which the CR and the NR strategies were applied to a PovRay application [FHK98].

5.7.1 Implementation of Strategies

Although task replication is a conceptually simple strategy, some extensions had to be included in MW in order to support it. We will describe these changes below.

As was briefly introduced in chapter 2, MW handles a *workers* list with information about workers participating in the computation. It also handles lso the *ToDo* list and the *Running* list for tasks. The former contains all the tasks pending for execution for a particular iteration, and the later contains the tasks that are currently being executed.

Support for the CR (Complete Replication) strategy was achieved basically by adding a mechanism for dealing with replicated tasks in the *ToDo* list. At the beginning of each cycle, and after having sorted the *ToDo* list according to the *Random & Average* scheduling policy, the largest X tasks, which correspond to the first X tasks in the ToDo list, are replicated. They are then inserted into the *ToDo* list, keeping the list sorted. X corresponds to the number of extra machines used.

When a worker becomes idle, it receives another task to be executed if there are remaining tasks in the *ToDo* list. It could get either a normal or replicated task, indistinctly. However, when a machine is lost, different actions are taken, depending on the strategy considered. In the case of the NR strategy, the task running on the reclaimed machine is rescheduled, that is, it is placed in its respective position in the *ToDo* list. On the other hand, under the CR strategy, if the task running on the reclaimed machine is or has a replica, this task is not rescheduled; it is simply lost. But if it is not or does not have any replica, this task is rescheduled, as in the case of NR.

It is worth pointing out that this technique would work directly in a heterogeneous environment, because the largest tasks and their replicas would be executed in the fastest machines.

5.7.2 Experimentation on a Real Platform, and Results

This subsection presents the initial experimentation performed with the replication strategies.

The experimentation was carried out using a homogeneous cluster of 10 Sun4x Solaris2.5 workstations, at 29 Mips and 72 Mb RAM, interconnected by Ethernet and running Condor. The *PovRay* application for rendering graphics was parallelized following a master-worker paradigm. *PovRay* creates and/or animate images by means of a *Raytracing* algorithm, and falls into the category of embarrassingly parallel, because it can be decomposed into independent tasks. The resulting parallel application, *MW-PovRay*, distributes different segments of an image (tasks) among the workers available. When the computation of an image finishes then a cycle is completed, and the master sorts the pieces of the next image to be completed according to the results obtained in the previous cycle. It is important to remark that in a complete image sequence, the differences between one image and that following it are minimal. Each image was stored in a file, and the set of files constituted the animation.

In the experimentation, we used the *skyvase* image, provided with the *PovRay* program, and shown in Figure 5-10. All the executions had the following fixed parameters:

- *Image size*: 640 x 480 pixels.
- *Number of Tasks per cycle* (*T)*: In order to perform the computation, the image was divided into 10 tasks per cycles.
- *Cycle Length*: The duration of the cycles was approximately 60 seconds on average. The largest and smallest task duration was on average 56 and 21 seconds, respectively.
- *Workload Distribution (W)*: The workload obtained when dividing the image into 10 tasks was 33% approximately.

- *Perfect Number*: The experiments carried out with the *skywase* image corresponded to a *Perfect Number* of 7 machines or workers.
- *Number of Iterations or Cycles*: This was fixed to a value of between 30 and 50 depending on the probability of loosing a machine, with the aim of distributing machine loss uniformly among the machines.

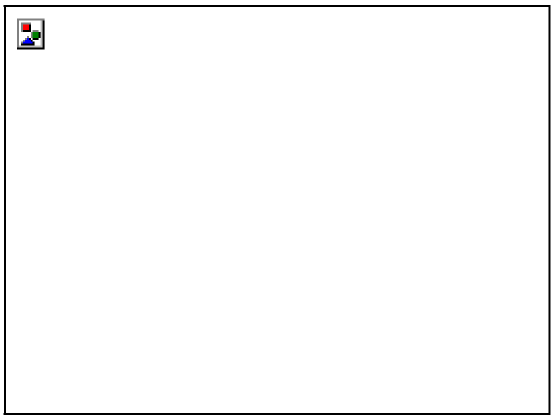


Figure 5-10. *skyvase* image

Our experiments were carried out using a mechanism integrated in the master process that allowed us to lose machines in a controlled way. At the beginning of each cycle, machine loss was computed to occur in a pseudo-random time according to a determined probability. Once the machine to be lost was determined, the master process killed the corresponding worker at the specific time. As a consequence, the task that was running in that worker was automatically reinserted in the ToDo list, waiting for another machine to execute it. All machines were available again at the beginning of the next iteration. This mechanism allowed us to repeat and compare results from different executions, although it did not behave exactly as a pure opportunistic environment.

Table 5-3 shows the worsening percentage for execution time (*ET* rows) and efficiency (*E* rows) for certain determined probabilities of loss (20%, 50% and 90%) considering the *perfect number* of machines. The first column shows the worsening percentage when having 7 machines (perfect number) and losing 1, with respect to the case of having 7 machines without losses. The other two columns show the worsening percentage with respect to 7 fixed machines when having 7 machines plus 1 extra machine, and 1 machine is lost at each cycle with *losing probability*. These columns correspond to the NR and CR strategy respectively. All measures include the time incurred in both killing a worker in order to lose it, and in restoring it at the beginning of the next iteration.

