University of Alberta

Library Release Form

Name of Author: Nicholas Kazouris

Title of Thesis: Dynamic Scheduling on a Network of Workstations

Degree: Master of Science

Year this Degree Granted: 1997

. . . .

Nicholas Kazouris
18-20, Ionias Avenue
Athens
Greece, 104 46

Date: . . . .

University of Alberta

Dynamic Scheduling on a Network of Workstations

by

Nicholas Kazouris

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 1997

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Dynamic Scheduling on a Network of Workstations** submitted by Nicholas Kazouris in partial fulfillment of the requirements for the degree of **Master of Science**.

. . . .

Jonathan Schaeffer

. . . .

Duane Szafron

. . . .

Ioanis Nikolaidis

. . . .

Duncan Elliott

Date: . . . .

To my parents

Abstract

Networks of workstations provide an attractive and inexpensive parallel system. However, the workstations and the network are usually shared and heterogeneous. Therefore, it is difficult to develop parallel applications that achieve high performance in such an environment without some help from parallel programming tools. In this thesis, we present two dynamic scheduling algorithms, one for parallel divide & conquer applications and another for master-worker applications. These algorithms attempt to achieve high performance by exploiting the characteristics of these paradigms, and by using information collected at run-time. The algorithms do not need any information from the user or the compiler, although if the user or the compiler provides a cost estimation function, they take advantage of it.

We incorporated these schemes into the Enterprise parallel programming system, and conducted experiments. The experimental results show that the schemes described here have the potential to improve performance and perform better than other simpler and popular scheduling schemes.

Acknowledgements

I would like to take this opportunity to thank my supervisors, Dr. Jonathan Schaeffer and Dr. Duane Szafron, for their patience, persistence, constructive criticisms, and funding.

I am grateful to the members of my examining committee, Dr. Duncan Elliott and Dr. Ioanis Nikolaidis, for taking the time to read and comment on this thesis. Thanks also go to Dr. Hong Zhang, who chaired my thesis defense.

I want to express my special thanks to: Steve MacDonald, for building the Enterprise run-time system that used as the basis of this work; Diego Novillo, for doing a great job in the Enterprise compiler and the maintenance of the Enterprise source repository; Ian Parsons, for his thoughtful comments; and all the other inhabitants of the Parallel Programming Systems Laboratory, Mark Brockington, William Hui, David Woloschuk, Alberto Zubiri, and Roel van der Goot.

Rod Johnson, Carol Smith, and Steve Sutphen provided valuable technical assistance in this project, for which I thank them.

The financial support of the Department of Computing Science and the Department of Accounting & Management Information Systems through Graduate Teaching Assistantships is gratefully acknowledged.

I would also like to thank all my colleagues in the Department of Computing Science for their support and encouragement. They include: Yngvi Bjornsson, Michael Boshra, Andreas and Manuela Junghanns, Yoo-Shin Lee, Dean Mah, and Ping Yuan.

Finally, my thanks go to my goddess, Constance, for rocking my world.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Nowadays, local area networks (LANs) of powerful workstations are ubiquitous. Since most of the machines in those LANs are underutilized, networks of workstations (NOWs) provide an attractive and inexpensive parallel system. However, the machines and the network are usually shared and heterogeneous. As a result, it is difficult to develop parallel applications that achieve high performance in such an environment without some help from parallel programming tools.

1.1 Thesis Motivation

Programmers invest considerably more time and effort to develop, debug and test parallel applications than sequential ones. This is because in parallel applications, developers have to consider issues that do not exist in sequential applications, such as the coordination of processes, deadlock prevention, and serialization of events.

Complete parallel programming system environments (PPS) (like Enterprise [32], HeNCE [5] and Paralex [2]) can ease the burden of development. Usually, to achieve this, high level tools impose restrictions on the type of parallelism and how it can be expressed, thus sacrificing a portion of the performance. For example, Enterprise supports control but not data parallelism, even if data parallel algorithms generally achieve higher performance than control parallel algorithms [31]. Additionally, the run-time system of high level tools has to be general enough to encompass all the possible situations supported by the tool, and do the relevant bookkeeping. This hurts performance too, since such an overhead does not exist in a hand-coded version of the same program.

Since the primary reason for resorting to parallel processing is high performance, programmers are reluctant to use a high level tool if it incurs a large performance penalty. Instead, they use low level tools (like PVM [14]) with better performance but more burdensome development.

Furthermore, most users of parallelism are not programmers of parallel applications by trade. Generally, they do not know enough about the issues affecting the performance of a parallel application. They want the performance gains of parallelism without any of its "hassles." They should not be forced to spend time trying to enhance performance. This time can be better spent on developing the algorithm rather than tinkering with the implementation.

Even if a programmer is willing to invest a lot of effort to increase performance, a multitude of decisions must be made (how to decompose a problem, in how many pieces, etc). Moreover, the resulting application will probably be non-portable across systems with different characteristics. For example, what is considered a coarse-grained task on a multiprocessor may become a fine-grained task on a NOW, if the communication cost is high. Hence, the programming effort will be wasted and have to be repeated when the application runs on a different system.

Usually, the parallel applications run on NOWs consisting of heterogeneous non-dedicated resources. The heterogeneity of the workstations has to be taken into account when the tasks are scheduled on processors. For example, a faster processor should execute larger tasks than a slower one. Also, at any time, users can log on any machine of the NOW, execute programs and log off. As a result, the load of the machines can fluctuate during the execution of a parallel application.

All these considerations increase the complexity of managing an application in such an environment to maximize performance. So, the programmer can use all the help that a PPS can provide.

1.2 Thesis Goals

In this thesis, we attempt to accomplish the following goals. Several of these goals are closely related.

- *High performance.* We cannot overemphasize how important this objective is.

If it was not for performance, programmers would not use parallelism. Ideally, we want to achieve performance comparable to the performance of hand-coded parallel applications.

- *Ease of use.* High performance can usually be attained, if the burden falls on the programmer. The PPS can request the programmer to provide a lot of information about the behavior of an application and make most of the decisions related to performance. However, this will increase the programmer's involvement, and make the PPS intimidating and unusable by users that lack a knowledge of performance issues.

- *Minimization of the modifications to the sequential code.* The code of a parallel application should not contain statements that are inserted by the programmer only to enhance performance. This would increase the programmer's effort, and obscure the parallel algorithm with irrelevant code. This approach would go in the wrong direction (adapting the user's program to the PPS) instead of going in the right direction (adapting the PPS to the user's program). Increasing the effort required by the programmer and the need to alter the code hinders the experimentation that the programmer would do. Ideally, the code of a parallel application should be the same as the code of its sequential counterpart.

- *Minimization of the run-time system overhead.* The administrative duties executed by the run-time system (like sharing the load) introduce some overhead. If this overhead becomes large, there is a possibility that the performance will degrade instead of improve.

1.3 Thesis Summary

To achieve the above goals, we propose two dynamic scheduling algorithms, one for parallel divide & conquer applications, and another for master-worker applications. These algorithms attempt to achieve high performance by exploiting the characteristics of these paradigms, and by using information collected at run-time. The algorithms do not need any information from the user or the compiler, although if the user or the compiler provides a cost estimation function, the algorithms take advantage of it.

The scheduling algorithms have been implemented, and experiments have been conducted to evaluate their performance and compare it with simpler scheduling schemes. The experimental results show that our algorithms have the potential to improve performance.

1.4 Thesis Organization

Chapter 2 provides an overview of Enterprise (the PPS used for the development of our algorithms), and describes the programming and machine model assumed. In chapter 3, the problem of scheduling tasks for parallel divide & conquer applications is presented and a new dynamic scheduling algorithm is proposed and evaluated. Similarly, chapter 4 discusses dynamic scheduling for parallel master-worker applications. Chapter 5 presents our conclusions and states the future directions of this work. Finally, Appendix A describes the environment used for the experiments, and displays the experimental data.

Chapter 2

Preliminaries

2.1 The Enterprise Parallel Programming System

The research described in this thesis uses the Enterprise PPS. This section provides an overview of this system. It is not intended to be a detailed description of Enterprise ([32] and [18] serve this purpose).

Enterprise is an integrated graphical development environment where the programmer can code, compile, execute and debug parallel applications. The development of applications is done in two steps. First, the programmer specifies the parallelism by drawing an *asset diagram*, where each *asset* represents a resource or a group of resources that are used to execute a task (the *metaprogramming model*). Then, the programmer enters the code for each asset using C (the *programming model*). Finally, the precompiler automatically inserts code that takes care of the details of communication and synchronization between the processes. The applications communicate via message passing, and can use any from a number of communication systems.

The rest of section 2.1 describes some features of the metaprogramming model, programming model and run-time system of Enterprise.

2.1.1 Metaprogramming Model

In the metaprogramming model, terminology from a business organization is used to describe the structure of the parallel application. Assets are the building blocks of the organization. Each asset corresponds to a process type in the program, contains a number of functions and has the same name as one of them. Assets can be combined hierarchically to construct complex applications.

The following asset types represent the parallel techniques currently supported by Enterprise:

- *line*, which stands for a pipeline,
- *department*, which is the equivalent of a master with a number of different types of workers,
- *division*, which expresses a divide-and-conquer computation, and
- *service*, which denotes a global repository.

Parallelism can be increased through asset *replication*. When an asset is replicated, multiple processes for this asset are created and placed on different processors. The minimum and maximum number of replicas is user-defined. At run-time, Enterprise is responsible for distributing the work to the asset replicas.

2.1.2 Programming Model

Enterprise uses an asynchronous RPC model to exploit parallelism. In sequential programming, when a function A invokes a function B, A is suspended and B is activated. Instead in Enterprise, when a function A invokes an asset B (so this is a call between processes), B is activated and A continues to run. The synchronization between A and B is done implicitly using the concept of futures (described below). In addition, Enterprise takes care of the passing of parameters from A to B and results from B to A.

Currently, there is no support for global variables in Enterprise programs. Additionally, static variables in functions are not supported. This is because between invocations of a replicated asset there is no guarantee that the values of the static variables will follow the sequential semantics. Therefore, assets cannot have any state information.

Futures

Futures were first introduced by Baker and Hewitt [4], but became popular in Multilisp [17]. Lately, many systems use futures as a synchronization mechanism and as a way to increase parallelism between caller and callee processes (such as Mentat [15], ABC++ [3], and Tera MTA [1]). It is remarkable that in Tera, each memory word has a empty/full bit that is also used for the efficient implementation of futures in hardware. This shows how much Tera's designers believe in the usefulness of futures.

According to the future concept, a future is created when a remote process is called and a result is expected. The future acts as a flag that denotes whether the result has been returned. If the result has been returned, we say that the future has been *resolved*. The process expecting the result blocks only when it needs this result to continue its execution. For example, in the following code fragment:

```
status = func(input_params);
...
...
tmp = status + 1;
```

where `func` is an asset, the current process A will invoke the remote process `func`. A, instead of waiting to store the result of `func` to `status`, continues with the execution of the rest of its code until it reaches a statement that accesses the variable `status`. At this point, if the future has been resolved, A will continue its execution, otherwise it will block until the result arrives.

The concept of futures in Enterprise has some differences from the one generally used. In Enterprise, each call to an asset always generates a future. Hence, the futures need not be declared, they are implicit. In many systems, the programmer has to declare which calls to a function will generate futures. Furthermore, Enterprise futures can only be generated for calls to assets and not for calls to arbitrary functions. Finally, a restriction of Enterprise futures is that they cannot be passed as parameters or returned as results. This decreases the complexity of the run-time system implementation, but at the same time decreases the potential concurrency.

The advantage of the Enterprise futures is that they reduce programming effort, since programmers do not have to modify the sequential code. On the other hand, in the other systems, programmers can avoid using futures when they know that a function call will be fine-grained. So, there will be no performance impact from the execution of fine-grained calls since they will be executed locally and not remotely, like in Enterprise.

Parameter Passing

When an asset is called, Enterprise automatically marshals the input arguments, and sends them to the appropriate process. When a result from an asset call arrives, Enterprise unmarshals the data and resolves the appropriate future. Analogous operations are done when an asset receives the input arguments and generates the results.

Pointers and arrays are allowed to be passed as input and/or output parameters to an asset call. However, the programmer has to annotate these calls by providing the number of elements for each pointer argument. In C, without the annotation, it would be impossible to determine the exact length of the pointers, and thus the system would be unable to marshal and unmarshal the arguments correctly. Note that the run-time system can determine the size of the parameter and result messages before the call is executed (this information is used by our scheduling algorithms).

As we stated earlier, Enterprise does not support global nor static variables. This means that the results of an asset call depend entirely on its input parameters, and will be the same independently of the processor where the call is executed on. Consequently, an asset call can be executed on any processor as long as this processor has the input call parameters. In the following, we will use the term *task* to refer to an asset call.

2.1.3 Run-time System

The run-time system is responsible for spawning processes, scheduling tasks and resolving futures [27].

Specifically, the scheduling of tasks is the responsibility of *managers*. A manager can either be a separate process or be implemented as a part of the process for an asset (in this case, the manager is called *collapsed*). There exists a manager for each asset that is called by other assets. Except for two cases, the manager is collapsed either at the caller or at the callee asset. The exceptions are when the callee asset is a division and when more than one asset calls a replicated asset. Then, the manager becomes a separate process.

2.2 Scheduling

In this work, we use the term *scheduling* to refer to the initial distribution of tasks to processors, as well as the redistribution of tasks from busy to idle processors. The responsibility of scheduling can be left to the compiler, the user, the run-time system or one of their combinations.

Scheduling decisions cannot be made by the compiler alone (static scheduling), since these decisions are irrevocable and there are cases where a dynamic mechanism is needed to balance the load. Note that dynamic scheduling is still needed when the tasks have the same computation cost and the parallel application is executed in a dedicated homogeneous environment. Even in this case, tasks can still have variable execution time. For example, on a NOW where Ethernet is the communication medium, the communication cost will vary since it is influenced by the traffic produced by the other processors. On a multiprocessor, variance can originate from simultaneous accesses to the bus. Moreover, in some PPSs without mature compiler support (like Enterprise) this is a necessity too.

For a specific computation on a specific system configuration, the user will achieve better performance after a number of tuning sessions. In these sessions, the user has to discover by trial-and-error the number of processors, the task grain size and the distribution that delivers the best performance. However, if at least one problem or system parameter changes, then another tuning session may be required. Furthermore, the user must have in-depth knowledge about all the issues affecting the performance of an application to be able to carry out a tuning session.

Consequently, the involvement of the run-time system in the scheduling decisions is necessary. The problem is that the use of the run-time system incurs an overhead during the execution of the application that other approaches (like static scheduling) do not have. In this work, the run-time system makes the scheduling decisions. The user or the compiler can help by providing a cost estimation function that the run-time system uses (look at Section 2.3.1 to see how the user can influence the scheduling decisions made by the run-time system).

2.3 Models

In this section, we describe the programming and machine model assumed in the rest of this thesis.

2.3.1 Programming Model

Our work has been developed on top of Enterprise. This does not mean that the proposed solutions are applicable only to Enterprise. They can be used in any PPS that has the following features:

- support for control parallelism,
- a task will produce the same result independent of the processor and order that it is executed,
- the message sizes of a task and its result are known before the task is executed.

Clearly, many systems provide these features, so our work is not restricted to Enterprise only.

Granularity Concerns

In Enterprise, the user has to decide on the number and size of the tasks that the work will be decomposed into (a system where this decision is made by the system with the help of the user is Mentat [16, 41, 40]). This decision affects the performance of the application. If it yields fine-grained tasks, then the overhead of scheduling these tasks will be noticeable and the performance will degrade. In contrast, if the tasks are very large or very few, load balancing cannot be performed since the run-time system cannot decompose a task into smaller subtasks (and thus some processors will execute the large or few tasks while others will stay idle). This brings up the question about what should be the number and the granularity of the tasks provided by the user. Here, we do not attempt to answer this question. We do not assume that the user will make the perfect decision, but just a reasonable one that will not be an obstacle to the scheduling algorithms or compromize the performance.

2.3.2 Machine Model

We assume that the parallel applications will run on a NOW. Each workstation can have different processing power. We assume that the load fluctuations by other activities are momentary (like in [20]). The network is shared between all the workstations and is homogeneous (that is, communication does not transmitted through a router or a gateway).

2.4 Measures

2.4.1 Performance Measures

The measure we aim to minimize is the total completion time of the parallel application. To evaluate our scheduling algorithms we will use two measures, speedup and efficiency. The *speedup* S_p of a program executed on p processors is the ratio between the time taken by one processor to execute the fastest sequential algorithm of the program (T_{seq}) and the time taken by the p processors to execute the corresponding parallel program (T^p_{par}):

$$S_p = \frac{T_{seq}}{T^p_{par}}$$

The *efficiency* E_p of a program executed on p processors is the speedup S_p divided by the number of processors:

$$E_p = \frac{S_p}{p}$$

Speedup measures the absolute performance of the parallel program, while efficiency measures how well the parallel program takes advantage of the available processors.

2.4.2 Processor Speed

The portion of an application distributed to each processor should be proportional to that processor's speed. For a specific application, the processor speed depends on several parameters, both hardware (processor type, clock speed, amount of cache and primary memory, etc), and software (operating system, compiler, programming language, etc). It is difficult to find which parameters influence the performance of the application, quantify their influence, and combine these numbers into a single number describing the processor's speed. Therefore, we should use a metric that provides a single number, and captures the cumulative influence of all the parameters.

