FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

A MapReduce Construct for Yap Prolog

Joana Côrte-Real

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Supervisors: Prof. Inês Dutra and Prof. Ricardo Rocha

July 2013

Resumo

Neste trabalho, desenhou-se e implementou-se uma primitiva de alto nível para Prolog, baseada no paradigma de programação MapReduce. MapReduce é um modelo de programação funcional popularizado pela Google em 2008, apesar de ter raizes consideravelmente mais antigas. Este modelo é constituído por duas operações simples, *map* e *reduce*, que podem ser facilmente aplicadas a um vasto número de algoritmos. Prolog, por sua vez, é uma linguagem assente em lógica de predicados de primeira ordem, com elevado poder declarativo, o que permite ao programador focar-se no algoritmo de resolução de um dado problema em vez de nos seus detalhes de mais baixo nível. Prolog é um modelo de programação vocacionado para o armazenamento e tratamento de dados, havendo mesmo aplicações que estão preparadas para fazer inferências sobre esses dados. Um construtor MapReduce aplicado neste cenário permitiria escalar eficientemente todo o processo, reduzindo muito significativamente o tempo de execução.

A criação de uma primitiva de programação baseada em MapReduce para Prolog apresenta três contribuições principais: (i) proporciona ao utilizador uma construção de alto nível de abstração no modelo funcional MapReduce, mantendo a característica declarativa dos programas; (ii) disponibiliza ao utilizador uma construção não existente em Prolog e que é representativa de aplicações em várias áreas; (iii) permite paralelização, acelerando a execução de programas que utilizam esta primitiva. Este último ponto é particularmente relevante dado que os processadores de vários núcleos se têm tornado a escolha dominante em equipamentos informáticos, mesmo aqueles destinados a uso pessoal. Este facto, aliado à crescente quantidade de dados que, cada vez mais, são produzidos diariamente, faz com que uma ferramenta que utilize arquiteturas multiprocessador – eficientemente – para processamento de dados, suscite interesse.

O foco de MapReduce para Prolog são as arquiteturas multi-processador, apesar de a nossa primitiva estar preparada para suportar ambientes híbridos (memória distribuída e memória partilhada), de forma implícita e transparente para o utilizador. MapReduce para Prolog foi implementado no sistema Yap e é constituído por uma arquitetura do tipo mestre-escravo, onde o mestre é responsável pela divisão do trabalho e os escravos pelo processamento das tarefas que lhes são atribuídas. A interface do construtor dispõe ainda de vários níveis de customização, e um dos objetivos deste trabalho é a integração do nosso contrutor MapReduce com o sistema Yap sob a forma de uma biblioteca. O nosso sistema foi testado com sucesso através da construção de quatro aplicações distintas comuns na literatura: duas contendo dados numéricos, e as restantes contendo termos de Prolog. Os testes foram feitos com duas implementações para a mesma interface de programação, uma para um *cluster* de máquinas e outra para uma arquitetura multi-processador. Determinou-se que o construtor escalou consistentemente o tempo de execução de forma quase ideal para todas as aplicações, quer em memória partilhada, quer distribuída. Desenvolveram-se e analisaram-se quatro técnicas de escalonamento de trabalho, das quais as mais eficazes serão disponibilizadas na versão final da biblioteca. Finalmente, avaliou-se ainda o efeito da variação do tamanho das unidades de trabalho distribuídas aos escravos a fim de establecer os parâmetros por defeito para MapReduce para Prolog.

Abstract

This work's aim was to design and implement a high-level Prolog primitive, based on the MapReduce programming paradigm. MapReduce is a programming model made popular by Google in 2008, even though its origins are more remote. It is composed by two simple operations, *map* and *reduce*, which can easily be applied to numerous algorithms. On the other hand, Prolog is a first-order logic predicate language with significant declarative power. This allowing the programmer to focus on the resolution strategies for a problem in preference to the execution technicalities. Prolog is also especially suited for data storage and processing; in fact, ILP deals with making inferences from that data. A MapReduce construct applied in these circumstances would be able to efficiently scale that process and thus significantly reduce execution times.