		Perfect Number	**1 Extra Machine**	
Losing Probability		**NR**	**NR**	**CR**
20%	ET	6.45%	3.41%	7.47%
	E	1.6%	10.63%	16.69%
50%	ET	16.73%	8.42%	10.24%
	E	7.64%	14.01%	16.26%
90%	ET	27.92%	15.77%	17.02%
	E	12.27%	18.23%	19.74%

Table 5-3. Execution time and efficiency worsening percentage for a 30% workload application.

When measuring efficiency, it is important to remark that with the NR strategy, for each cycle, all the work except that performed by the lost worker is productive work. However, with the CR strategy, if a task *t* is replicated, and neither the machine executing *t* nor the machine executing the replica, are lost, then only one of these is recorded as useful work.

The results shown in Table 5-3 agree with the simulations that were previously presented. We can see that with a probability of loss of 50% or higher, using an extra machine significantly reduces the worsening percentage for execution time. In these experiments, the NR strategy exhibited better results than the CR strategy as the application workload was close to 30%. This means that the execution time for tasks does no present important differences among them. Consequently, if one task is replicated but another is lost, then that replication was less useful than just having an additional machine.

The loss of machines was proportionally more significant in these experiments because, as was seen in the experimentation regarding scalability, the application was running with a small *perfect number* of machines. However, we observed that both in the simulations and experimentation, when having one extra machine, the NR strategy reduces the worsening percentage for execution time by more than a half.

SCHEDULING IN THE PRESENCE OF MACHINE LOSS

Chapter 6

Conclusions and Future Work

Abstract

This chapter presents the conclusions obtained from this thesis, in addition to work currently being undertaken and the work plan to be followed in the future in order to continue research on scheduling of master-worker applications in opportunistic environments.

6.1 Conclusions

In this work, we have proposed and developed solutions to certain challenges in executing master-worker applications within opportunistic environments. We shall now review each of the main objectives in this work and comment on how each of these has been attained.

Our work was aimed at developing efficient scheduling algorithms for a particular class of parallel applications that follow a master-worker paradigm. The execution environment assumed in our work was opportunistic clusters of machines, a particular type of non-dedicated distributed systems that are characterized by harnessing idle machine times for executing user jobs. We started our work by reviewing the most significant solutions proposed in the literature, paying attention to the system model, the programming model and the performance model adopted in each solution. According to the main families of scheduling solutions found in the literature, our work would be included within the group of application-aware dynamic scheduling strategies, including aspects with regard to system-awareness.

We identified the main features that characterize both the generalized master-worker programming model and the opportunistic set of non-dedicated heterogeneous machines. Our initial problem was divided into three main sub-problems. The first was related to the design of a scheduling strategy responsible for controlling the order in which tasks were assigned to processors. The second dealt with the allocation of a proper number of resources to the application. The last sub-problem focussed on the attenuation of the effect produced by machine pre-emption.

An initial simulation study was carried out in order to evaluate different scheduling policies by considering a fixed number of homogeneous machines. This study served to understand what happens in a simple scenario in which master-worker applications exhibiting a wide range of different characteristics were scheduled. We developed the *Random & Average* scheduling policy, which does not use any *a priori* information about the behavior of the application, and works basically by sorting tasks according to their average execution time, then assigning them in accordance with this order. We compared this with the *LPTF, LPTF on Average* and *Random* policies. All these strategies exhibited different degrees of dynamic adaptability and a priori information required of the application. We found that *Random & Average* performs similar to other policies, such as *LPTF,* which are not applicable in practice because they require precise information beforehand about the execution time of the application tasks. This study can be found in:

[HS+00a] E. Heymann, M. A. Senar, E. Luque and M. Livny, "Evaluation of an Adaptive Scheduling Strategy for Master-Worker Applications on Clusters of Workstations", Proc. of 7th International Conference in High Performance Computing (HiPC 2000), Lecture Notes in Computer Science series, Vol. 1970, pp. 310-319, 2000.

[HSL00a] E. Heymann, M. A. Senar, E. Luque, "Adaptive Scheduling for Master-Worker Applications on Clusters of Workstations", in Actas de las XI Jornadas de Paralelismo (Proceedings of the XI Spanish Workshop on Parallel Computing), pp. 205-210, Granada, Spain, September 2000.

[HSL00b] E. Heymann, M. A. Senar, E. Luque, "Gestión dinámica de aplicaciones master-worker sobre sistemas distribuídos", in Actas del VI Congreso Argentino de Ciencias de la Computación (Proceedings of the VI Argentinian Conference on Computer Science) CACIC 2000, pp. 1077-1088, October 2000.

From the simulation study mentioned above, we derived the concept of *ideal interval*, which corresponds to a number of machines for executing a master-worker application obtaining both a reasonable execution time and good efficiency. We developed a self-adjusting strategy that dynamically attempts to reach a number of machines belonging to the *ideal interval*. A preliminary version of the strategy used an empirical table obtained from the simulations. This method presented certain drawbacks in terms of algorithm complexity and stability properties, therefore we developed a new version that performed the adjustment in a completely dynamic way, without the need for any external table. The decision-making process in this dynamic self-adjusting strategy is guided only by direct performance information collected from the application during its execution. According to the execution time and the efficiency exhibited by the application in each one of its main iterations, the strategy allocates or releases resources following certain simple rules.