One possible metric is MIPS, or MFLOPS (for example, MFLOPS was used in [16]). Almost always, both these numbers quantify the peak performance of only one instruction under ideal conditions. Thus, they are meaningless. For example, if MFLOPS is based on register floating-point additions, this number does not take into account the cost of accessing the memory. Or, if MIPS is used to compare a CISC with a RISC processor, then this is like comparing apples and oranges, since the complexity and the number of instructions needed for the same program is different between these two architectures.

Another metric is one of the popular benchmarks, such as Linpack [38], Dhrystone2 [38], or SPEC95 [33]. Each of these benchmarks tries to be representative for a category of applications. However, since these benchmarks are generic, they do not match the exact characteristics of the application at hand. For example, Linpack is representative of numerical applications, but it does not contain any floating-point divisions. Besides, there are cases where the benchmarks are not even representative for the class of applications they claim to represent. From measurements, Calder et al. [6] concluded that Dhrystone2 fails to capture some of the important features of C system programs. On top of that, because of marketing reasons, vendors tend to optimize their compilers for the benchmarks, thus giving an erroneous indication of the true processor speed. This was why the SPEC organization dropped the `eqntott` benchmark from its SPEC95 suite [39]. At the same time, it is burdensome for the user to pick the right benchmark, since this requires a good knowledge of the characteristics of her application and of the available benchmarks.

Therefore, only the application itself can be a good descriptor of the processor speed. Unfortunately, unresolved issues exist even here. The problem size during benchmarking must correspond to the task size during production runs. If the problem size is too small, we end up measuring the processor speed for the run-time system. Moreover, a small problem size is influenced more heavily from the sizes of the cache and I/O buffers. On the other hand, maybe the programmer's intention is to take advantage of the cache, and so the tasks are created small enough to fit into the cache of any of the processors.[1] In this case, if two processors have the same characteristics except for the amount of cache, then the use of a larger problem size during benchmarking will incorrectly overestimate the speed of the processor with the larger cache.

Furthermore, when the used algorithm decomposes a problem into non-uniform tasks, or when the task size depends on the problem size, the tasks given to a processor may have different sizes. Fortunately, if the tasks do not fit into the cache, there is no need to consider the influence of the different task sizes. From experiments done in [44], it was observed that when the same application is executed with different problem sizes that do not fit into the cache, there is not much variance in the normalized processor speed.

Clearly, it is difficult to take all the above issues into consideration. Thus, we use a simpler method to measure the processor speed. On each machine (when it is idle), we execute the entire application sequentially for the same problem size. The programmer should select the problem size to either fit into the cache of every machine (if the cache is important), or not fit into any cache. Since we are only interested in the relative speeds of the processors, we normalize the speeds. For a specific application, we define the *normalized processor speed* of a processor P as the ratio of the execution time on the slowest processor to the execution time on P. The same definition is also used in [44, 45].

[1] In fact, this is one way to achieve superlinear speedup [31].

Chapter 3

Dynamic Scheduling for Divide & Conquer Applications

3.1 Overview

In this chapter, we examine scheduling for parallel divide & conquer (D&C) applications. We discuss their characteristics and problems, and present a number of different approaches used for scheduling. Then, we describe our dynamic scheduling algorithm. Finally, we conduct experiments and compare the performance of our algorithm with that of a static scheduling algorithm.

3.2 The Divide & Conquer Paradigm

Parallelism is inherent in the D&C paradigm. Each D&C computation recursively partitions the *original problem* into independent subproblems that can be solved concurrently, and combines their solutions to form the solution for the original problem. Each subproblem can be stored as a task that contains enough state information that it can be executed on any processor. Here, the term *subproblem* refers to an abstract problem and *task* refers to the incarnation of a subproblem on a particular processor.

A *computation tree* can be used to represent a D&C computation. Each node represents a problem, and its children represent the subproblems that this problem is divided into. The tree's root corresponds to the *original problem*, and the leaves correspond to the *base cases*[1] of the D&C computation. An example of a part of a computation tree is shown in Figure 3.4 on page 27.

[1]The subproblems that are solved non-recursively.

The plethora of opportunities for parallelism in the D&C paradigm can become an obstacle to good parallel performance. Tasks should correspond to large subproblems, so that the benefits of possible remote execution outweigh the costs of transferring these tasks to other processors. Hence, few tasks closer to the root of the computation tree should be generated. However, if few tasks are generated, then some processors may not find a task to execute, will stay idle, and parallelism will be lost.

On the other hand, tasks should correspond to smaller subproblems, so that load balancing can be achieved more easily. This implies that many tasks closer to the leaves of the computation tree should be generated. But this will incur excessive overhead from converting the subproblems to tasks and storing the tasks into the local task repository. Therefore, we must decide how many and which subproblems will be converted to tasks and which of these tasks will be executed remotely.

Each D&C algorithm attempts to divide a problem into subproblems having an equal amount of work. This cannot always be achieved because of the nature of the algorithm at hand (like in the computation of Fibonacci numbers), or because the data cannot be divided into equal pieces (as in multiplication of odd dimension matrices). Even in a perfectly balanced computation, subproblems have to be moved between processors and this communication cost is added to the computation cost of these subproblems producing unbalanced computation trees. Therefore, a mechanism for load balancing is needed.

Because of the form of the D&C computations, most of the processors are idle at the beginning (where not enough tasks have been created yet) and at the end (where there are no more tasks to process) of the computation. At the end of the computation, nothing can be done to decrease the idle time (only implicitly, by better balancing the load). Still, at the beginning of the computation, the idle time can be decreased if a task is sent to a processor as soon as it is created.

Generally, the internal and leaf nodes of a computation tree have different computation costs. For some applications most of the work is done at the leaves (like in the Mandelbrot set computation), while for others most of the work is done at the internal nodes (as in the computation of Fibonacci numbers). As a result, a distribution of tasks to processors should not be merely based on the number of tasks.

To make scheduling easier, we could assume that the *degree*[2] of a D&C computation is always equal to two. This is restrictive, since there are D&C algorithms that have higher degree (for example, in matrix multiplication the degree is equal to eight). Of course, all the algorithms can be expressed as having degree two. If the degree of an algorithm is equal to a power of two 2^k ($k > 1$), then each division can be expressed as k consecutive divisions of a corresponding algorithm with degree two. If the degree is not equal to a power of two, we can add empty subproblems until the next power of two is reached, and then to follow the above procedure. However, this increases the effort of the programmer, and obscures the algorithm with irrelevant code.

Our goal is to devise a scheduling algorithm that takes into account the characteristics of the D&C computations and addresses these issues.

3.3 Related Work

In this section, we present the different scheduling schemes used for the parallel execution of D&C computations. Note that the works of Mohr et al. [28], Wagner [36, 37] and Chakrabarti et al. [7] deal with general recursive computations, and not specifically with D&C computations. Some experimental results from these works are shown in Table 3.1 on page 21 (for each application, we selected the number of processors that provides the best efficiency). These results should not be used to compare the performance of the different approaches, since the assumptions, hardware and goals of these projects are different. They are presented to give an idea about the different D&C computations used for parallel execution, and the magnitude of speedup and efficiency someone can typically expect from the parallel execution of D&C applications.

In the most simplistic scheduling scheme for D&C computations, the processors form a complete binary tree. Each processor divides the problem it receives from its parent, distributes the two newly generated tasks to its children, and waits idle until both its children return their solutions. Then it combines the solutions, and passes the results to its parent. Consequently, most of the computation is done on the leaf

[2] The number of subproblems that a problem is recursively divided into at each level.

processors, leaving the internal processors idle for a potentially long duration. This scheme was used in [8] and Enterprise [18].

3.3.1 "Keep half, send half" Scheduling Scheme

"*Keep half, send half*" is a popular scheduling scheme [21, 25, 46, 26, 13, 10]. In this scheduling algorithm, when a processor divides a problem into two subproblems, it sends one to an idle processor and executes the other itself. If a processor cannot find an idle processor to execute the other half, it executes both subproblems. This approach assumes that the D&C computation is always divided into two subproblems resulting in a full binary computation tree. Lo et al. [25] show that with the "keep half, send half" scheme, the computation tree maps to a binomial processor tree.[3]

The processor that receives a subproblem can be selected either statically or at run-time. In static selection, each processor sends the subproblem to a specific child in its binomial processor subtree. As a result, the run-time system knows instantly if a processor is idle and avoids the overhead of having to find one. On the other hand, processors that could work on a subproblem from a different parent stay idle. Thus, the best performance is obtained when the division of each problem creates two subproblems with the same amount of work, and the number of processors is a power of two (since a binomial tree B_k has 2^k nodes). Additionally, it is important how the binomial tree is mapped onto a specific architecture. Static mappings have been proposed for hypercubes [25], 2-dimensional meshes [25, 26], and binary de Bruijn networks [46].

Run-time selection is used in the work of Freisleben and Kielmann [13]. When a processor has work to send (sender), it asks the *scheduler* (which is unique in the whole system) for the address of an idle worker. When the worker finishes and returns the results, the processor informs the scheduler that the worker is idle again. A work transfer in this scheme is initiated by the sender. This is beneficial at the beginning of the computation where the number of idle processors is large. However, as the computation progresses, most of the processors will become busy and it will be increasingly difficult for the scheduler to find an idle processor. So, most of the requests

[3] A *binomial tree* B_k is defined recursively. B_0 is a single node, and B_k consists of two binomial trees B_{k-1} where the root of one is the leftmost (or rightmost) child of the root of the other.

from the senders will not be served, but as a result of the increased communication, the scheduling overhead will increase and the scheduler will become a bottleneck.

In this system the compiler automatically parallelizes sequential D&C programs, but the problems must always be divided into two subproblems, which is restrictive. The programs are written in C with some extensions used to describe the formal function parameters (similar to the Enterprise extensions).

3.3.2 APRIL

APRIL [10] is a programming language and system developed for the automatic parallelization of sequential D&C applications. It uses PVM for communication. The language's syntax and semantics are similar to that of Pascal. The designer of APRIL could easily use Pascal and add extensions or impose limitations where needed, instead of forcing programmers to learn a new programming language. The APRIL code passes through a precompiler and is converted to C. Exactly two statements can be executed in parallel and must be enclosed in a `parbegin-endbegin` construct.

The "keep half, send half" scheme is used for scheduling. Before program execution, the user specifies the binomial processor tree to use for the computation. The processor where a subproblem will be sent is selected statically. A run-time library function, `child_available()`, is exposed to the programmer and can be used to check if there is an idle child processor. Thus, it can be decided at run-time whether a problem should be partitioned further or executed sequentially by a more efficient algorithm than D&C.

3.3.3 Inlining and Lazy Task Creation

Mul-T [22] is a parallel implementation of Scheme [9]. The `future` construct is used to express the parallelism. The programmer identifies which computations can be executed in parallel by using expressions of the form `(A (future B))` to declare the potential parallelism, where `B` is the computation that can become a future and executed concurrently with its continuation `A`.

The run-time system decides which of these computations will have a separate process created for it. To throttle the process creation in a loaded system and increase the granularity of the caller task, two run-time schemes are considered [28]. In the first scheme, *load-based inlining*, the "future call" is executed as a separate process when the system is not loaded, and executed in the same process as its continuation when the system is loaded. The system is considered loaded when the number of processes stored in the process queue is greater than a user-supplied threshold T. In the other scheme, *lazy task creation*, each "future call" is executed in the same process as its continuation, but its continuation can be stolen by an idle processor. The idle processor always steals the oldest available continuation.

As Mohr et al. state in [28], load-based inlining is inferior to lazy task creation, because with inlining, the programmer has to put in more effort (provide the threshold), the decision is irrevocable, and sometimes too many processes are created. On the other hand, in the implementation of lazy task creation, enough information must be maintained to allow another processor to steal the continuation of the provisionally inlined future. Since this requires the splitting of the existing stack (which means copying of continuous stack frames or pointer manipulation of linked stack frames), the lazy task creation approach can be applied efficiently only on a multiprocessor or on specialized hardware. More importantly, stack manipulation makes this approach highly non-portable.

3.3.4 Leapfrogging

In *Leapfrogging* [36, 37], the target language of the system is C++. Futures, which are instances of the class `Future`, are used to define which computations can be executed concurrently. Each processor has a FIFO queue of tasks, and can insert tasks into and remove tasks from any queue. By default, a processor inserts tasks into its own queue. However, the user can specify another queue where the task should be inserted and play an active role in load balancing. Unfortunately, since the user does not have any load information about the remote queues, it is hard to make intelligent decisions.

Each processor removes tasks from its local queue until either its queue is empty, or it is waiting for a future (say F) to be resolved. In the first case, the processor tries to steal a task from a remote queue. In the latter case, the processor does not stay blocked waiting for F to be resolved. Instead, it attempts to get one of the futures created by F (if one exists) and resolve it. That is, it executes part of the work needed to resolve F. Since the same action can be taken by the processor executing F, we have a situation where one processor *leapfrogs* over the other. Each time leapfrogging occurs, the obtained task is closer to the leaves of the computation tree. Hence, with increased probability, its granularity will be finer and its execution on a different processor will be wasteful, even on a multiprocessor (which is the architecture the algorithm was proposed for).

3.3.5 Central Queue

In [21, 43], a central queue is used to balance the load between the processors. In [21], a FIFO queue resides on one processor. Each processor removes a subproblem from the beginning of the queue and takes one of three actions: divides the subproblem further and stores the generated subproblems at the end of the queue, solves the subproblem and stores the result, or combines two solutions from the queue and stores their result.

In [43], each processor maintains a local stack where it stores its generated subproblems, and one processor maintains a *global pool* (GP) that at first stores the original problem. Each processor divides subproblems in a depth-first order and when its local stack is empty, it requests a subproblem from the GP. When the GP has given all its subproblems, it requests from all the processors the subproblems in their local stacks at the lowest level. This procedure is repeated until the end of the computation.

Since the above algorithm introduces contention to the processor that maintains the GP, Wu [42] attempts to avoid this problem by assigning a local GP to each processor. Here, load sharing is done by using a redundant binary processor tree. The embedding of such a binary tree on k-dimensional mesh, hypercube and perfect shuffle interconnections is given in [42].

3.3.6 Randomized Scheduling Scheme

In [19, 7], a randomized load sharing algorithm is considered for tree-structured task graphs produced from branch-and-bound and exhaustive search computations. Exhaustive search can be thought of as a D&C computation where the combine step is missing. In this algorithm, every processor has a local queue. When the processor finishes its current work, it selects the "best" task (based on a user-supplied cost function) from its local queue for execution. When a processor generates a task, it sends this task to the queue of a randomly chosen processor. For branch-and-bound applications, each processor periodically broadcasts the minimum cost of the leaves it has expanded so far, so that all the processors know the global minimum cost.

In the results shown in Table 3.1, the data were replicated on all the processors beforehand, so locality does not affect the execution times.

Ref.	*Hardware & Communication*	*No of procs*	*Application*	*Speed up*	*Efficiency %*
[28]	ALEWIFE (detailed simulator)	16	20-th Fibonacci number	8.06	50.37
			speech understanding s/w	10.94	68.37
			8 queens problem-all solutions	12.53	78.31
[36]	Sequent Symmetry B (multiprocessor)	16	10 queens problem-all solutions	12.00	75.00
			gamma function	14.00	87.50
[13]	MEIKO transputer (asynchronous RPC)	32	quicksort	1.10	3.43
			matrix multiplication	3.00	9.37
			adaptive integration	10.00	31.25
			prime factoring, set covering	12.00	37.50
			knapsack	24.00	75.00
[7]	CM-5	15	Gröbner basis problem	11.00	15.00
		60	bisection eigenvalue problem	30.00	50.00
[11]	HP 9000/720 (PVM on Ethernet)	16	chip layout program	1.40	8.75
		16	Mandelbrot	9.31	58.21

Table 3.1: Performance results of the different scheduling schemes for D&C applications.