Including a MapReduce programming primitive in Prolog has three major benefits: (i) to make available a high-level abstract construct which implements the MapReduce functional model maintaining the declarative nature of the programs; (ii) to give access to a previously non-existent Prolog construct which is relevant to applications in numerous fields of knowledge; (iii) to allow for parallelism, thus speeding-up the execution of programs using this construct. The latter is particularly relevant now that multicore processors have become the favourite choice to assemble machines, even those for personal use. This, along with the fact that there are increasingly larger data processing requirements in everyday life, renders a framework using multicore architectures for *efficient* data processing highly relevant.

MapReduce for Prolog's focus are multicore architectures, but our primitive supports hybrid environments (shared and distributed memory), implicitly and transparently. MapReduce for Prolog was implemented in the Yap system and it follows a master-slave paradigm, in which the master is responsible for dividing and assigning the work and the slaves for processing the chunks dispatched to them. This construct's interface has various customisation levels, and our aim is that it will come to integrate the Yap Prolog system as built-in construct. Our system was successfully tested using four distinct applications common in the literature: two of these were numeric, and the other two were composed of Prolog terms. The test were made using two implementations for the same programming interface, one for a cluster of machines and another for a multicore architecture. It was determined that our construct scaled almost ideally for these datasets, both in shared and distributed memory. Four scheduling methods were also developed and assessed, and the two more efficient ones will be made available in the final version of the library. An evaluation of the effect of the chunk size variation for different datasets and scheduling methods was performed as well, in order to define standard parameters for MapReduce for Prolog.

Acknowledgements

Firstly and foremost, I would like to thank my supervisors Inês Dutra and Ricardo Rocha for their constant attention and support. Professor Inês sat tirelessly at my workstation, helping me overcome problems, and Professor Ricardo always gave me sharp and very pertinent advice, at just the right moment. I am most grateful to both for their time and interest, which contributed greatly towards the quality of this thesis work.

I am grateful to Hugo Ribeiro, for readily providing all the technical support I needed, and still teaching me while doing it. I would like to thank PhD student Miguel Areias for helping me track down an elusive issue, that surely would have been much more so without his help. I am also grateful to Professor Vítor Santos Costa, for showing me around the Yap Prolog system when I was just getting started, and for supporting me the for the remainder of this work. I would like to thank Joana Dumas, the CRACS secretary, for making all the bureaucracy simpler for me.

I am grateful to project LEAP (PTDC/EIA-CCO/112158/2009) and to Fundação da Ciência e Tecnologia for their support, under the research grant BI/120048/Leap_CRACS, from which I have profited during the development of this thesis.

To PhD student João Santos, Rui Vieira and Hugo Sousa, my colleagues at the DCC, thank you for all the moments well spent. To my colleagues at FEUP I am grateful for five wonderful years, full of new experiences and lasting friendship. To Paulo Alcino and Joana Grifo, both physicists and currently residing in London, may you be *both there and here*.

I am grateful to my family, for constant and unfaltering support; my mother Paula, my aunt Isabel and my grandmother Maria Emília even offered to read this document. Bless you! To my boyfriend Luís I would like to thank his continual reminder that I can surpass myself.

I am deeply indebted to you all, thank you again.

Joana Côrte-Real

"Once you eliminate the impossible, whatever remains,
no matter how improbable, must be the truth."

Sir Arthur Conan Doyle

Contents

List of Figures

List of Tables

Abbreviations

API	Application Programming Interface
DCC	Departamento de Ciência de Computadores
DMA	Distributed Memory Architecture
GM	Global Master
HDFS	Hadoop Distributed File System
ILP	Inductive Logic Programming
IP	Internet Protocol
LM	Local Master
Prolog	PROgrammation en LOGique
MPI	Message Passing Interface
SL	Slave
SLD	Selective Linear Definite
SMA	Shared Memory Architecture
WAM	Warren Abstract Machine
Yap	Yet Another Prolog

Chapter 1

Introduction

In the modern world there is a growing need for the efficient processing of immense amounts of data in a simple and incisive way. Hardware is becoming increasingly more complex and powerful, as well as much more affordable, due to competitive manufacturing processes and greater economies of scale. In particular, the vulgarization of multicore processors presents a clear opportunity for taking advantage of these components' architecture in order to significantly shorten task processing times using parallelism, even in a common personal laptop. As such, there is an emerging demand for straightforward parallel interfaces for otherwise computationally taxing tasks, in which users will not necessarily have an extensive programming background.