Initially, the algorithm worked on homogeneous environments, but later it was corrected in order to also be executed on heterogeneous environments. The use of a normalization factor was included, due to the need for sorting tasks according to their relative importance in terms of computational complexity.

As a means of evaluating the self-adjusting strategy, we implemented an image thinning master-worker application, which was executed on a real opportunistic system. We compared the execution of the thinning application both with and without using the self-adjusting strategy in homogeneous as well as heterogeneous environments. In all cases, by using our self-adjusting strategy, efficiency was close to 0.8, while execution time was not greater than 15% of the time obtained when using as many workers as tasks in the non self-adjusting case. The improvement in efficiency achieved by our strategy implied that the application was able to save 40% of resources in most of cases, with only a small degradation in the overall execution time.

The version using static tables, the fully dynamic version of the self-adjusting strategy and their corresponding experimentation carried out on homogeneous environments can be found, respectively, in:

[HS+00b] E. Heymann, M. A. Senar, E. Luque and M. Livny, “Adaptive Scheduling for Master-Worker Applications on the Computational Grid”, Proceeding of 2000 International Workshop on Grid Computing (GRID’2000), Lecture Notes in Computer Science series, Vol. 1971, pp. 214-227, 2000.

[HS+01b] E. Heymann, M. A. Senar, E. Luque and M. Livny, “Self-Adjusting Scheduling of Master-Worker Applications on Distributed Clusters”. Proceedings of Euro-Par 2001 Parallel Processing, LNCS series, Vol. 2150, pp. 742-751, August 2001.

Our final objective was to consider the effects caused by machine pre-emption in opportunistic environments. We carried out certain experiments in order to evaluate the number of machine losses incurred during the execution of a master-worker application. These results led us to focus on the study of the cases in which only one machine is lost in an iteration of the application, as this was the situation that was more likely to occur in practice.

By simulation, we then evaluated the impact of machine reclaim on both efficiency and execution time, concluding that a noticeable degradation may be incurred when the application is executed using a number of machines that belong to the ideal interval and one machine is lost. In order to alleviate this effect, we proposed strategies, evaluated by simulation, based on both task replication and on using additional machines. One strategy (denoted as Complete Replication) used the extra machines to replicate the largest application tasks. The second strategy (denoted as Single Replication) only used one of the machines to replicate the largest task. The third strategy (denoted as No Replication) replicated no task at all.

We evaluated the strategies in order to obtain an understanding of their scalability properties and their effectiveness in scenarios in which different probabilities of machine loss were assumed. Our results show that, in general, Complete Replication performs better than the other two strategies, when the application uses a number of machines bigger than or equal to the ideal interval for machines. No Replication tends to perform better when the application uses a small number of machines (fewer than the ideal interval), or when the execution time for all tasks is very similar and the number of extra machines is only one. In general, the results show that the effectiveness of each strategy basically depends on the temporal characteristics of the tasks and the number of available machines. However, replication proves to be the best choice when the system can deliver all the machines to the application that the application itself requires. If the system cannot provide the application with all the machines requested, then extra machines should be used as ordinary machines within the pool.

We also concluded that the number of extra machines that must be allocated to the application depends on the width of the ideal interval. The specific number of machines to be added ultimately depends on the degree of attenuation that we want to achieve in the overall execution time, bearing in mind that every extra machines implies an fixed penalty in terms of efficiency. Our simulations with 50 and 150 machines shown that significant attenuation could be achieved in most cases by using the CR strategy with only three extra machines. Moreover, we have seen that the attenuation effect of extra machines is significant when the probability of machine loss is high. No extra machines should be used when machine loss exhibits a low-to-moderate frequency.

Finally, an implementation and evaluation of the NR and CR strategies were carried out in a practical environment using a *PovRay* master-worker application. These experiments also confirmed the effectiveness of using extra machines in reducing the negative impact of machine pre-emptions.

Evaluation of machine reclaim impact and the strategies for alleviating such impact can be found in the following publications:

[HS+01a] E. Heymann, M. A. Senar, E. Luque and M. Livny, “Evaluation of Strategies to Reduce the Impact of Machine Reclaim in Cycle-Stealing Environments”, Proceedings of Cluster Computing and the Grid Conference (CCGrid2001), IEEE Press, pp. 320-328, 2001.

[HS+01c] E. Heymann, M. A. Senar, E. Luque and M. Livny, “Effective Use of Resources in Opportunistic Systems”, Proceedings of the 5th World Multiconference on Systemics, Cybernetics and Informatics (SCI2001), Vol. XIV, pp. 242-247, July 2001.

The preliminary implementation of strategies for alleviating the impact of machine loss is described in:

[LH+01] C. López, E. Heymann, M. A. Senar, E. Luque “Estudio de Estrategias para Aliviar los Efectos de la Pérdida de Máquinas en Entornos Oportunísticos”, in the Actas de las XI Jornadas de Paralelismo (Proceedings of the XII Spanish Workshop on Parallel Computing), pp. 69-74. Valencia, Spain, September 2001.

6.2 Current and Future Work

We now outline current and future lines of work, as well as their present degree of development.