3.4 Proposed Scheduling Algorithm

In this section, a new scheduling algorithm is presented. It contains many of the elements used in the above approaches while eliminating many of their restrictions, as well as minimizing user involvement. An outline of the algorithm will be given, followed by the details for each component of the algorithm. Pseudo-code for the algorithm is shown in Figure 3.1 (for the manager), and Figures 3.2 and 3.3 (for the workers).

```
localQueue // the local queue of unscheduled original problems
idleWrkList, busyWrkList // lists containing the idle and the busy workers

for ever {
    msg = blocking receive;
    W_j = worker sending the message;
    switch (msg.Tag) {
        case TASKS :
            // Original problem from the caller
            store original problem in localQueue;

        case IDLE_WORKER :
            // Notification from a worker that it is idle
            add W_j to idleWrkList;

        case INFO :
            // Information from a worker about the sum of the heuristic values
            // of all the tasks in its queue
            store info of W_j;

        case REQ_FAILED :
            // the request for work transfer has failed
            reset info of W_j;
            add the corresponding idle worker to idleWrkList;

        case REQ_SUCCEEDED :
            // the request for work transfer has succeeded
            store info of W_j;

        case WORKERS :
            // Addresses of the workers that did not receive tasks
            add workers to idleWrkList;
    }
    while ((at least one worker is idle) and (there exist tasks locally or at the workers)) {
        W_idle = fastest idle worker;
        if (localQueue is not empty) then {
            // send an original problem from the local queue
            coworkers = list containing the rest of the idle workers;
            send original problem, coworkers (tag = TASKS) to W_idle;
            add W_idle, coworkers to busyWrkList;
        } else {
            // transfer work from a busy worker
            W_busiest = busy worker with the highest (Σ heur_{W_j}) / nSpeed(W_j)
                that is not involved in another task transfer;
            send W_idle (tag = TASKS_REQUEST) to W_busiest;
        }
    }
}
```

Figure 3.1: Pseudo-code for the manager of a D&C application.

```
localQueue // the local queue of unscheduled tasks
coworkers // list containing the workers received by the manager

set an alarm that calls periodically MsgPolling();
for ever {
    while (localQueue is not empty) {
        if (I have not sent info about the remaining tasks
                in my localQueue to the manager) then {
            // send to the manager the sum of the heuristic values of the tasks in my queue
            send info (tag = INFO) to manager;
        }
        if (number of generated tasks < total number of workers) then {
            // More tasks should be generated
            t = remove highest cost task from localQueue;
            divide t and store generated tasks in localQueue;
            if (coworkers is not empty) then {
                // We are at the beginning of the computation,
                // so send tasks to other workers
                nLists = number of tasks in localQueue;
                split the coworkers list into nLists lists;
                distribute tasks and lists (tag = TASKS);
            }
        } else {
            // Inline tasks
            t = remove lowest cost task from localQueue;
            solve t;
            resolve the future that corresponds to t;
        }
    }
    if (coworkers is not empty) then {
        // No more work exists, so send any remaining addresses of workers to the manager
        send coworkers (tag = WORKERS) to manager;
    }
    // the reply is sent
    send solution (tag = REPLY) to the creator of the received task;
    // notify the manager that I am idle
    send (tag = IDLE_WORKER) to manager;
    blocking receive;
}
```

Figure 3.2: Pseudo-code for the worker of a D&C application.

3.4.1 Outline

There are three different types of processes used in this algorithm. First, there is the *caller* process which is the source of the original problem to be solved (tag = TASKS) and the destination of the solution. Then, there exists a number of *worker* processes that work on the solution of tasks. Finally, there is a *manager* process that

```
function MsgPolling() {
    msg = non-blocking receive;
    switch (msg.Tag) {
        case TASKS :
            // tasks received from the manager or another worker
            store task in localQueue;
            store list of workers in coworkers;

        case REPLY :
            // reply received from another worker
            resolve the future that corresponds to the solution received;

        case TASKS_REQUEST :
            // request from the manager to send work to another worker
            if (localQueue is empty) then {
                // notify the manager that no task was sent
                send (tag = REQ_FAILED) to manager;
            } else {
                // a task is sent to the idle worker
                t = remove second highest cost task from localQueue;
                send t (tag = TASKS) to idle worker;
                // notify the manager that a task was sent
                send (tag = REQ_SUCCEEDED) to manager;
            }
    }
}
```

Figure 3.3: Pseudo-code for the worker of a D&C application (*continued*).

communicates with the caller and coordinates the workers.

At the beginning of the D&C computation, the manager receives the original problem from the caller. It selects an idle worker where it sends the problem to, along with the addresses of the other idle workers. The worker uses these addresses to send new tasks directly to the idle workers.

Each worker maintains a local task queue, which is initially empty. When the worker receives a problem (tag = TASKS), it recursively divides this problem into subproblems until a certain number of subproblems have been generated and places the corresponding tasks in its local queue. Then, it solves the generated subproblems directly without further division. After the worker solves the problem it received, it returns the solution to the process that generated the problem (tag = REPLY), and notifies the manager that it is idle (tag = IDLE_WORKER). When the manager is notified by the idle worker, it requests that a busy worker provides a task to the idle worker (tag = TASKS_REQUEST).

3.4.2 Details

Computation Cost of the Tasks

Tasks have varying computation costs. In the next sections, we will see that many of the decisions in our scheduling algorithm take into account the computation cost of the tasks. Therefore, we assign to each task a value, called its *heuristic value*, that represents a positive number proportional to the computation cost of the task. It is computed when the task is created (i.e. when the corresponding asset is invoked). Note that to decrease the scheduling overhead, the computation of the heuristic value should be as fast as possible.

Generally, the computation cost for different invocations of the same asset depends on the size of its input and output parameters, as well as the values of its input parameters. For example, the computation cost of an asset that calculates a dense matrix product depends only on the size of the parameters, while the cost of an asset that calculates a Fibonacci number depends only on the value of the Fibonacci number to be calculated.

Based on our philosophy of minimal user involvement, the ideal scenario would consist of the compiler automatically generating an estimation function, where the only variables are the asset parameters. At run-time, the heuristic value of an asset invocation could be calculated by calling this estimation function with the same parameters that the asset itself is called with. Currently, the use of a compiler is not a workable option, since the necessary compiler technology is not available. The short-term solution is to let the user provide the estimation function (similar to the call-back functions used in Mentat [16, 41, 40]). We expect that the user's function will always provide more accurate estimates than the compiler-generated function. However, the burden will fall on the user.

There are cases where the estimation function cannot be determined by the user (for example, the user knows few things about the computational complexity of the algorithm used), or the compiler (for example, the asset calls third-party functions where the source code is not available). In these situations we have to use other measures. Since the run-time system does not have any knowledge about the semantics of the algorithm used, we can compute the heuristic values using only externally observable metrics.

One simple approach to generating a heuristic value is to use the total size of the input and output parameters of an asset invocation. The rationale of this measure is based on the assumption that in an asset invocation, more calculations must be performed for more data to be consumed and/or produced.

If the parameters of the different invocations have the same size, then we use the height of the task in the computation tree as its heuristic value. When an invocation of a D&C asset is closer to the root of the computation tree there is a higher probability that this invocation will need more computation than another invocation which is deeper in the computation tree. Here, the assumption is that an invocation deeper in the computation tree has gone through more divisions which results in a smaller subproblem with less computation. This is always true when each problem is divided into subproblems with the same computation cost, or for invocations in the same root-leaf path of the computation tree. But there are cases where this measure is far from perfect (like for example at the calculation of a Fibonacci number).

Thus, when a worker wants to select from its queue the task with the highest cost, this would be the task with the largest heuristic value (if the measure is the user-supplied estimation function or the size of the parameters), or the task with the smallest heuristic value (if the measure is the height of the task in the computation tree).

The measure that will be used is selected at run-time, when the original problem is divided into subproblems. If the user-supplied estimation function exists, it is always preferred, since it is based on better knowledge about the algorithm and should approximate the computation cost more closely. Otherwise, if the total size of the parameters of the first generated subproblem is different from the total size of the parameters of the original problem, the size is used as the measure. Finally, if the above condition is false, the invocation height is used.

The measure that the workers should use is included in each message containing tasks (tag = `TASKS`). In this way, the measure is propagated from the worker that decides which measure to use (this is the worker where the original problem was sent) to the other workers, so that for the entire computation the same measure is used. If the workers used different measures, it would be difficult to compare heuristic values from different measures. In addition, the local queue of each worker is sorted using the heuristic value. So, if a task needs to be removed from the queue using another measure, then the queue must either be sorted again or the time to select a task from the queue using the new measure would increase.

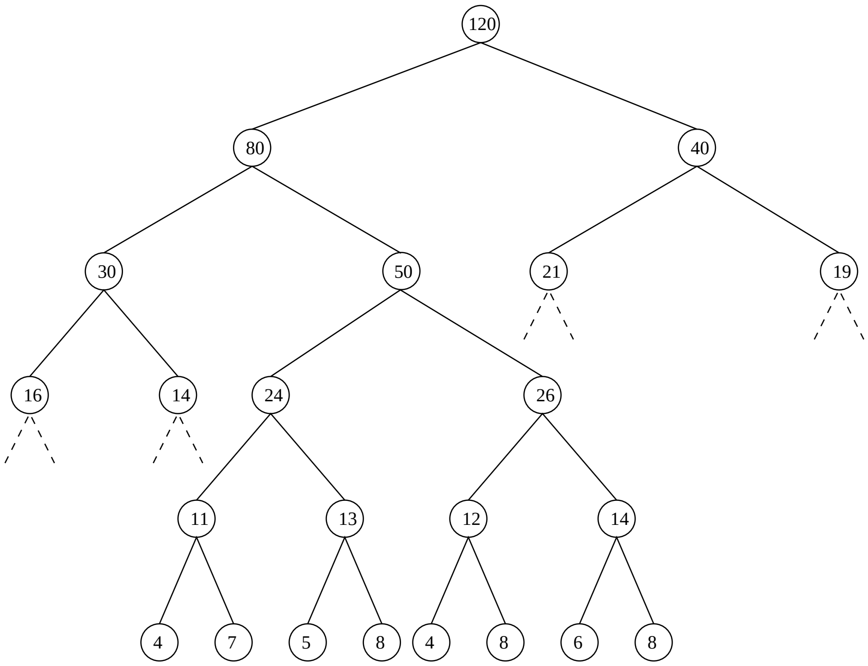


Figure 3.4: Part of the computation tree of a D&C application.

In the following sections, we will use as an example a D&C application where a part of its computation tree is shown in Figure 3.4. The measure used for the heuristic values is the total size of the parameters. The number within each node of the tree denotes its heuristic value. We assume that this computation is to be executed using four workers (W_1, W_2, W_3 and W_4), where $nSpeed(W_1) \geq nSpeed(W_2) \geq nSpeed(W_4) \geq nSpeed(W_3)$ ($nSpeed(W_j)$ denotes the processor speed of worker W_j).

Beginning of the Computation

As mentioned earlier, the duration when the workers are idle at the beginning of the computation can be decreased by providing tasks to these workers as soon as they are available. If the manager is used as an intermediary (as it is used when all the workers are busy and one of them becomes idle), then the worker generating the task would have to notify the manager, and the manager would have to direct where to send the task. If the worker generating a task knows beforehand the address of an idle worker, it can send the task directly to this worker and avoid the overhead of going through the manager.

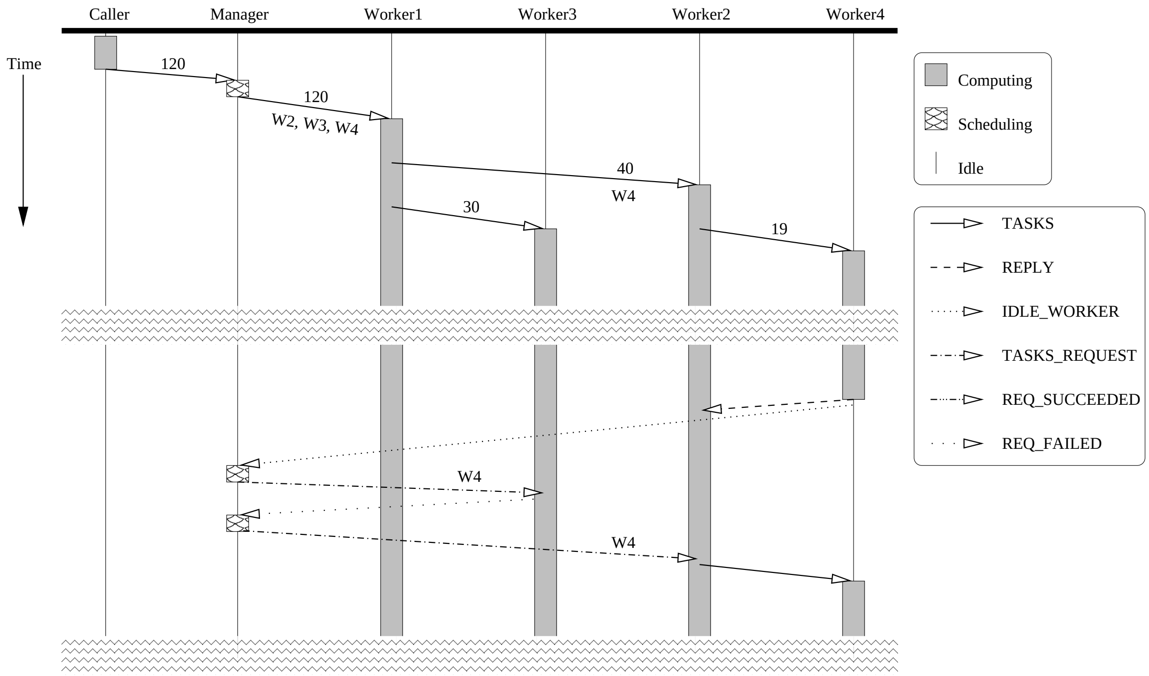


Figure 3.5: Message exchange at different phases of the D&C scheduling algorithm.

In our scheme, when the manager receives the original problem from the caller, it chooses the fastest idle worker (W_1) and sends the original problem ("120") to this worker (tag = `TASKS`). If the manager knows of other idle workers, it sends their addresses together with the task (W_2, W_3 and W_4) to W_1. After dividing the received problem into subproblems, the worker (W_1) adds itself to the list of workers it has. It splits this list into as many lists as the generated subproblems, attempting to keep balanced the sum of the processor speeds in each list. In our example, two subproblems are generated having heuristic values "80" and "40", so two lists are created, one containing W_1 and W_3, and the other containing W_2 and W_4. For each list (except for the list where W_1 is a member), W_1 selects the fastest worker and sends the task with the second highest cost to this worker, along with the other workers in this list. Here there is only one list not containing W_1, the list of W_2 and W_4. So, W_1

sends the task "40" to W_2, along with W_4. Then, the worker removes from the queue the remaining task and goes through the same procedure until no more idle workers exist (see Fig. 3.5).

Task Selection from a Worker's Queue

Each worker stores tasks in and removes tasks from its local queue. These actions entail overhead that does not exist when the computation is executed sequentially. To decrease this overhead, at some point the worker should stop creating subproblems that become tasks; it should start executing them sequentially.[4] This action is similar to inlining [28]. None of the subproblems of an inlined problem is stored as a task in the queue.

Moreover, coarse-grained tasks need to be generated for potential remote execution. To accomplish that, each worker selects from the queue the task with the highest cost, because this task will generate at least one subproblem with high computation cost. This division should not continue *ad infinitum* because finally the generated tasks will become too fine-grained. So, the worker continues this operation until the number of generated tasks in the queue is equal to the number of workers. Note that the number of generated tasks is counted after all the other workers sent to this worker have received a task. The threshold used here is simplistic. The rationale is to create one task for each of the workers, for the case where all the other workers become idle before this worker. A more sophisticated threshold would need to take into account many parameters that would increase the overhead required to gather the necessary information and decide on when to stop creating tasks.

After the specified number of subproblems have been generated, the worker selects the task with the lowest cost, inlines it, and continues until its local queue becomes empty. The advantage of selecting the lowest cost task is that the larger tasks (which usually have coarser granularity) are left for transfer to other workers. On the other hand, it is possible to increase the load imbalance at the end of the D&C computation, since a worker may be "stuck" with a large subproblem to solve while the rest are waiting idle.

[4] There is still more overhead than in the sequential execution that originates from the Enterprise semantics. See Section 3.4.3 on page 32 for more details.

Another problem that would exist if new subproblems were constantly generated instead of being inlined is that excessive amounts of memory would be needed to store the corresponding tasks.

When the worker has to provide a task to another worker, it always selects the task with the second highest cost from the queue. The second largest task is selected instead of the largest one to decrease the period where the worker will wait for the result of the remotely executed task to return.

In our example, the order that W_1 selects tasks for division will be: "120", "80" (after this division, W_1 has sent tasks to all the workers it had), "50" (after this division the tasks in the queue will be: "24", "26"), "26" (tasks in the queue: "12", "14", "24"), "24" (tasks in the queue: "11", "12", "13", "14"). Since now there are as many tasks in the queue as the number of workers, W_1 inlines tasks starting from the smallest one: "11", "12", "13" and "14".

Selection of a Busy Worker

Periodically (currently the period is 4 seconds), each worker sends to the manager (tag = `INFO`) the sum of the heuristic values of all the tasks residing in its local queue ($\sum heur_{W_j}$ denotes the sum of the heuristic values for worker W_j).

When a worker notifies the manager that it is idle (tag = `IDLE_WORKER`) and there is not an original problem in the manager's queue, the manager goes through the list of the busy workers that are not involved in another task transfer. It requests from the busy worker with the highest $\frac{\sum heur_{W_j}}{nSpeed(W_j)}$ ratio to send a task to the idle worker (tag = `TASKS_REQUEST`). If the busy worker's queue is not empty, a task is sent to the idle worker (tag = `TASKS`) and a notification to the manager that the operation was successful (tag = `REQ_SUCCEEDED`). Otherwise, the busy worker notifies the manager that it cannot carry out the request (tag = `REQ_FAILED`). If the request failed, the manager selects another busy worker until it has asked for work from all the busy workers (see Figure 3.5).

```
( 1)  int divisionAsset(parameters) {
( 2)      int subsolutionA,subsolutionB,solution;
( 3)
( 4)      if (base_case) then {
( 5)          solution = straightforward_solution(parameters);
( 6)      } else {
( 7)          divide(parameters);
( 8)          subsolutionA = divisionAsset(parametersA);
( 9)          subsolutionB = divisionAsset(parametersB);
(10)          solution = combine(subsolutionA,subsolutionB);
(11)      }
(12)      return solution;
(13)  }
```

(a) Before being processed by the Enterprise precompiler.

```
( 1)  int divisionAsset(parameters) {
( 2)      int subsolutionA,subsolutionB,solution;
( 3)
( 4)      if (base_case) then {
( 5)          solution = straightforward_solution(parameters);
( 6)      } else {
( 7)          divide(parameters);
( 8)          subsolutionA = _ENT_SendCall_divisionAsset(parametersA);
( 9)          subsolutionB = _ENT_SendCall_divisionAsset(parametersB);
(10)          _ENT_Wait(&(subsolutionA));
(11)          _ENT_Wait(&(subsolutionB));
(12)          solution = combine(subsolutionA,subsolutionB);
(13)      }
(14)      return solution;
(15)  }
```

(b) After being processed by the Enterprise precompiler.