Logic Programming is strongly based on mathematical and logical concepts, making it an accessible tool for users with relatively little programming experience but a relevant scientific background, and allows for implicit parallelization by hiding implementation details from the programmers. The distinct declarative style of Prolog makes it an ideal tool for analysing, processing or making inferences about data, having applications on a wide range of areas of knowledge, such as machine learning [3], natural language processing [4] or program analysis [5], among many others. The Prolog language also presents an interesting alternative to standard relational databases, having some relevant applications in this area as well [6]. Furthermore, declarative languages are typically very high level languages, meaning that the Prolog's syntax is mostly independent of its low level implementation. This allows users to detach their algorithms from almost any concern with technical detail, since compilers already implement effective translation mechanisms.

In addition to ease-of-use, Prolog's non-determinism allied to its declarative semantics invite the use of parallelism as a tool to improve program efficiency, without increasing the program's complexity whatsoever. The aim of this work is then to introduce a widely known parallel programming model - MapReduce - into the Prolog language, by designing and implementing an API native to the Yap Prolog system [1]. The original MapReduce model [7] allows for handling data throughout a cluster of machines, thus processing it in parallel and under a distributed architecture. This system is composed of two user-defined operations - Map and Reduce - which conduce to an extremely flexible programming model due to their structure.

Prolog is a programming language specially suited to store and analyze data, and even to make inferences based on that data. This is a feature that is increasingly more requested by programmers, but the scalability of the existing data analyzing tools in Prolog has often been questioned. The aim of this MapReduce for Prolog implementation is to provide the language – and the Yap system in particular – with a flexible and easy-to-use framework for data processing in Prolog, with focus on native data types. The MapReduce programming model is an ideal choice, since it is both well-known and straightforward, presenting programmers with an attractive framework, which hides all parallelization details but whose performance is efficient.

The MapReduce construct presented in this document can not only establish a processing grid within a cluster of machines, but it can also take advantage of multicore processors in each machine, if they exist. The latter feature is found to be especially relevant now that most processors being built already incorporate at least two physical cores. Our implementation of a MapReduce construct is aimed at relatively modest computing capabilities, and small to medium dataset sizes. Under these conditions, it has proved to be agile and flexible, as well as highly efficient in terms of speeding-up process executions for both computing and logical applications.

1.1 Thesis Purpose

Due to the vulgarization and growing affordability of computers, it is now common for people to have access to more than one machine. In the last decade these machines were often equipped with multicore processors, which have gained increasing significance as a standardized and inexpensive option. Both these facts combined provide ample possibilities for software designers to take advantage of this emerging type of architecture composed of several machines with multicore processors but relatively modest capabilities.

This thesis' contribution lies in the fact that the MapReduce Construct for Prolog is applicable to both multicore processors and clusters of machines, thus attaining high efficiency and much shorter processing times in running tasks, while still using a straightforward declarative semantic which implements the MapReduce model. This model is composed of two very basic operations which are widely suitable for the processing of data concerning various applications [8]. It could be argued that the lack of complexity of this model renders it trivial research-wise, nevertheless we find that its simplicity is one of the key features which makes the paradigm so widely accepted and used.

Logic Programming could be considered an unusual choice to implement a MapReduce model since its focus is not on implementation details such as basic parallel constructs (threads or processes); however, it presents an unique suitability to store facts in a background knowledge form and draw conclusions from them, whilst it can still efficiently process most other forms of data. Some criticisms have been made to logic programming languages regarding the reduced autonomy of the programmer in terms of system definition and parallel optimization. This work addresses that issue by implementing several possible levels of customization, from basic usage of the construct to the definition of a grid of machines with their respective IP addresses and multicore architectures. We hope this will effectively accommodate needs from users with strikingly different goals and backgrounds.