In chapter 4, we presented the design and results obtained with our self-adjusting algorithm when it was executed on heterogeneous environments, both for efficiency and execution time. A new idea would be that of representing machine heterogeneity by a cost function. Up to now, we have handled efficiency as a matter of time. It can be more generally redefined as a measure of cost:

$$E = \frac{\sum_{i=1}^{n} \left(T_{work,i} * Cost_i \right)}{\sum_{i=1}^{n} \left(T_{up,i} * Cost_i \right) - \sum_{i=1}^{n} \left(T_{susp,i} * Cost_i \right)}$$

n: Number of workers.

$\boldsymbol{T_{work,i}}$: Amount of time that worker *i* spent doing useful work.

$\boldsymbol{T_{up,i}}$: Time elapsed since worker *i* is alive until it ends.

$\boldsymbol{T_{susp,i}}$: Amount of time that worker *i* is suspended, that is, when it cannot do any work.

$\boldsymbol{Cost_i}$:The cost associated to machine *i*.

In accordance with this definition of efficiency, all work done with homogeneous machines is a particular case corresponding to having $Cost_i$ = 1 for all machines.

With regard to the scheduling policy, we used the execution time average for the previous iterations in determining the order in which tasks will be executed in the next iteration. More sophisticated prediction models can be used in order to support a wider range of master-worker applications. Examples of prediction models include using the last iteration execution times, the average of the last *K* iterations (with low values for *K*), or using complex prediction models such as those used in the analysis of temporal series

(ARIMA models and variants). Selecting a particular predictor would be done on-line, by analysing the degree of success produced by each one of the predictors during the initial iterations. Another open line related to this would be the use of predictors that work only with the position of the tasks in the list, instead of considering execution times, i.e., that try to predict which task will be 1st, 2nd, and so on, by taking in account which task was 1st, 2nd, 3rd, etc., in previous iterations.

In chapter 5, we commented on the initial implementation of the strategies for reducing the impact of machine losses. We continue working on this, and observe that, in particular, additional research must is still required in order to detect, from a practical point of view, the moment at which the scheduling strategy should change its policy from using no replication to replication.

This work has assumed that workers were not very distant from the master, and that the amount of data to be moved was significantly small in comparison to the computation time needed to process it; therefore, the effect of network delays was not considered. We plan to extend the ideas presented in this thesis to a grid environment, where workers can be very distant, perhaps even being located in different continents. In such cases it would be expensive to send only one task to each worker. A scheduler that incorporates ideas aimed at data latency reduction should be developed and tested.

A new line of work would arise if, instead of having a single type of worker, we had workers with different functionality. Simply by having two different workers, all the ideas considered up to now would need to be reviewed and modified. We therefore plan to investigate scheduling strategies in opportunistic heterogeneous environments, with this new scenario in mind.

Conclusions and Future Work

References

[AB95] J. Arabe, A. Beguelin, B. Lowekamp, E. Seligman, M. Starkey, P. Stephan, “Dome: Parallel programming in a heterogeneous multi-user environment”, Technical report TR CMU-CS-95-137, Carnegie Mellon University, Pittsburgh, April 95.

[AD+95] R. H. Arpaci, A.C. Dusseau, A. Vahdat, L. Liu, T. Anderson and D. Patterson, "The Interaction of Parallel and Sequential Workloads on a Network of Workstations", Proceedings of SIGMETRICS 95, pp. 267-278, May 1995.

[AD+96] P. Au, J. Darlington, M. Ghanem, Y. Guo, H. To, J. Yang, “Coordinating heterogeneous parallel computation”, Proceedings of the 1996 Euro-Par Conference, Lecture Notes in Computer Science, Berlin: Springer-Verlag, pp. 601-614, 1996.

[AF+97] D. Abramson, I. Foster, J. Giddy, A. Lewis, R. Sosic, R. Sutherst, “The Nimrod Computational Workbench: A Case Study in Desktop Metacomputing”, Proceedings of the 20th Australasian Computer Science Conference, pp. 17-26, Feb. 1997.

[AGK00] D. Abramson, J. Giddy and L. Kotler, “High Performance Parametric Modeling with Nimrod/G: Killer Application for the Global Grid?”, Proceedings of International Parallel and Distributed Processing Symposium (IPDPS), pp. 520- 528, May, 2000.

[AI98] “Wind: The Production Flow Solver of the NPARC Alliance”, AIAA 98-0935, available from http://www.grc.nasa.gov/www/winddocs/aiaa98/aiaa-98-0935.html.

[Ant97] M. Antonioletti. “Load Sharing Across Networked Computers”. Edinburgh Parallel Computing Centre, EPCC-TR1997-06, Available from http://www.epcc.ed.ac.uk/epcc-tec/documents, December 1997.

[AS+95] D. Abramson, R. Sosic, J. Giddy and B. Hall, “Nimrod: a tool for performing parameterized simulations using distributed workstations", Symposium on High Performance Distributed Computing, Virginia, pp. 112-121, August 1995.

[AS91] G. Andrews, F. Schneider, "Concepts and Notations for Concurrent Processing", Benjamin/Cummings Publishing Company, Inc., 1991.

[Ben90] M. Ben-Ari, "Principles of Concurrent and Distributed Programming", Prentice-Hall Publishers, 1990.