Figure 3.6: A D&C application.

3.4.3 Implementation Details Relevant to Enterprise

Figure 3.6 shows the pseudo-code of a D&C computation before (Fig. 3.6(a)) and after[5] (Fig. 3.6(b)) it has passed through the Enterprise precompiler. The lines with the gray background denote the lines changed by the Enterprise precompiler. When an asset is called to solve a subproblem (line 8), the call is not executed immediately. Instead, all the parameters of this call are stored in the queue and the execution

[5] Some of the calls to the run-time system library have been removed for clarity.

continues (line 9). When the run-time function `_ENT_Wait()` is called, the run-time system first checks if the specified future has already been resolved. If the future has not been resolved (line 10 for the subproblem `subsolutionA`), the run-time system removes a subproblem from the queue and calls `divisionAsset` with the parameters of this subproblem. Since the selection is based on the cost of the subproblem, it is possible that this subproblem is different from the subproblem that would resolve the future. Even so, the work for this subproblem is not wasted, since in a D&C computation all the subproblems have to be solved. For example, suppose that the pseudo-code in Fig. 3.6 produces the computation tree shown in Fig. 3.4. Then, a part of the calling sequence for worker W_1 is shown in Fig. 3.7 on the next page.

Usually, the caller of a D&C computation creates the original problem and then blocks, waiting for the problem's solution. The manager does not do any useful computation and is mostly idle. Since these two processes do not use much processor time, they are placed on the same processor together with a worker.

Previously, in Enterprise, the user had to provide the depth and breadth of the workers to be used for a D&C application [18]. Thus, the user was constructing manually the tree of workers that would correspond to the computation tree. This is not needed anymore. The user needs only to specify the number of workers used by the D&C computation.

Overhead Originating from the Enterprise Semantics

A source of overhead originates from the passing of the input pointers and arrays. Because of the Enterprise parameter passing semantics, all the changes done to an input pointer's data in the callee should not be visible from the caller. Thus, when a call is executed on the same worker as its caller, the data of the input pointers must be copied. Otherwise, the execution on the same worker would violate the Enterprise semantics.

We can avoid this overhead by making the user aware of this inconsistency in the semantics and put the burden on the user to deal with this problem (like APRIL [10] does). However, this may confuse the user (who must already deal with the change of the sequential C semantics). In addition, the consistency between remote and local execution is useful when the user is sequentially debugging a program, since it will

```
...
divisionAsset(50) {
    _ENT_SendCall_divisionAsset(24);
    _ENT_SendCall_divisionAsset(26);
    _ENT_Wait(24) {
        divisionAsset(26) {
            _ENT_SendCall_divisionAsset(12);
            _ENT_SendCall_divisionAsset(14);
            _ENT_Wait(12) {
                divisionAsset(24) {
                    _ENT_SendCall_divisionAsset(11);
                    _ENT_SendCall_divisionAsset(13);
                    _ENT_Wait(11) {
                        divisionAsset(11) {
                            divisionAsset(4) {
                                straightforward_solution(4);
                            }
                            divisionAsset(7) {
                                straightforward_solution(7);
                            }
                        }
                        // 11 is resolved
                    }
                    _ENT_Wait(13) {
                        divisionAsset(12) {
                            divisionAsset(4) {
                                straightforward_solution(4);
                            }
                            divisionAsset(8) {
                                straightforward_solution(8);
                            }
                        }
                        // 12 is resolved
...
```

Figure 3.7: Calling sequence for worker W_1.

have the same behavior, independently of where a subproblem will be executed.

3.4.4 Deadlock Freedom

The presented scheduling algorithm would be useless if it could lead to deadlock. In this section, it is proven that deadlock is impossible in the algorithm. The proof is similar to the one given in [36]. Note that this proof does not imply that the implementation of the algorithm is deadlock-free, which is something that cannot be proved so easily.

In our system, when a worker waits for the solution of a subproblem, it selects from its local queue a subproblem for execution. If the queue is empty, then the worker is blocked and waiting. Hence, deadlock can happen only when all the queues are empty and all the workers are blocked waiting for the solutions of problems executed by the same or other workers. The necessary condition for deadlock is the existence of a circular chain of two or more workers, where each worker is waiting for the solution of a subproblem from the next worker in the chain. An example of such a chain of P workers is shown in Figure 3.8, where X_{sub} is the subproblem that problem X is waiting for, and the arrow is directed from the caller X to the callee X_{sub}.

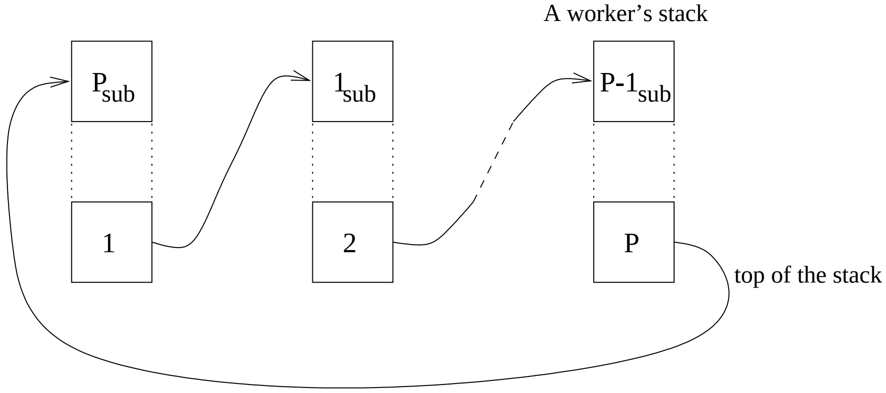


Figure 3.8: Circular chain of P workers necessary for deadlock.

The proof is done by contradiction. Let $A \prec B$ denote that invocation A has started execution before invocation B. If the subproblem B is generated from invocation A, then $A \prec B$ holds (**proposition 1**). Also, if A and B are invocations in the same worker and A is deeper in the stack than B, then $A \prec B$ holds (**proposition 2**). When there are P workers (which do not have to be distinct) where invocation X waits for invocation X_{sub} to end (for $X = 1, 2, ..., P$), then from proposition 1 we have the following expressions:

$$\begin{aligned} 1 &\prec 1_{sub} \\ 2 &\prec 2_{sub} \\ &\cdots \\ P &\prec P_{sub} \end{aligned}$$

If we assume that there is a circular chain of these P workers as depicted in

Figure 3.8 we have the following expressions (from proposition 2):

$$\begin{aligned} 1_{sub} &\prec 2 \\ &\dots \\ P - 1_{sub} &\prec P \\ P_{sub} &\prec 1 \end{aligned}$$

Clearly, the $\prec$ relation is transitive. By interchangeably taking one expression from each of the above groups of expressions we have:

$$1 \prec 1_{sub} \prec 2 \prec 2_{sub} \prec \ldots \prec P - 1_{sub} \prec P \prec P_{sub} \prec 1$$

or $1 \prec 1$, which is a contradiction. We reached a contradiction because we assumed there exists a circular chain of workers. Therefore, the algorithm is deadlock-free.

3.5 Experimental Results

Experiments were conducted to evaluate the performance of our method. The environment used for the experiments is described in Appendix A.

We compare the performance of our system with APRIL version 1.0 (for details on APRIL see Section 3.3.2 on page 18) since, as far as we know, APRIL is the only publicly available system that produces C code and can execute parallel D&C applications on top of PVM.

The application used was the Mandelbrot set computation on $[-2, -2]$ to $[2, 2]$ using a 1000x1000 pixel window (each pixel is represented by a `char`). At each division step, the window is divided on the largest dimension into two windows. At the base case, the familiar sequential Mandelbrot set computation is used. With APRIL, the division stops when there are no more available processors. With Enterprise, the division stops when the window has dimensions 100x100. The computation tree of the Mandelbrot set is binary and complete. However, for each leaf node the amount of computation varies (that is, the computation tree is unbalanced), since it depends on the location of the window in the Mandelbrot set.

Figure 3.9 shows the speedup and Figure 3.10 shows the efficiency for the range of $2 - 15$ processors. For Enterprise, we also show results based on the maximum

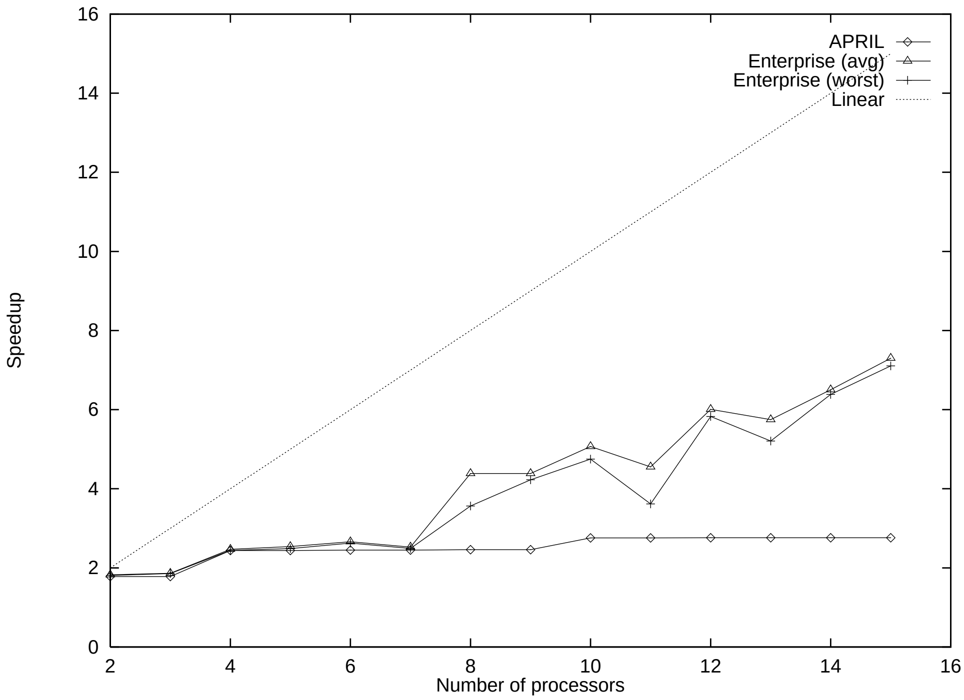


Figure 3.9: Speedup for the Mandelbrot set computation.

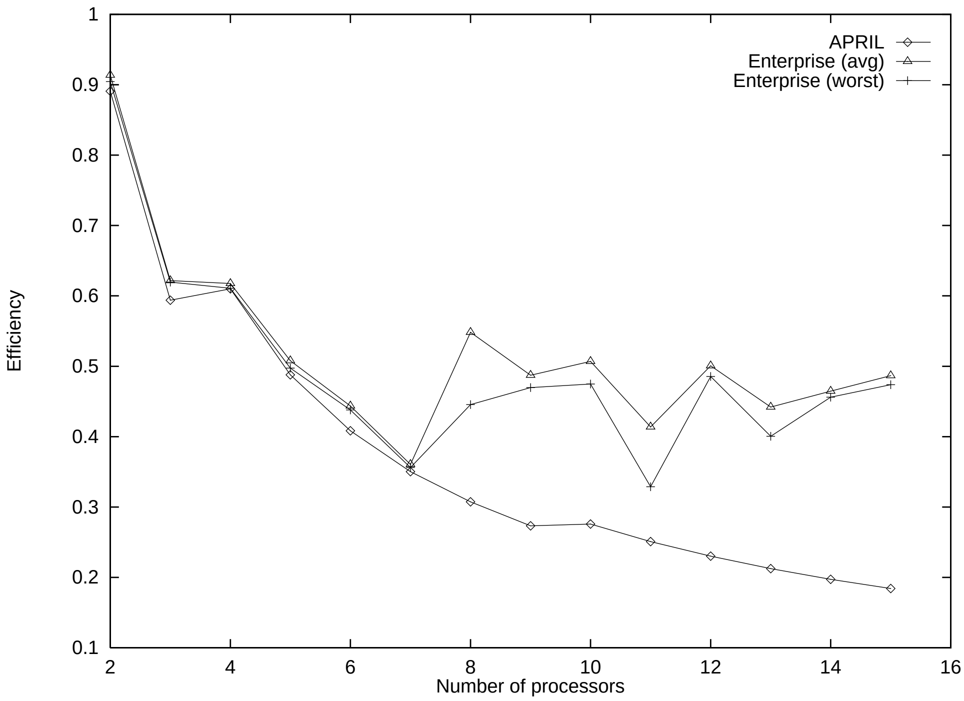


Figure 3.10: Efficiency for the Mandelbrot set computation.

execution times. Because our scheme is non-deterministic, the maximum time gives an idea about the worst performance of our algorithm. To increase the performance of APRIL, for some configurations of processors we had to put two processes on some of the leaf processors until the total number of processes reached the next power of two.

As expected, Enterprise results are better than APRIL's results. This happens because APRIL uses a binomial processor tree and an idle processor can receive work only from its parent in this tree. This means that even if there is work at a processor different from its parent, the system cannot give it work and thus the load becomes unbalanced. This situation does not arise when the computation tree is a complete balanced binary tree and the number of processors is a power of two. Here, where Mandelbrot's computation tree is unbalanced, the load becomes unbalanced and the performance suffers. Moreover, note that even when more processors are used for the Mandelbrot set computation, APRIL's performance stays unchanged, since APRIL cannot take advantage of the available processors. On the other hand, our scheme can utilize all the available processors, and thus, its performance increases.

The Enterprise results based on the average execution times do not follow a straight line as would be expected. This is explained by the non-determinism in the algorithm. Some orders of events are more "unlucky" than others and increase the average of the execution times.

We can also observe that for a small numbers of processors, where the load imbalance is small, our scheme performs at least as good as APRIL. This suggests that the scheduling overhead in our scheme is at an acceptable level.

Chapter 4

Dynamic Scheduling for Master-Worker Applications

4.1 Overview

This chapter addresses scheduling for parallel master-worker (M-W) applications. It describes the features most of the M-W applications have, and determines the goals that a scheduling algorithm for M-W applications has to achieve. It states the different scheduling schemes that exist. Then, it presents our scheduling algorithm and compares its performance with the performance of the previous Enterprise algorithm. Finally, the chapter concludes with the extensions needed to our scheme in order to support composite applications.

4.2 Task Model

The M-W model is the most commonly used model in parallel program design. In this model, the *master* generates tasks that are processed by a number of *workers*. Most of the time, the master creates and distributes all the tasks, and then waits for all the replies before it continues execution (barrier). Nevertheless, there are cases where the master alternates between the creation, waiting, and computation phases with no particular order. Usually, the M-W model is expressed using loops.

The computation cost for different invocations of the same asset depends on the size and/or the input values of the asset's arguments. To correlate the computation cost of an invocation with the input values of its arguments, we need semantic information about the asset that can only be provided by the compiler or the user. Hence, we cannot readily use the input values to estimate the computation cost.

On the other hand, we can easily calculate the size of an asset's arguments. Frequently, the message size distribution of the tasks (both of the request and the reply messages) is related to the computation cost distribution of the tasks. We expect that when the message size distribution is increasing (decreasing), the computation cost distribution will likely be either uniform or increasing (decreasing). When the message size is uniform or irregular, the computation cost will likely be either uniform or irregular.

```
parallel for (i = 1; i ≤ N * N; i++) {
    AdjointConvolution(i);
}
```

(a) Master.

```
void AdjointConvolution(int i) {
    int k;

    for (k = i; k ≤ N * N; k++) {
        A(i) = A(i) + B(k) * C(k-i);
    }
}
```

(b) Worker.

Figure 4.1: The Adjoint-Convolution application.

We can observe that in some M-W applications, there exists a correlation between the creation order of tasks and their computation cost. For example, in adjoint-convolution (Fig. 4.1) as the index `i` increases, the computation cost of the function `AdjointConvolution()` decreases. So, if we consider the computation cost of the tasks as a function of their creation order, then the computation cost can have one of the following four distributions: uniform (as in matrix multiplication), increasing (as in reverse adjoint convolution), decreasing (as in adjoint convolution), or irregular (as in Mandelbrot set computation). The last distribution is encountered when there is no correlation between the creation order of tasks and their computation cost.

M-W applications may consist of *work cycles*. Each cycle includes the creation of a group of tasks, and a barrier for these tasks which separates the successive cycles. One such application is the Jacobi algorithm where each pass over the entire matrix constitutes a cycle. Generally, most of the applications that use iterative methods consist of cycles. Since the characteristics of the tasks are often approximately the same between cycles, we can exploit the information from previous cycles to schedule the next ones.