[Ber99] F. Berman, "High Performance Schedulers", in I. Foster, C. Kesselman (editors). "The Grid. Blueprint for a New Computing Infrastructure". Morgan Kaufmann Publishers, Inc, San Francisco, USA, pp. 279-309,1999.

[BC+97] S. N. Bhatt, F.R. K. Chung, F. T. Leighton and A. L. Rosenberg, "On Optimal Strategies for Cycle-Stealing in Networks of Workstations", IEEE Trans. on Computers, Vol. 46, No, 5, pp. 545-557, 1997.

[BDK95] A. Baratloo, P. Dasgupta, Z. Kedem, "Calypso: A Novel Software System for Fault-Tolerant Parallel Processing on Distributed Platforms", Proceedings of the 4th IEEE International Symposium on High Performance Distributed Computing, August 1995.

[BG96] T. B. Brecht and K. Guha, "Using parallel program characteristics in dynamic processor allocation policies", Performance Evaluation, Vol. 27 and 28, pp. 519-539, 1996.

[BK+93] H. Burkhart, C. Korn, S. Gutzwiller, P. Ohnacker, S. Waser. "BACS: Basel Algorithms Classification Scheme". Technical Report 93-03, Univ. Basel, Switzerland, 1993.

[Bla90] D. Black, "Scheduling Support for Concurrency and Parallelism in the Match Operating System", IEEE Computer, Vol. 23, No. 5, pp. 35-43, May 1990.

[BRS97] J. Budenske, R. Ramanujan, H. Siegel, " On-line use of off-line derived mappings for iterative automatic target recognition tasks and a particular class of hardware platforms", Proceedings of Heterogeneous Computing Workshop, pp. 96-110. IEEE Computer Society Press, 1997.

[BRL99] J. Basney, B. Raman and M. Livny, "High throughput Monte Carlo", Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing, San Antonio Texas, 1999.

[BS95] D. Bakken, R. Schilchting, "Supporting Fault-Tolerant Parallel Programming in Linda", IEEE Transactions on Parallel and Distributed Systems, Vol. 6, No. 3, pp. 287-302, March 1995.

[Buy99] R. Buyya (ed.), "High Performance Cluster Computing: Architectures and Systems", Volume 1, Prentice Hall PTR, NJ, USA, 1999.

[BW97] F. Berman, R. Wolski, "The AppLeS Project: A Status Report", Proceedings of the 8th NEC Research Symposium, Berlin, Germany, May 1997.

[BW+96] F. Berman, R. Wolski, S. Figueira, J. Schopf, G. Shao, "Application-Level Scheduling on Distributed Heterogeneous Networks", CD-ROM Proceedings of Supercomputing'96, Pittsburgh, IEEE Society Press, 1996.

[Can98] E. Cantu-Paz, "Designing efficient master-slave parallel genetic algorithms", in J. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. Fogel, M. Garzon, D. E. Goldberg, H. Iba and R. Riolo, editors, Genetic Programming: Proceeding of the Third Annual Conference, San Francisco, Morgan Kaufmann, pp. 455, 1998.

[CD97] H. Casanova, J. Dongarra, "NetSolve: A Network Server for Solving Computational Science Problems". International Journal of Supercomputing Applications and High Performance Computing", Vol. 11, No. 3, pp. 212-223,1997.

[CD98] H. Casanova, J. Dongarra, “NetSolve’s Network Enabled Server: Examples and Applications”, IEEE Computational Science and Engineering, Vol. 5, No. 3, pp. 57-67, Sept. 1998.

[CF+95] N. Carriero, E. Freeman, D. Gelernter and D. Kaminsky, “Adaptive Parallelism and Piranha”, IEEE Computer, 28 (1), pp. 40-49, 1995.

[CK+99] H. Casanova, M. Kim, J. S. Plank and J. Dongarra, “Adaptive scheduling for task farming with Grid middleware”, International Journal of Supercomputer Applications and High-Performance Computing, Vol. 13, No. 3, pp. 231-240, 1999.

[CMZ94] B. Chapman, P. Mehrotra, H. Zima, “Extending HPF for advanced data-parallel applications”, IEEE Parallel and Distributed Technology, Vol. 2, No. 3, pp. 15-27, 1994.

[Con99] Condor Team, “Condor Version 6.2.0 Manual”, University of Wisconsin-Madison, 1999.

[CPG99] D. Culler, J. Pal, A. Gupta, “Parallel Computer Architecture. A Hardware/Software Approach”, Morgan Kaufmann Publishers, Inc., 1999.

[DAC96] A. C. Dusseau, R. H. Arpaci and D. E. Culler, "Effective Distributed Scheduling of Parallel Workloads", Proceedings of SIGMETRICS, pp. 25-36, 1996.

[DQS00] “Distributed Queuing Systems”, information available from http://www.scri.fsu.edu/-pasko/dqs.html.

[Don01] J. Dongarra, “Performance of Various Computers Using Standard Linear Equation Software”, Report available from http://www.netlib.org/benchmark/performance.ps, July 2001.

[EZL89] D. L. Eager, J. Zahorjan and E. D. Lazowska, “Speedup versus efficiency in parallel systems”, IEEE Transactions on Computers, vol. 38, pp. 408-423, 1989.

[Fei94] D. Feitelson. "A Survey of Scheduling in Multiprogrammed Parallel Systems". Technical Report RC 19790 (87657), Revised version from August 1997, IBM Research Division, October 1994.

[FG+98] I. Foster, J. Geisler, W. Nickless, W. Smith, S. Tuecke, "Software Infrastructure for the I-WAY metacomputing experiment", Concurrency: Practice and Experience, Vol. 10, No. 7, pp. 567-581, June 1998.

[FHK98] B. Freisleben, D. Hartmann, T. Kielmann. "Parallel Incremental Raytracing of Animations on a Network of Workstations". Proceedings of the International Conference on Parallel and Distributed Processing Technologies and Applications (PDPTA'98), vol. 3, pp. 1305-1302, July 1998.

[FK97] I. Foster, C. Kesselman, "Globus: A Metacomputing Infrastructure Toolkit", International Journal on Supercomputer Applications, Vol. 11, No. 2, pp. 115-128, 1997.

[FK99] I. Foster, C. Kesselman, "Computational Grids", in I. Foster, C. Kesselman (editors). "The Grid. Blueprint for a New Computing Infrastructure". Morgan Kaufmann Publishers, Inc, San Francisco, USA, pp. 15-51, 1999.

[Fos95] I. Foster, "Designing and Building Parallel Programs", Addison-Wesley, 1995.

[Fox89] G. Fox. "Parallel Computing Comes of Age: Supercomputer Level Parallel Computations at Caltech". Concurrency: Practice and Experience, vol. 1, no. 1, pp. 63-103, September 1989.

[FR+97] D. Feitelson, L. Rudolph, U. Schwiegelshohn, K. Sevcik, P. Wong, "Theory and Practice in Parallel Job Scheduling", Proceedings of 3rd Workshop on Job Scheduling Strategies for Parallel Processing, LNCS series, Vol. 1291, pp. 1-34, 1997.

[GB+94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek and V. Sunderam, “PVM: Parallel Virtual Machine A User’s Guide and Tutorial for Networked Parallel Computing”, MIT Press, 1994.

[Gel95] D. Gelernter, “Parallel Programming in Linda”, Technical Report 359, Department of Computer Science, Yale University, January 1985.

[GF96] V. Govindan and M. Franklin, “Application Load Imbalance on Parallel Processors”, in Proceedings of the 10th International Parallel Processing Symposium (IPPS’96) CD-ROM, IEEE, April 1996.

[GGU72] M. Garey, R. Graham, D. Ullman, “Worst case analysis of memory allocation algorithms”, Proceedings of the 4th ACM Symposium on Theory of Computing, pp. 143-150, 1972.

[GH92] Z. Guo and R. Hall. “Fast Fully Parallel Thinning Algorithms”. CVGIP: Image Understanding. Vol. 55, No. 3, pp. 317-328, May 1992.

[GJK93] D. Gelernter, M. Jourdenais, D. Kaminsky. “Piranha Scheduling: Strategies and Their Implementation”, Department of Computer Science, Yale University, CT 06520, Sept. 1993.

[GK92] D. Gelernter, D. Kaminsky. “Supercomputing out of recycled garbage: preliminary experience with Piranha”. International Conference on Supercomputing, pp.417-427, June 1992.

[GK+00] J.-P. Goux, S. Kulkarni, J. Linderoth, M. Yoder, , “An enabling framework for master-worker applications on the computational grid”, Proceedings of the Ninth IEEE Symposium on High Performance Distributed Computing (HPDC9), Pittsburgh, Pennsylvania, pp 43-50, August 2000.

[GL+96] W. Gropp, E. Lusk, N. Doss, A. Skjellum, “A High-Performance, Portable Implementation of the Message Passing Interface (MPI)

Standard", Parallel Computing Journal, Vol. 22, No. 6, pp. 789-828, Sept. 1996.

[Gra66] R.L. Graham. "Bounds for Certain Multiprocessing Anomalies". Bell System Technical Journal 45, pp. 1563-1581, 1996.

[GH92] Z. Guo and R. Hall. "Fast Fully Parallel Thinning Algorithms". CVGIP: Image Understanding. Vol. 55, No. 3, pp. 317-328, May 1992.

[GMM97] S. Ghosh, R. Melhem, D. Mosse, "Fault-Tolerance through Scheduling of Aperiodic Tasks in Hard Real-Time Multiprocessor Systems", IEEE Transactions on Parallel and Distributed Systems, Vol. 8, No. 3, 272-284, 1997.

[GR96] J. Gehring, A. Reinefeld, "MARS — a framework for minimizing the job execution time in a metacomputing environment. Future Generation Computer Systems, Vol. 12, No. 1, pp. 87-99, 1996.

[GW+94] A. Grimshaw, W. Wulf, J. French, A. Weaver, P. Reynolds, "A Synopsis of the Legion Project", Technical Report CS-94-20, Department of Computer Science, University of Virginia, 1994.

[Hall97] L. A. Hall, "Aproximation algorithms for scheduling", in Dorit S. Hochbaum (ed.), "Approximation algorithms for NP-hard problems", PWS Publishing Company, pp. 1-45, 1997.

[Han93] P.B. Hansen. "Model Programs for Computational Science: A Programming Methodology for Multicomputers". Concurrency: Practice and Experience, Vol. 5, No. 5, pp. 407-423, 1993.