Most of the related work [23, 29, 35, 30, 45] deals with loop scheduling, and it is assumed that the number of created tasks is known just before the loop is entered. But this implies that either the programmer has to supply this number, or the compiler has to deduce it from the program. In the latter case, the loop should have a simple form, such as:

```
for (i = limitL; i < limitU; i += step)
```

If the loop body contains a selection statement where at least one of its branches does not generate a task, then the user or the compiler will overestimate the number of created tasks. This can lead to a poorly balanced distribution of tasks to the workers.

Hence, it is restrictive to assume that the total number of generated tasks is known beforehand. One way to find the number of created tasks is to execute the entire loop without distributing any of the tasks. After the loop execution, this number will be known. Obviously, the existing concurrency between creation of succeeding tasks and processing of early created tasks will be lost. If the loop body only consists of the asset calls or the number of iterations is small, then the loop execution time and, thus the lost parallelism will be small. However, it is possible that other statements will exist inside the loop body. Most of the time, these statements are the preprocessing needed before each task's creation. Here too, an intelligent compiler could detect whether the loop body contains other statements.

Furthermore, there are applications where it is not possible to know the number of tasks in advance. For example, iterative algorithms that subdivide a problem until a prescribed error threshold is met. Or more generally, the applications where the master administers a task queue, and the workers remove from and store tasks in it.

Here, we will not assume any knowledge about the number of generated tasks, or the form of the loops.

4.3 Problem Description

A scheduling scheme has to achieve two fundamental objectives: minimize the scheduling overhead, and share the load between workers. These two goals conflict with each other. At one extreme, the tasks can be assigned to processors at compile time (*static scheduling*). In this scheme, there is no overhead, but the load imbalance can be large when the tasks have non-uniform execution times. At the other extreme lies *self scheduling* [34] where each idle worker obtains a task from a central task queue.[1] In self-scheduling, the difference in finishing times between workers will be at most one task, thus the load will be perfectly balanced. However, the overhead can be high, because continuous access to the central queue can cause a bottleneck, and the workers may waste too much idle time between the processing of successive tasks.

Dynamic scheduling can be divided into two phases. The first phase deals with the initial distribution of tasks from the master to the workers. The second phase deals with the redistribution of tasks from heavily loaded to lightly loaded workers. The more balanced the initial work distribution, the fewer tasks need to be transferred between workers in the second phase. It is better to redistribute as few tasks as possible, since each transfer wastes processor cycles (idle time at the receiver) and network resources (increased contention and wasted bandwidth). Still, in a non-dedicated computing environment with load fluctuations, task transfers are unavoidable.

In the first phase, the scheduling scheme has to decide on the number of chunks that the available work will be decomposed into. If the tasks have non-uniform computation cost, then the scheme must also decide which tasks will be included in each chunk. Furthermore, if the workers have different processor speeds, the scheme must decide the destination worker for each chunk. All these decisions are strongly interrelated.

In the second phase, when a worker becomes idle, it should receive tasks from a busy worker. Here, the scheduling scheme has to decide from which busy worker to request tasks and the number of tasks to be transferred.

To accomplish its objectives, the scheduling algorithm has to achieve the following:

[1] Enterprise used this scheme [27].

- *Minimize the communication overhead.* If a chunk of tasks is sent in one message to a worker, communication latency is minimized, because it is only paid by one task in the message. On the other hand, it is better to send few tasks to a worker, so it will be less probable that some of these tasks will be transferred later to other workers.

- *Minimize idle time at the workers.* This can be achieved by sending more than one task to a worker, either many tasks in a chunk or one task at a time before the worker finishes processing its current task. Consequently, the worker can immediately start processing its next task.

- *Avoid bottlenecks at the master.* The bottleneck can be minimized by giving many tasks to each worker, so workers will request work less often.

- *Balance the workload.* At the beginning of the computation there is no need to be concerned with balancing the load. Therefore, many large tasks can be given to the workers. But at the end of the computation, where we do not want to have many idle workers waiting for a few other workers to finish, the tasks obtained by a worker should be few and small.

From these considerations, we see that we have to adopt a middle course to the number of tasks sent to the workers. This is difficult to achieve if we also consider that all the tasks may not have the same cost, and that all the processors do not consume tasks at the same or even a constant rate.

4.4 Related Work

The schemes presented in this section deal with the dynamic scheduling of *parallel loops* (DOALL loops). That is, loops without any cross-iteration dependences. Thus, the term iteration is used here to refer to tasks. A common assumption is that the number of iterations is known just before the execution of the loop.

The different architecture characteristics and goals between UMA (Uniform Memory Access) multiprocessors and non-dedicated NOWs result in different scheduling algorithms. Here we divide the scheduling schemes into two categories based on the architecture model they were proposed for.

4.4.1 Dynamic Loop Scheduling on a Multiprocessor

The following approaches were proposed for UMA multiprocessors. The processors are homogeneous and it is assumed that there is no external load. The unscheduled tasks are stored in a central queue, and when a processor becomes idle, it obtains a chunk of tasks from this queue.

In multiprocessors, the primary overhead arises from the mutually exclusive accesses to the central queue. To decrease the number of accesses to the queue, and thereby minimize the overhead, the number of iterations assigned initially to a processor must be large. At the same time, this number should not be too large, otherwise a load imbalance may arise. In the following, N denotes the number of tasks and p the number of workers.

In *uniform-sized chunking* [23], each idle processor obtains from the queue a chunk of K tasks, where K stays constant during the entire computation. Kruskal and Weiss assume that the computation cost of the task is an independent random variable. However, this is true only for a small portion of the parallel applications expressed with loops. Their scheme can lead to load imbalance when the computation cost distribution is increasing or decreasing; the first or the last processor respectively, will obtain the largest portion of the total work. Moreover, if a processor obtains the last K iterations when the other processors finish their part of the work, these processors will stay idle for K iterations. Of course if N is huge, the probability for load imbalance becomes smaller.

In *guided self-scheduling* [29], Polychronopoulos and Kuck consider loops where the computation cost is either uniform or can take one of many possible values with equal probability. The processors can join the computation at arbitrary times. In this scheme, a processor removes x_i of the remaining R_i iterations from the queue (where $R_1 = N$, $x_i = \lceil \frac{R_i}{p} \rceil$ and $R_{i+1} = R_i - x_i$). Hence, the chunk of iterations obtained by a processor is always decreasing. All the processors finish executing the loop within one iteration difference from each other. The major weakness of this scheme is that if the computation cost distribution is decreasing, the first processor will obtain more than $\frac{1}{p}$-th of the total work, thus causing load imbalance.

Trapezoid self-scheduling [35] is similar to guided self-scheduling with the exception that the number of iterations obtained by a processor is a linear decreasing function that has a simple implementation,[2] thus decreasing the scheduling overhead. Furthermore, with this function, the first processor cannot obtain more than $\frac{1}{p}$-th of the total work, so this scheme does not have the load imbalance problem encountered in guided self-scheduling. From Tzen and Ni's experiments with uniform, irregular, linearly increasing and decreasing computation cost distributions, they conclude that trapezoid self-scheduling achieves better results than guided self-scheduling [35].

4.4.2 Dynamic Loop Scheduling on a NOW

To our knowledge, few researchers have worked on dynamic loop scheduling for heterogeneous non-dedicated NOWs.

In [45], four different dynamic load balancing strategies for uniform loops are compared. The authors show that different scheduling schemes are needed for different parameters of the same application (different number of processors, task sizes, etc). The network is considered homogeneous. Each strategy uses one of the four possible combinations along the two dimensions: centralized *vs* distributed, and global *vs* local. The general form of the four strategies is as follows. Initially all the iterations are distributed equally among the processors. The first processor to finish interrupts the other processors. Then they exchange information about their load and number of remaining iterations, and a new distribution of the unfinished iterations is calculated. If there is an improvement in the execution time, the iterations are redistributed between the processors and only then, the processors continue with their work.

In this scheme, the synchronization phase (the period when load balancing occurs) is expensive, since during this time the processors are not allowed to execute any work. On the other hand, if the processors continued to compute, the load information would be less accurate and load balancing would be more difficult. At the end of the computation (when many processors become idle), many synchronization phases will occur, resulting in increased overhead. Furthermore, the initial distribution of the iterations does not take into account the different processor speeds, possibly leading to more redistributions.

[2]Linearity makes the function simple enough to be implemented with a single atomic instruction instead of a critical section (like in guided self-scheduling).

A novel approach to load sharing loops is proposed in [20]. In this algorithm, the processors are virtually connected forming a static *task migration network* (currently a ring) based on their speeds. When a processor becomes idle, it requests tasks from one of the neighboring processors in the migration network. This algorithm is based on the observation that the sender and the receiver of the migrated tasks can be easily identified. For example, the most likely sender is the slowest processor and the most likely receiver is the fastest processor. Since no exchange of load information is needed to choose the sender of the tasks, the load sharing overhead is minimized.

The disadvantage of this algorithm originates from the same source as its advantage - the static structure of the migration network. When the load fluctuates, or many processors have the same speed, the sender can be one of many processors and the same thing holds for the receiver. So, some chances for migration will be lost or take too much time to be performed.

Pruyne and Livny [30] study parallel processing in an environment where the processors available to an application change during its execution. They developed CARMI (Condor Application Resource Management Interface), which provides services for writing parallel programs on such an environment using Condor [24]. On top of CARMI, they built WoDi (Work Distributor), a framework for M-W applications. Among other things, WoDi is responsible for allocating the workers and scheduling the tasks provided by the master. If the program consists of work cycles, the user has to specify where each cycle begins and ends, as well as the number of tasks per cycle. WoDi maintains a history of the computation cost of all the tasks within a cycle, and at the next cycle uses this history to distribute the largest tasks first and to the fastest processors.

4.5 Proposed Scheduling Algorithm

In this section, we propose a new dynamic scheduling scheme for M-W applications. Our scheme does not require any *a priori* knowledge about the number of tasks or their computation cost. We only assume that the master needs the task replies "close" to one another (like when there exists a barrier). This is not restrictive, and allows the workers to process the tasks in any order. Otherwise, since usually the master waits for the replies in the same order as the tasks are created, the creation order had to be taken into account when tasks are scheduled.

The master and each worker maintains a queue of unprocessed tasks. A worker sends a reply to the master even when there is no return value. This is needed to determine whether a worker is idle. The reply also contains the computation time of the processed task.

Below, the *normalized computation time* of a task refers to the time needed to execute this task on the base processor (that is, the slowest processor). The *normalized estimated computation time function* is a linear function used to estimate the normalized computation time of a task from its heuristic value. The notion of heuristic values has been introduced in Section 3.4.2 on page 25. The symbols used in this section are defined as follows:

- p: the number of workers.
- W_j: the worker j.
- $nSpeed(W_j)$: the normalized speed of worker W_j.
- $w(W_j)$: the fraction of the processing power of worker W_j relative to all the workers. $w(W_j) = \frac{nSpeed(W_j)}{\sum_{m=1}^{p} nSpeed(W_m)}$.
- f: the maximum number of unprocessed tasks that a worker can have at the beginning of the scheduling algorithm.
- d: the minimum number of unprocessed tasks that a worker can have before it is considered idle.
- r: the minimum number of computation time samples needed to calculate the coefficients of the normalized estimated computation time function.
- $nEstimCost(t)$: the estimation of the normalized computation time for task t.
- $EstimCost(W_j, t)$: the estimation of the computation time for task t if it will be executed on worker W_j. $EstimCost(W_j, t) = \frac{nEstimCost(t)}{nSpeed(W_j)}$.
- $Cost(W_j, t)$: the computation time of task t executed on worker W_j.

- $nCost(t)$: the *normalized computation time* of task t. If t has been executed on worker W_j, then $nCost(t) = Cost(W_j, t) * nSpeed(W_j)$.
- $heur(t)$: the heuristic value of task t.
- λ: the slope of the normalized estimated computation time function.
- H: the heuristic value for a fixed point of the normalized estimated computation time function.
- NC: the normalized computation time for a fixed point of the normalized estimated computation time function.

f, d and r are constants (their value is determined at the end of this section), $nSpeed(W_j)$ is benchmarked off-line for each worker, and p and $w(W_j)$ are calculated each time the number of workers changes. $nCost(t)$ and $heur(t)$ are measured by the worker and the master, respectively. The coefficients λ, H and NC are calculated at run-time by the master.

The algorithm consists of two phases. In the first phase, the master distributes tasks to the workers. In the second phase, the master redistributes tasks from busy to idle workers. First, we present how the computation time of unprocessed tasks is estimated from their heuristic values. Then, the components of the algorithm are described. Pseudo-code for the algorithm is shown in Figures 4.2 and 4.3 (for the master), and Figure 4.4 on page 50 (for the worker).

4.5.1 Heuristic Values and Estimated Computation Cost of Tasks

In D&C applications, a heuristic value is assigned to each task using the value from one of three measures: the user-supplied estimation function, the size of the parameters of the asset call, or the height of the task in the computation tree. Obviously, in M-W applications, we cannot use the height as a measure. Instead, we use the creation order of the task. When a task is created, it is assigned a consecutive number starting from one. The creation order is used as a measure because, as we have stated in the section discussing the task model, usually the computation cost of tasks is correlated with their creation order.

```
localQueue  // local task queue

switch (event) {
    case task t created :
        if ((less than r replies have been received)
                and (there is a worker with less than f tasks)) then {
            // this is the beginning of the computation, so send the task immediately to a worker
            W_j = a worker with less than f tasks;
            send t (tag = TASKS) to W_j;
            update the number of tasks and the sum of heuristic values that W_j has;
        } else {
            // the workers have work to execute, so no need to send the task immediately
            store t in localQueue;
        }

    case message received :
        ProcessMessage();

    case idle worker exists :
        W_idle = fastest idle worker;
        if (localQueue is not empty) then {
            // send a portion from the local work to the worker
            nEstimCost_total = normalized estimated computation time for the tasks
                in the queues of all the workers and the master (use formula 4.1);
            workAmount = w(W_idle) * nEstimCost_total;
            remove highest cost task t from localQueue until ∑ nEstimCost(t) ≥ workAmount;
            send tasks (tag = TASKS) to W_idle;
            update the number of tasks and the sum of heuristic values that W_idle has;
        } else if (there are busy workers) then {
            // transfer work from a busy worker
            W_busiest = busy worker that is estimated to finish its work last (use formula 4.2);
            workAmount = w(W_idle) * ∑_{m=1}^{p} nTime_fin(W_m) (from formula 4.3);
            if (condition 4.4 is satisfied) then {
                // there is improvement in the execution time
                send workAmount, W_idle (tag = TASKS_REQUEST) to W_busiest;
            }
        } else {
            // Do nothing
        }
}
```

Figure 4.2: Pseudo-code for the master of a M-W application.

In M-W applications, the master - and not a worker - decides which measure will be used at run-time and propagates this information to the workers. When the user does not supply an estimation function, the master compares the size of the parameters of the first two asset calls. If they are equal, the creation order is used as

```
function ProcessMessage() {
    msg = blocking receive;
    W_j = worker sending the message;
    switch (msg.Tag) {
        case REPLY :
            // reply received from a worker
            store the reply;
            update the number of tasks and the sum of heuristic values that W_j has;
            if (reply is among the 10 most recent ones) then {
                recalculate the coefficients λ, H and NC;
            }
            if (W_j has less than d tasks) then {
                mark W_j as idle;
            }

        case REQ_FAILED :
            // the request for work transfer has failed
            mark W_idle as idle again;

        case REQ_SUCCEEDED :
            // the request for work transfer has succeeded
            update the number of tasks and the sum of heuristic values that W_j and W_idle have;
    }
}
```

Figure 4.3: Pseudo-code for the master of a M-W application (*continued*).

a measure, otherwise the size is used.

The computation cost of a task is expressed by its computation time. We assume that the heuristic value of a task and its normalized computation time are linearly related. For this reason, we define a linear function, the *normalized estimated computation time function*, to estimate the normalized computation time of unprocessed tasks from their heuristic values. Because of the linear relation, the normalized estimated computation time of a task t with heuristic value $heur(t)$ is given by the usual equation for lines:

$$nEstimCost(t) = \lambda * (heur(t) - H) + NC.$$

If we have β tasks $(t_1, t_2, \ldots, t_\beta)$, then by using the above formula for each task and adding all the expressions, the normalized estimated computation time of these β tasks ($nEstimCost_{total}$) is given as:

$$nEstimCost_{total} = \lambda * (\sum_{k=1}^{\beta} heur(t_k) - \beta * H) + \beta * NC. \tag{4.1}$$

```
localQueue  // local task queue

set an alarm that calls periodically MsgPolling();
for ever {
    wait blocked for a message containing tasks;
    while (localQueue is not empty) {
        t = remove highest cost task from localQueue;
        process t, and measure Cost(W_j, t);
        nCost(t) = Cost(W_j, t) * nSpeed(W_j);
        send reply, nCost(t) (tag = REPLY) to master;
    }
}


function MsgPolling() {
    msg = non-blocking receive;
    switch (msg.Tag) {
        case TASKS :
            // tasks received from the master or another worker
            store tasks in localQueue;

        case TASKS_REQUEST :
            // request from the master to send work to another worker
            get from msg the max amount of work to be sent, the receiver W_idle,
                and the coefficients λ, H and NC;
            remove highest cost task t from localQueue until ∑ nEstimCost(t)
                becomes larger than expression 4.6;
            if (no tasks have been removed) then {
                // notify the master that no tasks were sent
                send (tag = REQ_FAILED) to master;
            } else {
                // tasks are sent to the idle worker
                send tasks (tag = TASKS) to W_idle;
                // notify the master that tasks were sent
                send number of tasks sent and the sum
                    of their heuristic value (tag = REQ_SUCCEEDED) to master;
            }
    }
}
```

Figure 4.4: Pseudo-code for the worker of a M-W application.