[Haz97] V. Hazlewood, "Cluster Computing: A Survey and Tutorial", published in SysAdmin, March 1997, and available from http://www.sdsc.edu/projects/production/NQE/SysAdmin/SysAdmin_Batch.html.

[Hoc97] Dorit S. Hochbaum (ed.), “Approximation algorithms for NP-hard problems”, PWS Publishing Company, 1997.

[HQ91] P. Hatcher, M. Quinn, “Data-Parallel Programming on MIMD Computers”, MIT Press, 1991.

[HS+00a] E. Heymann, M. A. Senar, E. Luque and M. Livny, “Evaluation of an Adaptive Scheduling Strategy for Master-Worker Applications on Clusters of Workstations”, Proceedings of the 7th International Conference on High Performance Computing (HiPC’2000), LNCS series, vol. 1971, pp. 214-227, 2000.

[HS+00b] E. Heymann, M. A. Senar, E. Luque and M. Livny, “Adaptive Scheduling for Master-Worker Applications on the Computational Grid”, Proceedings of 2000 International Workshop on Grid Computing (GRID’2000), LNCS series, vol. 1970, pp. 310-319, 2000.

[HS+01a] E. Heymann, M. A. Senar, E. Luque and M. Livny, “Evaluation of Strategies to Reduce the Impact of Machine Reclaim in Cycle-Stealing Environments”, Proceedings of the First Cluster Computing and the Grid Conference (CCGrid2001), IEEE Press, pp. 320-328, 2001.

[HS+01b] E. Heymann, M. A. Senar, E. Luque and M. Livny, “Self-Adjusting Scheduling of Master-Worker Applications on Distributed Clusters”. Proceedings of Euro-Par 2001 Parallel Processing, LNCS series, Vol. 2150, pp. 742-751, August 2001.

[HS+01c] L. Hluchy, M. A. Senar, M. Dobruchy, T. D. Viet, A. Ripoll, A. Cortés, “Mapping and Scheduling of Parallel Programs”, in “Parallel Program Development for Cluster Computing: Methodology, Tools and Integrated Environments” edited by J. Cunha, P. Kacsuk, S. Winter, pp. 51-78, Nova Science Publishers, Inc., 2001.

[KF99] N. Kapadia, J. Fortes, “PUNCH: An Architecture for Web-Enabled Wide-Area Network-Computing”, Cluster Computing: The Journal of Networks, Software, Tools and Applications, special issue on High Performance Distributed Computing, pp. 153-164, September 1999.

[KRR88] V. Kumar, K. Ramesh and V. N. Rao, “Parallel best-first search of state-space graphs: a summary of results”, Proceedings of the 1988 National Conference on Artificial Intelligence, pp. 122-127, August, 1988.

[KSW98] K. Krüger, Y.N. Sotskov, F. Werner, “Heuristics for Generalized Shop Scheduling Problems Based on Decomposition”, International Journal of Production Research Vol. 36, No. 11, pp. 3013-3033, 1998.

[KW85] C. Kruskal, A. Weiss. “Allocation independent subtasks on parallel processors”. Transactions on Software Engineering. Vol. 11, No. 10, pp. 1001-1016, October 1985.

[LB+97] M. Livny, J. Basney, R. Raman and T. Tannenbaum, “Mechanisms for high throughput computing”, SPEEDUP, 11, 1997.

[LC88] A. Liestman, R. Campbell, “A Fault-Tolerant Scheduling Problem”, IEEE Transactions on Software Engineering, Vol. 12, No. 11, pp. 1089-1095, Nov., 1988.

[LLM88] M. Litzkow, M. Livny and M. Mutka, “Condor – A Hunter of Idle Workstations”, Proc. International Conference on Distributed Computing Systems, pp. 104-111, June1988.

[LS97] S. T. Leutenegger and X.-H. Sun, “Limitations of Cycle Stealing for Parallel Processing on a Network of Homogeneous Workstations”, Journal of Par. and Dist. Computing, Vol. 43, No. 3, pp. 169-178, 1997.

[McL97] D. McLaughlin. "Scheduling Fault Tolerant Parallel Computations in a Distributed Environment". PhD Thesis, Arizona State University, December, 1997.

[NQE00] "Network Queuing Environment", information available from http://www.cray.com/products/software/nqe.

[NQS94] "Network Queuing System", information available from http://umbc7.umbc.edu/nqs/nqsmain.html.

[NSS99] H. Nakada, M. Sato, S. Sekiguchi, "Design and Implementation of Ninf: Towards a Global Computing Infrastructure", Future Generation Computing Systems, Metacomputing Special Issue, pp. 649-658, October 1999.

[NVZ96] T. D. Nguyen, R. Vaswani and J. Zahorjan, "Maximizing speedup through self-tuning of processor allocation", in Proceedings of the International Parallel Processing Symposium (IPPS'96), CD-ROM, IEEE, 1996.

[OS91] Y. Oh, S. Son, "Multiprocessor Support for Real-Time Fault Tolerant Scheduling", Proceedings of the IEEE 1991 Workshop Architectural Aspects of Real-Time Systems, pp. 76-80, San Antonio, Texas, Dec., 1991.

[Ous82] J. Ousterhout, "Scheduling Techniques for Concurrent Systems", Proceedings of the 3rd International Conference on Distributed Computing Systems, pp. 22-30, 1982.