In the above formulas, λ is the slope and (H, NC) is a point of the linear function. The values for these coefficients are calculated by the master using the computation times from previously executed tasks. When a worker W_j executes a task t, it measures t's computation time ($Cost(W_j, t)$) and from that, it calculates t's normalized computation time ($nCost(t)$). This value is sent to the master together with t's reply.

From the timings the master receives, it has to choose which ones will be used for the calculation of the coefficients. Because we expect that the near past will predict the future better than the distant past, the most recent timings should be used. The timings can be ordered in one of two ways: either by their arrival order, or by the creation order of the corresponding tasks. We use the creation order of the tasks to determine which timings are more recent. Otherwise, if we used the arrival order and the recently created tasks had smaller computation cost than the previously created tasks, then the timings of the previously created tasks could arrive after the timings of the recently created ones. Consequently, the calculation of the coefficients would be influenced from distant past results.

The ten[3] most recent timings are used for the calculation of the coefficients. H and NC are taken as the average of the heuristic values and the normalized computation times, respectively. λ is calculated by the formula giving the slope of a line from two of its points. One of these points is the average over the three most recent timings, and the other is the average over the three oldest timings (among the ten timings). So, if $heur(t_1), heur(t_2), \ldots, heur(t_{10})$ are the heuristic values of the ten most recently created tasks $t_1, t_2, \ldots, t_{10}$, and $nCost(t_1), nCost(t_2), \ldots, nCost(t_{10})$ are the respective normalized computation times, the coefficients are given by the formulas below:

$$H = \frac{heur(t_1) + heur(t_2) + \ldots + heur(t_{10})}{10},$$

$$NC = \frac{nCost(t_1) + nCost(t_2) + \ldots + nCost(t_{10})}{10},$$

$$h_{point_1} = \frac{heur(t_1) + heur(t_2) + heur(t_3)}{3}, h_{point_2} = \frac{heur(t_8) + heur(t_9) + heur(t_{10})}{3},$$

$$nCost_{point_1} = \frac{nCost(t_1) + nCost(t_2) + nCost(t_3)}{3},$$

$$nCost_{point_2} = \frac{nCost(t_8) + nCost(t_9) + nCost(t_{10})}{3},$$

$$\lambda = \frac{nCost_{point_2} - nCost_{point_1}}{h_{point_2} - h_{point_1}}.$$

Distinct points were not used for λ's calculation, because initial experiments with distinct points showed that λ's value fluctuates a lot.

[3] This value has been selected arbitrarily.

When more recent timings arrive at the master, the coefficients are recalculated. In this way, if the relation between the heuristic value and the computation time is not linear or the timings are inaccurate, the recalculation continuously adjusts the value of the coefficients.

4.5.2 Components of the Scheduling Algorithm

The master stores the number of the tasks residing in each worker, as well as the sum of their heuristic values. This information is updated each time a chunk of tasks is sent or a reply is received.

Initial Distribution of Tasks

At the beginning of a M-W computation, the master does not have any timings. So when a task is created, the master sends it immediately to a worker (tag = `TASKS`). That way, the first replies will be received as soon as possible, and no concurrency is lost between the master and the workers. The tasks are sent to the workers in a round-robin order, starting from the fastest worker. This procedure continues until either each worker has f tasks, or the master has received the first r replies. Any new tasks created after each worker has f tasks and before the first r replies have arrived, are stored in the master's queue.

After the first r replies have arrived, the master uses them to calculate the coefficients of the normalized estimated computation time function. From there on, it uses this function to estimate the computation cost of the unprocessed tasks and distributes the tasks based on these estimations.

When the master receives a reply from worker W_j (tag = `REPLY`), it checks whether W_j will become idle. The master considers a worker to be idle when the worker has less than d tasks in its queue. If W_j becomes idle and the master's queue is not empty, the master gives to W_j a chunk of tasks relative to W_i's speed. It estimates the total work in its queue and the queues of all the workers (using formula 4.1), and gives $w(W_j)$-th of this estimate to W_j.

The master selects the tasks from its queue starting from the largest tasks until the amount of the selected work is larger than or equal to the amount of work that it has decided to send to the worker.

Redistribution of Tasks

In our model, tasks can be created at any time. Therefore, it is possible that the master may have to redistribute tasks from one worker to another, and afterwards if new tasks have been generated to distribute them.

In this subsection, the following symbols will be used:

- $nTime_{fin}(W_j)$: the estimation of the normalized finish time for worker W_j.
- $Time_{fin}(W_j)$: the estimation of the finish time for worker W_j.
- $nTime_{NEWfin}(W_j)$: the estimation of the normalized finish time for worker W_j after the redistribution of tasks.
- $Time_{NEWfin}(W_j)$: the estimation of the finish time for worker W_j after the redistribution of tasks.

If the master considers a worker W_{idle} to be idle and no tasks exist in the master's queue, it will have to transfer tasks from another worker to W_{idle}. So, it goes through the list of busy workers that are not involved in another task transfer and have more than one task,[4] and estimates their finish time. If a worker W_j has k tasks where the sum of their heuristic value is hv, then its normalized estimated finish time is given by:

$$nTime_{fin}(W_j) = \lambda * (hv - k * H) + k * NC,$$

and its estimated finish time is given by:

$$Time_{fin}(W_j) = \frac{nTime_{fin}(W_j)}{nSpeed(W_j)}. \tag{4.2}$$

The master selects the busy worker $W_{busiest}$ with the maximum estimated finish time to be the worker that will transfer tasks from its queue to W_{idle}.

The amount of work that W_{idle} would receive should not increase the load imbalance between the workers. This means that W_{idle}'s finish time after the redistribution

[4]If a worker has only one task, this task has started execution on that worker and cannot be moved.

should not be larger than the finish time of a perfectly balanced distribution. That is:

$$
\begin{aligned}
Time_{NEWfin}(W_{idle}) &\leq \frac{\sum_{m=1}^{p} nTime_{fin}(W_m)}{\sum_{m=1}^{p} nSpeed(W_m)} \Rightarrow \\
nTime_{NEWfin}(W_{idle}) &\leq w(W_{idle}) * \sum_{m=1}^{p} nTime_{fin}(W_m). \qquad (4.3)
\end{aligned}
$$

So, W_{idle} should receive from $W_{busiest}$ at most the $w(W_{idle})$-th of the total work residing in all the workers. If the transfer takes place at the end of the computation, where the few busy workers have little work left, then it is possible that the amount of work to be transferred will be small. As a result, the finish time may increase due to the communication cost of moving the tasks. To avoid that, tasks are transferred only if there is at least a 10% improvement in $W_{busiest}$'s finish time. This condition is expressed by:

$$
\begin{aligned}
Time_{NEWfin}(W_{busiest}) &\leq 0.90 * Time_{fin}(W_{busiest}) \Rightarrow \\
\frac{nTime_{fin}(W_{busiest}) - nTime_{NEWfin}(W_{idle})}{nSpeed(W_{busiest})} &\leq 0.90 * \frac{nTime_{fin}(W_{busiest})}{nSpeed(W_{busiest})} \Rightarrow \\
0.10 * nTime_{fin}(W_{busiest}) &\leq nTime_{NEWfin}(W_{idle}) \overset{\text{(Eqn 4.3)}}{\Longrightarrow} \\
0.10 * nTime_{fin}(W_{busiest}) &\leq w(W_{idle}) * \sum_{m=1}^{p} nTime_{fin}(W_m). \qquad (4.4)
\end{aligned}
$$

If the above condition is satisfied, the master requests from $W_{busiest}$ to transfer at most a portion of its work to W_{idle} that satisfies condition 4.3 (tag = `TASKS_REQUEST`). Since the coefficients of the computation time function are needed, they are sent to $W_{busiest}$ along with the request. If $W_{busiest}$'s queue is empty, $W_{busiest}$ notifies the master that it cannot carry out the request (tag = `REQ_FAILED`). In this case, the master continues the above procedure until it has requested work from all the busy workers.

Suppose that $W_{busiest}$ has work to transfer. Then after the redistribution, W_{idle}'s finish time should not be larger than $W_{busiest}$'s finish time. Or equivalently:

$$
Time_{NEWfin}(W_{idle}) \leq Time_{NEWfin}(W_{busiest}) \Rightarrow
$$

$$\frac{nTime_{NEWfin}(W_{idle})}{nSpeed(W_{idle})} \leq \frac{nTime_{fin}(W_{busiest}) - nTime_{NEWfin}(W_{idle})}{nSpeed(W_{busiest})} \Rightarrow$$

$$nTime_{NEWfin}(W_{idle}) \leq nTime_{fin}(W_{busiest}) * \frac{nSpeed(W_{idle})}{nSpeed(W_{busiest}) + nSpeed(W_{idle})} \quad (4.5)$$

Finally, $W_{busiest}$ sends to W_{idle} a portion of its work that satisfies conditions 4.3 and 4.5, that is:

$$\min \left\{ \begin{array}{c} w(W_{idle}) * \sum_{m=1}^{p} nTime_{fin}(W_m) \\ \\ nTime_{fin}(W_{busiest}) * \frac{nSpeed(W_{idle})}{nSpeed(W_{busiest})+nSpeed(W_{idle})} \end{array} \right\} \quad (4.6)$$

After the tasks are sent to W_{idle} (tag = `TASKS`), a notification is sent to the master (tag = `REQ_SUCCEEDED`), containing the number of tasks transferred and the sum of their heuristic value.

Note that $W_{busiest}$ makes the final decision about the amount of work to be transferred to W_{idle}. This is done because $W_{busiest}$ knows the exact amount of work in its queue, while the master knows the amount of work that $W_{busiest}$ had at the time it sent the last reply to the master. Hence, the master may overestimate the amount of work, and request $W_{busiest}$ to transfer more tasks.

Determination of the Value of the Constants

The specific value for f is not so important, since this constant is used only at the beginning of the computation. Currently, f is equal to three. If f is equal to one, then a worker will stay idle after it processes a task. On the other hand, if f is large, it is possible that a slow worker will receive more work than it should. This may lead to unnecessary redistributions later on.

The value of d should be such that the worker can start processing a new task right after it finishes processing its current task, so there will be no idle time at the worker. Ideally, the value of d should not be fixed. It should depend on the computation time of the remaining tasks at the worker and the communication time to transfer tasks to the worker. Since it is difficult to calculate d at run-time, we set its value equal to two. The underlying reason is that from these two tasks, one is the task that the worker is currently processing, and the other is the task that the worker will be processing while the master attempts to send more work to this worker.

If d was equal to one, then the scheduling scheme degenerates to a form of self-scheduling. Between processing phases, the worker will have to stay idle for as long as it takes the reply to reach the master and a new chunk of tasks is received. Alternatively, if d was larger than two, then it would be more likely that a worker will have a new task in its queue after it finishes processing its current task. However, the master would consider a worker idle even if this worker had tasks to execute. This could lead to load imbalance.

We chose r to be equal to four. If r is small, the value of the coefficients λ, H and NC may be inaccurate, since it will depend on few samples. On the other hand, if r is large, then the calculation of the coefficients will delay to occur.

Task Selection from a Worker's Queue

Each worker always selects the largest task for local processing and for transfer to another worker. Otherwise, a worker may end up executing the largest task when all the other workers are finishing their work, thus resulting in a load imbalance.

Still, there are cases where this is not the best strategy. Suppose that the slowest worker obtains a chunk of tasks where one task is huge. If we know that a faster worker is going to request some tasks from this slow worker, then it is better to keep this task for the faster worker, and execute the smaller ones. Obviously, it is difficult to have such knowledge.

4.6 Experimental Results

Experiments were done to evaluate the performance of our scheduling algorithm. We compare our algorithm with self-scheduling (SS), which is the algorithm used previously in Enterprise. The environment used for the experiments is described in Appendix A.

The application used was the multiplication of two matrices ($\mathbf{C} = \mathbf{A} \times \mathbf{B}$). $\mathbf{A}$ has dimensions $(RowsA, ColsA)$, and $\mathbf{B}$ dimensions $(ColsA, ColsB)$. Each matrix element is a `double` (8 bytes). In the matrix multiplication algorithm, the master distributes the entire matrix $\mathbf{B}$ to all the workers. Then, it decomposes matrix $\mathbf{A}$ into n pieces where each piece consists of $RowsA/n$ rows. Each piece is sent to a worker

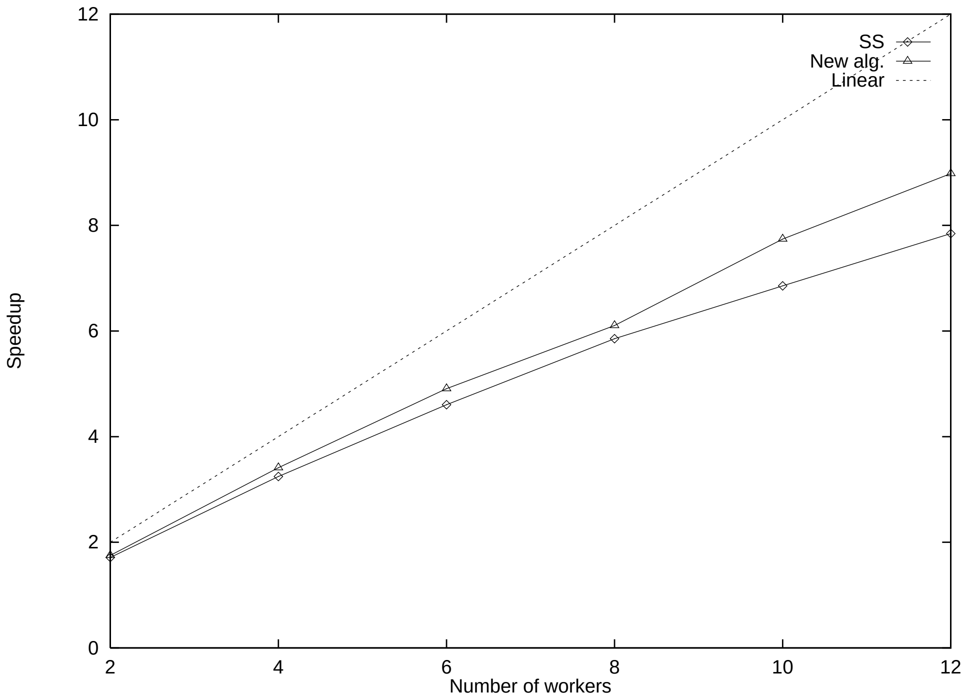


Figure 4.5: Speedup for the Matrix Multiplication ($RowsA = 900$).

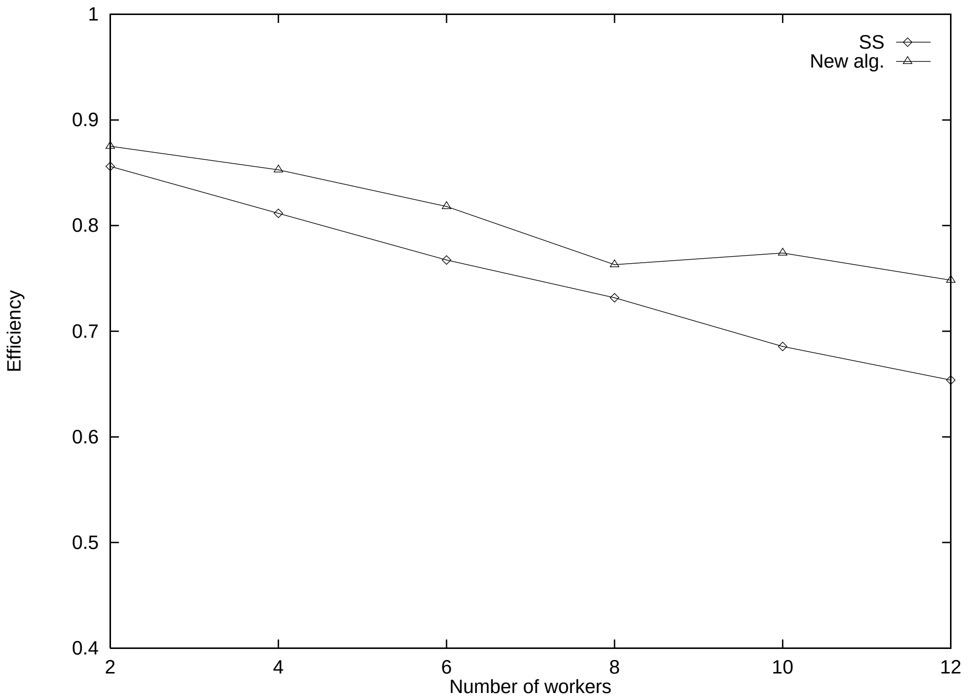


Figure 4.6: Efficiency for the Matrix Multiplication ($RowsA = 900$).

where the corresponding $(RowsA/n, ColsA)$ sub-matrix **C** is computed and returned to the master.

For our experiments, $ColsA = ColsB = 600$, $n = 300$, and $RowsA$ was 900 and 1200. In the measurements, we excluded the time needed to distribute matrix **B** (because the communication cost of transferring **B** was high and fluctuated a lot, perturbing the execution times). Note that in the diagrams, the number of workers and not the number of processors is shown. Each worker was placed on a different processor, but one more processor was used to place the master on.

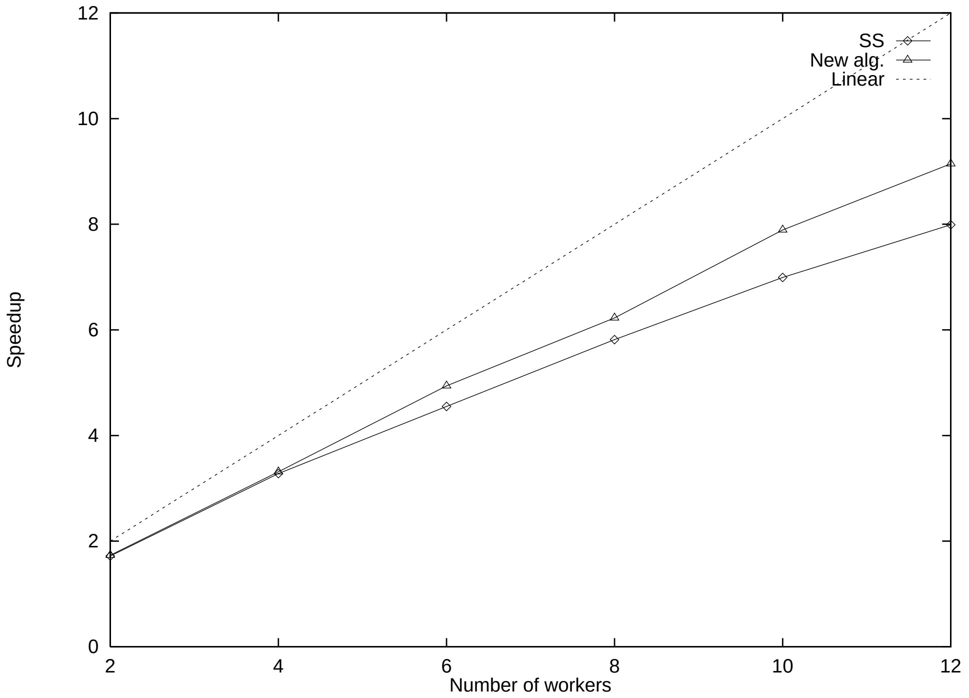


Figure 4.7: Speedup for the Matrix Multiplication ($RowsA = 1200$).

The speedup and the efficiency of both scheduling schemes is shown in Fig. 4.5 and 4.6 for $RowsA = 900$, and in Fig. 4.7 and 4.8 for $RowsA = 1200$.

When the number of workers is small (two or four), the performance of both algorithms is almost the same. As the number of workers increases, our algorithm performs better than SS. This occurs because in SS the master becomes a bottleneck, since it has to manage more workers. However, we observe that the efficiency of our algorithm decreases too as the number of workers increases, but at a slower rate. Hence, our algorithm does not alleviate entirely the bottleneck of the master. Of course, we expect that in our scheme the master will eventually become a bottleneck, but that this will happen with a larger number of workers.

The performance of our algorithm may not look impressive. The execution time

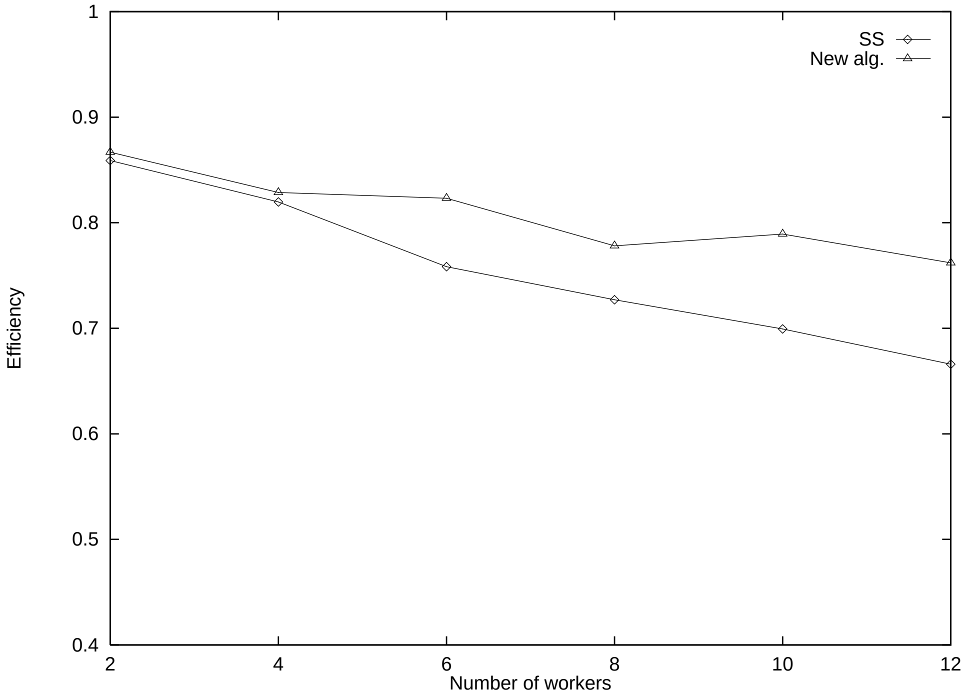


Figure 4.8: Efficiency for the Matrix Multiplication ($RowsA = 1200$).

improvement is 6% for six processors and 12% for twelve processors. These numbers would not justify the complexity of the implementation for our scheme. However, our scheme looks promising, if we consider that for the experiments we used a prototype, unoptimized implementation of our scheduling scheme. Moreover, note that the characteristics of the specific experiment are unfavorable to our scheme. This is because the machines used in the experiment are homogeneous and the created tasks have the same amount of work, while our scheme takes into account the heterogeneity of machines and the varying computation cost of tasks. In this unfavorable case, the scheduling overhead is only a liability to our scheme, because the information collected at run-time is not useful. Still, our scheme manages to increase performance.

4.7 Composite Applications

So far, we have only examined M-W applications consisting of one master and many workers. More complex applications can be composed using the building blocks provided by the metaprogramming model of the PPS. Here, we extend our scheduling scheme to support composite applications.

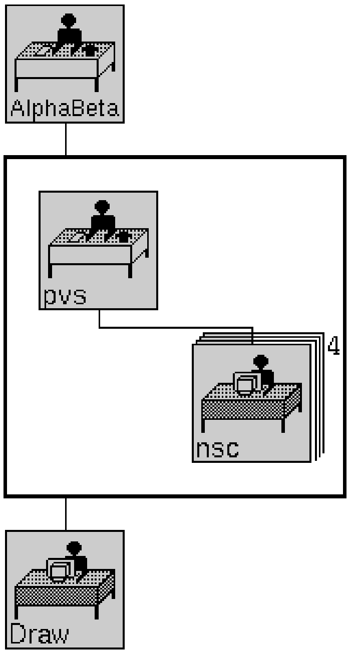
Generally, to manage the increased complexity, composite applications are structured hierarchically. The PPS may have to automatically insert processes that coordinate the user-specified processes. For example, in Enterprise when a number of masters generate work to be processed by more than one worker, a *manager* process has to be inserted between the masters and the workers to coordinate the distribution of work. For our purposes, the manager can be thought of as a worker to its masters and as a master to its workers.

By expanding the application hierarchy and adding the processes inserted by the PPS, we construct the process graph of the application. The *process graph* is a directed acyclic graph where the nodes represent the processes, and the edges connect the communicating processes, directed from the caller to the callee process. The processes without any outgoing edges are called *leaf processes*, otherwise they are called *non-leaf processes*. A task is called a *leaf* or *non-leaf task*, depending on whether it is processed by a leaf or non-leaf process. Note that the process graph should not be confused with the DAG, since the process graph refers to processes and not to tasks, and the communication is bidirectional. In Figure 4.9, the Enterprise expanded hierarchical diagram (Figure 4.9(a)) of the Alpha-Beta application is shown along with its corresponding process graph[5] (Figure 4.9(b)). `AlphaBeta`, `pvs`, `nsc1`, `nsc2`, `nsc3` and `nsc4` are non-leaf processes, while `Draw` is a leaf process.

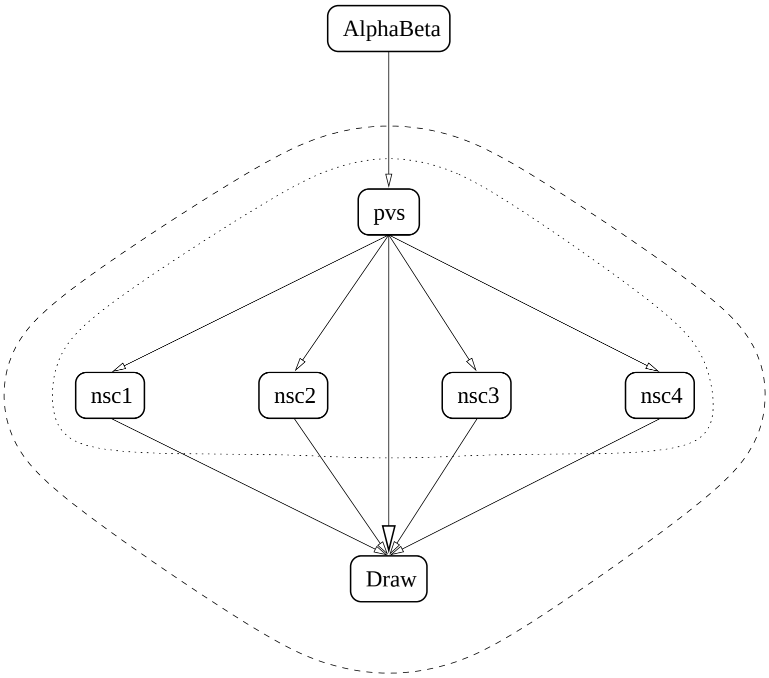
To make good scheduling decisions on composite applications, the entire process graph has to be considered. Clearly, it is difficult to model the interactions of the different application parts in order to minimize the global execution time. Therefore, we attempt primarily to minimize the execution time of each part separately with the expectation that this will minimize the global execution time. Obviously, the global execution time will be decreased only when the execution time of the parts belonging to the critical path of the application is decreased.

Here, we decompose the composite application into M-W subprograms. Each subprogram consists of either a master and its workers, or a worker and its masters. A *one-to-many* M-W subprogram consists of a node and all its callee nodes from the same asset. Similarly, a *many-to-one* M-W subprogram consists of a node and all its caller nodes. For example, in Figure 4.9(b) a one-to-many M-W subprogram

[5]Details about the Alpha-Beta application can be found in [18].



(a) Flattened hierarchical diagram.



(b) Corresponding process graph.

Figure 4.9: The Alpha-Beta application.

contains the processes `pvs` (master), and `nsc1`, `nsc2`, `nsc3`, `nsc4` (workers), but it does not contain `Draw`, since this process is from a different asset. A many-to-one M-W subprogram contains the processes `pvs`, `nsc1`, `nsc2`, `nsc3` and `nsc4` (masters), and `Draw` (worker).

We have already addressed scheduling for one-to-many M-W subprograms where the workers are leaf processes. In the following, we will discuss the extensions needed and the problems encountered to the above scheduling algorithm in order to incorporate many-to-one M-W subprograms, as well as subprograms where the workers are non-leaf processes.

4.7.1 Dynamic Scheduling for Many-to-One Master-Worker Subprograms

In many-to-one M-W subprograms, the masters are, at the same time, workers for their master(s). This is because we are dealing with an inherently sequential model, so somewhere up the process graph there should be one root (master). Since the masters are workers too, it means that the tasks they generate are related to the tasks they process. So, for example, when two masters receive tasks with different amounts of work, then one may create more or larger tasks than the other.

In one-to-many M-W applications, the master decides the measure to use. Here, where more than one master exists, different masters may decide to use different measures. Clearly, if the measure is the user-provided estimation function, then all the masters will use it. Still, with the other measures, which are guessed at run-time, there exists this possibility. This can occur when one master decides that the measure should be the size of the arguments and another master that it should be the task creation order. To overcome this problem, in many-to-one subprograms the worker itself decides which measure will be used.

We notice that since the worker executes or distributes (if it is a manager) the tasks starting from the largest ones, then the smaller tasks will be executed later. Because it is more likely that the smaller tasks will originate from the slowest processors (because of the scheduling on the upper levels), this will lead to the execution of the tasks from the faster processors first. So, this strategy is unfair and can eventually lead to live-lock of the slowest workers (since they will wait longer for the replies of their tasks). Moreover, this may create a "domino effect" to the upper levels of the process graph, because now the parent processes of the slowest masters may have to block too.

However, this is not really a problem, since it makes sense to process the tasks from the faster processors first. This is because we prefer the blocking times of the faster processors to be short, in order to minimize the wasted resources. Eventually the faster workers will have to wait for the slowest workers, since at an upper level they (or their parents) have the same master. On the other hand, this will not be an issue, since the faster workers will obtain more work by distribution or redistribution of tasks.

A problem arises if the creation order of the tasks is used as a measure. In this case, separate masters would use the same creation order number but have different computation costs. As a result, the coefficients of the computation time function may have wrong values.

4.7.2 Computation Time for Non-Leaf Tasks

Our scheduling algorithm makes decisions based on the computation cost from past task executions. For a leaf task where the only cost is processing, its computation time is equal to its elapsed time - that is the time between the start and the end of its execution. For non-leaf tasks, the computation time should be different from the elapsed time. This is because the elapsed time, except for the processing cost, includes the time spent sending subtasks, and waiting for and receiving their replies. Then again, the elapsed time does not reflect the cost of the subtasks.

Ideally, the non-leaf tasks with the largest processing cost should be assigned to the fastest processors. At the same time, the non-leaf tasks that stay blocked the longest amount of time should be assigned to the slowest processors, so fewer processor cycles will be wasted.

Here, we exclude the communication cost of sending subtasks and receiving their replies from the computation cost of a non-leaf task. Otherwise, even a fine-grained task can be falsely perceived as coarse-grained, since most of the time the communication cost of a task is high. Unfortunately, we have yet to find a good way to incorporate the blocking cost in the computation time, thus we exclude it too.

Instead, we add to the computation cost of a non-leaf task the computation cost of all its subtasks. However, this blurs whether a larger computation time means a non-leaf task with a larger computation cost, a non-leaf task with more subtasks, or subtasks with a larger computation cost. Still, it is reasonable to assume that the algorithm mapping the processes to processors will attempt to have the cumulative speed of the workers to be proportional to the speed of their master. This is because faster workers are needed to match the expected faster task creation rate of a faster master. Based on this assumption, it is not so important whether a task or its subtasks are large, since in both cases they will have to be distributed to the same process.

This suggests that the caller process should make decisions based also on the cumulative speed of a worker and its workers and not only on the speed of the worker itself.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

We proposed two scheduling algorithms, one for D&C and another for M-W computations, that capitalize on specific characteristics of the computations to achieve high performance. Then we compared our algorithms with two other simple and popular scheduling algorithms.

We showed that performance is improved by using more sophisticated scheduling algorithms. From the experiments, both our algorithms perform better than the simple scheduling schemes. However, the performance gains are not always proportional to the effort spent in the algorithms. Our D&C scheme is much simpler than our M-W scheme, yet it achieves better results.

Finally, we demonstrated that we can obtain better performance than the simpler schemes without any user involvement. For example, in our D&C scheme the user does not have to provide any information about the processor or the computation tree. On the other hand, with APRIL (and the previous Enterprise implementation) the user has to specify the processor tree.

5.2 Future Work

Our work in helping the developers of parallel applications to achieve high performance is far from complete. In the rest of this chapter, we discuss our future plans.

5.2.1 Scheduling Algorithms

More experiments are needed to evaluate the performance of our scheduling algorithms. We plan to use more applications with different characteristics. Furthermore, we will quantify the influence of each component and each parameter of the scheduling algorithms in the overall performance, and change the algorithms accordingly. For example, suppose that in D&C computations the performance improves slightly by sending the addresses of the other workers along with the tasks at the beginning of the computation. To simplify the D&C scheduling algorithm, this component should be dropped.

Usually, in a NOW, the machines and the network are shared, and the network is heterogeneous, consisting of subnets connected together. Consequently, the processor load and the communication costs may fluctuate. We plan to extend our scheduling schemes to take into account these fluctuations when scheduling decisions are made. The processor load can be easily incorporated in the current scheduling schemes by substituting the normalized speed of a processor with a value that combines the speed and the load of the processor. Still, from our initial experiments, we could not find a load index that would correctly predict the future load of a machine. More difficult will be to model the communication cost, since it depends on the communication medium and the traffic from other processors transmitted over the same wire at the same time.