[PL96] J. Pruyne and M. Livny, "Interfacing Condor and PVM to harness the cycles of workstation clusters", Journal on Future Generations of Computer Systems, Vol. 12, 1996.

[Pre96] A. Prenneis, "LoadLeveler: Workload Management form Parallel and Distributed Computing Environments. 1996. Available form ftp://ftp.austin.ibm.com/pub/lscftp/papers/supeur.paper.ps.

[Pri88] D. Pritchard. "Mathematical Models of Distributed Computation". Proceedings of OUG-7, Parallel Programming on Transputer based Machines, IOS Press, pp. 25-36, 1998.

[RF+96] S. Russ, B. Flachs, J. Robinson, B. Heckel. "Hector: Automated Task Allocation for MPI". 10th International Parallel Processing Symposium, pp. 344-348, April 1996.

[RH00] K. D. Ryu and J. K. Hollingsworth, "Exploiting Fine-Grained Idle Periods in Networks of Workstations", IEEE Transactions on Parallel and Distributed Systems, Vol. 11, No. 7, pp. 683-698, 2000.

[SB99] L. M. Silva and R. Buyya, "Parallel programming models and paradigms", in R. Buyya (ed.), "High Performance Cluster Computing: Architectures and Systems: Volume 2", Prentice Hall PTR, NJ, USA, pp. 4-27, 1999.

[SC+01] M. A. Senar, A. Cortés, A. Ripoll, L. Hluchy, J. Astalos, "Dynamic Load Balancing", in "Parallel Program Development for Cluster Computing: Methodology, Tools and Integrated Environments" edited by J. Cunha, P. Kacsuk, S. Winter, pp. 79-108, Nova Science Publishers, Inc., 2001.

[SG+94] V. Sunderam, A. Geist, J. Dongarra, R. Manckek, "The PVM Concurrent Computing System: Evolution, Experiences and Trends", Parallel Computing Journal, Vol. 20, No. 4, April 1994.

[SGE01] "Sun Grid Engine". Information available from http://www.sun.com/software/gridware.

[SL93] M. Squilante, E. Lazawka. "Using processor-cache affinity information in shared memory multiprocessor scheduling". IEEE Transactions on Parallel and Distributed Systems. Vol. 4, No. 2, pp. 131-143, Feb 1993.

[SM97] M. Sirbu, D. Marinescu, "A scheduling expert advisor for heterogeneous environments", Proceedings on Heterogeneous

Computing Workshop, pages 74-87, IEEE Computer Society Press, 1997.

[SS+96] S. Sekiguchi, M. Sato, H. Nakada, S. Matsuoka, U. Nagashima, “Ninf: Network Based Information Library for Globally High Performance Computing”, Proceedings of Parallel Object-Oriented Methods and Applications (POOMA), Santa Fe, 1996.

[SWB98] G. Shao, R. Wolski and F. Berman, “Performance effects of scheduling strategies for Master/Slave distributed applications”, Technical Report TR-CS98-598, University of California, San Diego, September 1998.

[TH+97] H. Topcuoglu, S. Hariri, W. Furmanski, J. Valente, I. Ra, D. Kim, Y. Kim, X. Bing, B. Ye, “The software architecture of a virtual distributed computing environment”, Proceedings on High Performance Distributed Computing Conference, pp. 40-49, IEEE Computer Society Press, 1997.

[TSS88] C. Thacker, L. Stewart, E. Satterthwaite, “Firefly: A multiprocessor workstation”, IEEE Transactions on Computers, Vol. 37, No. 8, pp. 909-920, August 1988.

[TY86] P. Tang, p. Yew, “Processor self-scheduling for multiple-nested parallel loops”. In proceedings of the International Conference on Parallel Processing, pp. 528-535, Aug. 1986.

[VA+94] V. Kumar, A. Grama, A. Gupta, G. Karypis, “Introduction to Parallel Computing. Design and analysis of algorithms”, The Benjamin/Cummings Publishing Company, Inc., California, USA, 1994.

[Wei84] R. Weicker, "Dhrystone Benchmark Program." Communications of the ACM, Vol. 27, No. 10, pp. 1013-1030, Oct. 1984.

[Wil95] G. Wilson, “Parallel Programming for Scientists and Engineers”. MIT Press, Cambridge, MA, 1995.

[WSH99] R. Wolski, N. T. Spring and J. Hayes, “The Network Weather Service: a distributed resource performance forecasting service for metacomputing”, Journal of Future Generation Computing Systems”, Vol. 15, No. 5-6, pp. 757-768, October 1999.

[WW95] K. Williams, S. Williams, “Implementation of an Efficient and Powerful Parallel Pseudo-random Number Generator”, available from http://www.cs.reading.ac.uk/cs/CCL/rand.epvm95.html.

[WZ97] J. Weissman, X. Zhao, “Runtime support for scheduling parallel applications in heterogeneous NOWS”, Proceeding of the Sixth IEEE Symposium on High Performance Distributed Computing, pp 347-355, 1998.

[ZZ+93] S. Zhou, X. Zheng, J. Wang, P. Delisle. “Utopia: a Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems”. Software – Practice and Experience. Vol. 23, No. 12, pp. 1305-1336, Dec. 1993.