Both scheduling schemes are centralized. As the number of workers increases, the coordinator (that is, the manager in D&C or the master in M-W computations) will become a bottleneck. We intend to make the schemes scalable by organizing the workers hierarchically. The coordinator can designate some workers as coordinators and delegate groups of workers to be administered by the new coordinators. These coordinators will behave as workers for the original coordinator, and as coordinators for their workers. Here, the problem is in deciding how many and which workers will exist in each group.

In M-W applications, we have an estimation about the computation time of a task. We can use this value to decide whether the task is coarse-grained. If it is, then it should be sent to another processor for execution. Otherwise, if the task is fine-grained, it should be either executed locally, or stored locally and be transferred to another processor only after a coarse-grained chunk of tasks has been accumulated. This method will improve the performance, and increase the portability of the application between different platforms. For example, suppose that the user implements a parallel program on a multiprocessor. Since in this architecture the communication cost between processors is negligible, the user can decompose the work into small tasks to achieve better load balance. When this program is ported to a NOW (where the communication cost is higher), these tasks may become fine-grained. By letting the run-time system combine the tasks into coarse-grained chunks, the user does not have to change anything in the program and the performance does not have to be sacrificed.

Moreover, we want to investigate which components of our M-W scheduling scheme can be used in scheduling data parallel applications. Our scheme cannot be used as it is for this type of application, because it was designed for control parallel applications which have different characteristics from the data parallel ones. For example, in data parallelism the work is executed on the processor where the data exists (the *owner computes rule*), while in control parallelism the tasks can be executed on any processor. Because it is costlier to migrate data between processors than to redistribute tasks, in data parallelism we must be more careful in the initial decomposition and distribution of data, and more cautious in migrating portions of data to other processors.

5.2.2 Compiler and Profiling Information

In the proposed schemes, no information from the compiler or from previous runs of the programs (profiling) was used. We intend to use compiler and profiling information to decrease the scheduling overhead and make more intelligent decisions. When the number and computation cost of the tasks are known in advance, the scheduling overhead can be decreased by composing statically the chunk of tasks to be sent to each worker, and not keeping statistics.

A problem with profiling is that profiling runs should use a smaller problem size (and thus may have different number and computation cost of tasks) than the production runs. This is because for large programs (like most of the parallel applications are) the user cannot waste the resources to execute the application with the production problem size just to obtain profiling information (unless the production size problem is executed often e.g. weather forecast). So, a smaller problem size is used and the compiler has to extrapolate the production run statistics from the profiling data.

5.2.3 Resource Management

One of the problems that we will tackle in the future is the mapping of processes to processors. If the application is not composite, then it is straightforward to place the processes on the processors. Each worker is assigned to a different processor. The coordinator is placed on the same processor as a worker if there is enough memory to avoid swapping. Otherwise, it is placed on a processor alone. However, the problem becomes much more difficult when the application is composite. Then we have to take into account which processes communicate (to be placed on the same or neighboring processors), which ones run concurrently (to be placed on different processors), and which ones do most of the work (to be assigned to the fastest processors).

Bibliography

[1] Gail Alverson, Simon Kahan, Richard Korry, Cathy McCann, and Burton Smith. Scheduling on the Tera MTA. In Feitelson and Rudolph [12], pages 19–44.

[2] Lorenzo Alvisi, Alessandro Amoroso, Özalp Babaoğlu, Alberto Baronio, Renzo Davoli, and Luigi Alberto Giachini. Parallel Scientific Computing in Distributed Systems: The Paralex Approach. Technical Report UBLCS-92-2, Laboratory for Computer Science, University of Bologna, Bologna, Italy, Feb. 1992.

[3] Eshrat Arjormandi, William G. O'Farrel, Ivan Kalas, Gita Koblents, Frank Ch. Eigler, and Guang G. Gao. ABC++: Concurrency by Inheritance in C++. *IBM Systems Journal*, 34(1):120–136, Jan. 1995.

[4] Henry Baker, Jr. and Carl Hewitt. The Incremental Garbage Collection of Processes. In *Proceedings of the Symposium on Artificial Intelligence and Programming Languages*, pages 55–59, Rochester, NY, Aug. 1977.

[5] Adam Beguelin, Jack Dongarra, Al Geist, Robert Manchek, Keith Moore, and Vaidy Sunderam. PVM and HeNCE: Tools for Heterogeneous Network Computing. In Jack Dongarra and Bernard Tourancheau, editors, *Advances in Parallel Computing*, volume 6. North-Holland, 1993.

[6] Brad Calder, Dirk Grunwald, and Ben Zorn. Quantifying Behavioral Differences Between C and C++ Programs. *Journal of Programming Languages*, 2(4):313–351, 1994.

[7] S. Chakrabarti, A. Ranade, and K. Yelick. Randomized Load Balancing for Tree-structured Computation. In *IEEE Scalable High Performance Computing Conference*, pages 666–673, Knoxville, TN, May 1994.

[8] M. I. Cole. *Algorithmic Skeletons: A Structured Approach to the Management of Parallel Computation*. PhD thesis, Dept. of Computer Science, University of Edinburgh, Edinburgh, U.K., July 1988.

[9] R. Kent Dybvig. *The Scheme Programming Language*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1987.

[10] T. Erlebach. *APRIL 1.0 User Manual*. Technische Universität München, Germany, 1995.

[11] T. Erlebach. Private communication. July 1995.

[12] Dror G. Feitelson and Larry Rudolph, editors. *IPPS'95 Workshop: Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*, Santa Barbara, CA, Apr. 1995. Springer-Verlag.

[13] B. Freisleben and T. Kielmann. Performance Evaluation of Parallel Divide-and-Conquer Algorithms. Technical Report THD-BS-1993-01, Institute for System Architecture, Operating System Division, Dept. of Computer Science, University of Darmstadt, Mar. 1993.

[14] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM : Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, Cambridge, MA, 1994.

[15] Andrew S. Grimshaw. Easy to Use Object-Oriented Parallel Programming with Mentat. *IEEE Computer*, pages 39–51, May 1993.

[16] Andrew S. Grimshaw, Jon B. Weissman, Emily A. West, and Ed C. Loyot, Jr. Metasystems: An Approach Combining Parallel Processing and Heterogeneous Distributed Computing Systems. *Journal of Parallel and Distributed Computing*, 21(3):257–270, June 1994.

[17] Robert H. Halstead, Jr. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, Oct. 1985.

[18] Paul Iglinski, Steve MacDonald, Chris Morrow, Diego Novillo, Ian Parsons, Jonathan Schaeffer, Duane Szafron, and David Woloschuk. Enterprise User's Manual - Version 2.4. Technical Report TR 95-02, Dept. of Computing Science, University of Alberta, Edmonton, AB, Canada, Jan. 1995.

[19] R. M. Karp and Y. Zhang. Randomized Parallel Algorithms for Backtrack Search and Branch-and-Bound Computation. *Journal of the ACM*, 40(3):765–789, July 1993.

[20] Tae-Hyung Kim and James M. Purtilo. Load Balancing for Parallel Loops in Workstation Clusters. Technical Report CS-TR-3591, Institute for Advanced Computer Studies, Dept. of Computer Science, University of Maryland, College Park, MD, Jan. 1996.

[21] R. Knecht. Implementation of Divide-and-Conquer Algorithms on Multiprocessors. In J. D. Becker, I. Eisele, and F. W. Mündemann, editors, *Parallelism, Learning, Evolution*, Lecture Notes in Artificial Intelligence, pages 121–136. Springer-Verlag, 1989.

[22] D. A. Kranz, Robert H. Halstead, Jr., and E. Mohr. Mul-T: A High-Performance Parallel Lisp. In *SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 81–90, Portland, OR, June 1989.

[23] Clyde P. Kruskal and Alan Weiss. Allocating Independent Subtasks on Parallel Processors. *IEEE Transactions on Software Engineering*, 11(10):1001–1016, Oct. 1985.

[24] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor: A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, pages 104–111, San Jose, CA, June 1988.

[25] V. M. Lo, S. Rajopadhye, S. Gupta, D. Keldsen, M. Mohamed, and J. A. Telle. Mapping Divide-and-Conquer Algorithms to Parallel Architectures. In *International Conference on Parallel Processing*, volume III, pages 128–135. CRC Press, 1990.

[26] V. M. Lo, S. Rajopadhye, J. A. Telle, and X. Zhong. Parallel Divide and Conquer on Meshes. Technical Report CIS-TR-92-22, Dept. of Computer & Information Science, University of Oregon, Eugene, OR, 1992.

[27] Steven MacDonald. An Object-Oriented Run-time System for Parallel Programming. Master's thesis, Dept. of Computing Science, University of Alberta, Edmonton, AB, Canada, Spring 1996.

[28] E. Mohr, D. A. Kranz, and Robert H. Halstead, Jr. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, July 1991.

[29] Constantine D. Polychronopoulos and David J. Kuck. Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Transactions on Computers*, 36(12):1425–1439, Dec. 1987.

[30] Jim Pruyne and Miron Livny. Parallel Processing on Dynamic Resources with CARMI. In Feitelson and Rudolph [12], pages 259–278.

[31] Michael J. Quinn. *Parallel Computing: Theory and Practice*. McGraw-Hill, 2nd edition, 1994.

[32] Jonathan Schaeffer, Duane Szafron, Greg Lobe, and Ian Parsons. The Enterprise Model for Developing Distributed Applications. *IEEE Parallel & Distributed Technology*, 1(3):85–96, Aug 1993.

[33] SPEC Announces SPEC95 Benchmark Suites As New Standard for Measuring Performance. Press Release, Aug. 1995.

[34] Peiyi Tang and Pen-Chung Yew. Processor Self-Scheduling for Multiple-Nested Parallel Loops. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 528–535, Aug. 1986.

[35] Ten H. Tzen and Lionel M. Ni. Trapezoid Self-Scheduling: A Practical Scheduling Scheme for Parallel Compilers. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):87–98, Jan. 1993.

[36] D. B. Wagner. Portable, Efficient Futures. Technical Report CU-CS-609-92, Dept. of Computer Science, University of Colorado at Boulder, Boulder, CO, Aug. 1992.

[37] D. B. Wagner and B. G. Calder. Leapfrogging: A Portable Technique for Implementing Efficient Futures. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 208–217, San Diego, CA, 19-22 May 1993.

[38] Reinhold P. Weicker. A Detailed Look at Some Popular Benchmarks. *Parallel Computing*, 17:1153–1172, 1991.

[39] Reinhold P. Weicker. An Example of Benchmark Obsolescence: 023.eqntott. SPEC Newsletter, Dec. 1995.

[40] Jon B. Weissman. *Scheduling Parallel Computations in a Heterogeneous Environment*. PhD thesis, Dept. of Computer Science, University of Virginia, Charlottesville, VA, Aug. 1995.

[41] Jon B. Weissman and Andrew S. Grimshaw. Network Partitioning of Data Parallel Computations. Technical Report CS-94-17, Dept. of Computer Science, University of Virginia, Charlottesville, VA, May 1994.

[42] I-Chen Wu. Efficient Parallel Divide-and-Conquer for a Class of Interconnection Topologies. In W. L. Hsu and R. C. T. Lee, editors, *Proceedings of the 2nd International Symposium on Algorithms (ISA '91)*, Lecture Notes in Computer Science, pages 229–240, Taipei, Republic of China, Dec. 1991. Springer-Verlag.

[43] I-Chen Wu and H. T. Kung. Communication Complexity for Parallel Divide-and-Conquer. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*, pages 151–162, San Juan, PR, Oct. 1991. IEEE Computer Society Press.

[44] Mohammed Javeed Zaki, Wei Li, and Michal Cierniak. Performance Impact of Processor and Memory Heterogeneity in a Network of Machines. In *Proceedings of the 4th Heterogeneous Computing Workshop*, Santa Barbara, CA, Apr. 1995.

[45] Mohammed Javeed Zaki, Wei Li, and Srinivan Parthasarathy. Customized Dynamic Load Balancing for a Network of Workstations. Technical Report TR 602, Computer Science Dept., University of Rochester, Rochester, NY, Dec. 1995.

[46] X. Zhong, S. Rajopadhye, and V. M. Lo. Parallel Implementation of Divide-and-Conquer Algorithms on Binary de Bruijn Networks. Technical Report CIS-TR-91-21, Dept. of Computer & Information Science, University of Oregon, Eugene, OR, Sep. 1991.

Appendix A

Experimental Data

A.1 Environment Used for the Experiments

All the experiments were performed on a network of Sun ELC workstations (running SunOS 4.1.4), interconnected via a 10 Mbit Ethernet. PVM and all the applications were compiled using `gcc` 2.7.2. PVM 3.3.10 was used as the communication system (it was compiled using the `-O` optimization flag). The applications were executed on dedicated machines on a "quiet" network. The execution times were averaged over five executions (in fact, six measurements were taken, but the first one was never used). Efficiencies and speedups are reported against optimized sequential code. The applications were compiled using the `-O2` optimization flag.

A.2 Mandelbrot set computation data

Number of Processors	Execution times avg (seconds)	Coefficient of Variation (%)	Speedup	Efficiency
2	218.41	0.10	1.78	0.89
3	218.43	0.05	1.78	0.59
4	159.48	0.06	2.44	0.61
5	159.55	0.13	2.44	0.49
6	158.84	0.09	2.45	0.41
7	158.76	0.09	2.45	0.35
8	158.24	0.08	2.46	0.31
9	158.24	0.08	2.46	0.27
10	141.14	0.08	2.76	0.28
11	141.08	0.10	2.76	0.25
12	140.88	0.07	2.76	0.23
13	140.93	0.09	2.76	0.21
14	140.91	0.08	2.76	0.20
15	140.87	0.10	2.76	0.18

Table A.1: APRIL results for the Mandelbrot set computation.

Number of Processors	Execution times avg (seconds)	worst (seconds)	Coefficient of Variation (%)	Speedup	Efficiency	Improvement (%)
2	212.99	215.11	1.07	1.83	0.91	2.48
3	208.63	209.39	0.38	1.87	0.62	4.48
4	157.51	159.29	1.18	2.47	0.62	1.23
5	153.25	156.49	1.21	2.54	0.51	3.95
6	146.23	148.08	1.17	2.66	0.44	7.94
7	154.17	156.13	1.40	2.52	0.36	2.89
8	88.73	109.18	13.13	4.39	0.55	43.93
9	88.74	92.08	3.51	4.39	0.49	43.92
10	76.76	81.96	7.11	5.07	0.51	45.61
11	85.50	107.61	23.38	4.55	0.41	39.39
12	64.76	66.79	2.98	6.01	0.50	54.03
13	67.72	74.73	11.88	5.75	0.44	51.95
14	59.83	60.95	2.22	6.50	0.46	57.54
15	53.31	54.76	2.35	7.30	0.49	62.16

Table A.2: Enterprise results for the Mandelbrot set computation.

A.3 Matrix multiplication data

Note that for the sequential matrix multiplication we used a naive multiplication algorithm (no unrolling, interchange of the inner loop, or multiplication by subarrays was used). According to the speedup definition, we should use the best possible sequential execution time. However, in our experiments it was more important to measure the improvement of our scheme over SS than to measure the speedup. We opted to use a naive multiplication algorithm, because it was easier to implement. If we used the best possible sequential algorithm, then the speedups presented here would be lower.

Number of Workers	Execution times avg (seconds)	Coefficient of Variation (%)	Speedup	Efficiency
2	224.30	0.08	1.71	0.86
4	118.29	0.35	3.25	0.81
6	83.39	0.90	4.60	0.77
8	65.61	1.01	5.85	0.73
10	56.01	1.74	6.86	0.69
12	48.95	2.39	7.84	0.65

Table A.3: SS results for the Matrix Multiplication ($RowsA = 900$).

Number of Workers	Execution times avg (seconds)	worst (seconds)	Coefficient of Variation (%)	Speedup	Efficiency	Improvement (%)
2	219.40	222.72	1.27	1.75	0.88	2.18
4	112.58	113.51	0.77	3.41	0.85	4.83
6	78.24	81.26	2.39	4.91	0.82	6.18
8	62.91	67.68	5.13	6.10	0.76	4.11
10	49.61	50.97	1.75	7.74	0.77	11.43
12	42.76	44.07	1.90	8.98	0.75	12.64

Table A.4: Enterprise results for the Matrix Multiplication ($RowsA = 900$).

Number of Workers	Execution times avg (seconds)	Coefficient of Variation (%)	Speedup	Efficiency
2	298.48	0.26	1.72	0.86
4	156.38	0.45	3.28	0.82
6	112.70	0.88	4.55	0.76
8	88.16	1.09	5.82	0.73
10	73.31	0.43	6.99	0.70
12	64.16	0.99	7.99	0.67

Table A.5: SS results for the Matrix Multiplication ($RowsA = 1200$).

Number of Workers	Execution times			Speedup	Efficiency	Improvement (%)
	avg (seconds)	worst (seconds)	Coefficient of Variation (%)			
2	295.74	301.78	1.39	1.73	0.87	0.92
4	154.70	160.91	3.11	3.31	0.83	1.07
6	103.82	106.86	1.74	4.94	0.82	7.88
8	82.37	92.27	6.80	6.22	0.78	6.57
10	64.96	66.09	1.22	7.89	0.79	11.39
12	56.07	58.30	2.34	9.14	0.76	12.61

Table A.6: Enterprise results for the Matrix Multiplication ($RowsA = 1200$).