1.2 Thesis Outline

This document is structured in 6 chapters, reflecting the different stages of the work. A brief description for each one is provided below.

Chapter 1: Introduction. The current chapter.

Chapter 2: Background and Related Work. Presents relevant information on both logic programming and MapReduce systems. In the first section, Prolog language basics such as first order logic and Horn clauses are briefly described, followed by some examples of Prolog syntax and semantics and an explanation regarding the various types of parallelism that can be exploited in this language. This section also includes an overview of the Yap Prolog system and of declarative programming in general. The latter section defines the MapReduce model and addresses several implementations described in the literature. It then details the most relevant works to this thesis, providing an in-depth analysis of their features.

Chapter 3: MapReduce in Prolog. The design of the system is detailed in this chapter, both for clusters and for multicore architectures. The interface is also presented, as well as some relevant examples of usage.

Chapter 4: Methodology. This chapter includes a thorough description of the datasets used to validate the implementation. There is also an account of the machines used, as well as the evaluation parameters for the results presented in the following chapter. In addition, the Yap Prolog file system is introduced and a number of modifications and difficulties encountered are detailed here. Some of the most relevant Yap Prolog libraries are briefly mentioned as well.

Chapter 5: Results. This section contains firstly a quantitative account of the results from the experiments with the system. Here are included the speedup plots and other measurements considered pertinent to the systems' assessment. The second part of this chapter contains a qualitative description and discussion of the results, in order to provide some insight on relevant points.

Chapter 6: Conclusions and Future Work. The work is summed up and some suggestions for the future are detailed.

Chapter 2

Background and Related Work

This chapter contains a summary of relevant state-of-the-art for both Logic Programming and the MapReduce model. The Prolog language is introduced in some detail, and examples of usage are provided. An explanation on how different forms of parallelism can be applied, with reference to those examples, serves as preamble for the description of the MapReduce model. A number of MapReduce implementations are presented and finally, an application of MapReduce to Logic Programming is described in some detail, since it is a pertinent start point for this work.

2.1 Logic Programming

Since the mid-1900's until the present time numerous programming languages have been developed. As such, a need arose to identify common features amongst the programming languages so as to classify them accordingly. Therefore, four main paradigms have emerged from this process, matching every programming language to one of these categories: imperative programming, functional programming, logic programming or object-oriented programming. Imperative programming's semantic is composed of strict translations from machine language to a set of user commands, whilst functional programming is concerned with features such as recursion or pattern matching. Object oriented languages are the most recent paradigm and are versatile and very complete in terms of functionalities. Declarative languages are also a relatively recent paradigm – stemming from functional programming – and they aim at creating a dettachment between a program's goal and its execution details by enhancing the *functional* characteristics of the language in preference to its technicalities. This allows the programmer to focus on the way in which the program should be executed rather than how the actual computation is performed; the programming task then becomes both easier and more efficient, as stated by J. W. Lloyd in [9]. Logic programming languages are a subset of declarative languages, meaning that the programmer is only required to specify what a program should do, and the language is responsible for executing the specification in a fairly efficient way. It is evident that this paradigm of programming allows for a detachment between the *logic* goals of the program and its *execution* goals, which can be explored towards greater efficiency. There are various languages in the logic programming category (such as the Datalog [10] or Godel [11] languages), but only the Prolog family will be discussed here since the remaining languages are out of the scope of this work. Prolog first appeared in 1972 [12], in result of extensive research on an experiment whose aim was to develop a strategy for computers to interpret natural language. Since then, it has evolved and branched out into a number of distributions such as SWI-Prolog [13], SICStus Prolog [14] or Yap [1].

2.1.1 The Prolog Language

In 1969, Cordell Green developed an automatic theorem proving procedure [15] applicable to first-order logic systems, from where stem the numerous declarative programming languages in existence today. Prolog's syntax, in particular, is composed of clauses that can be expressed as a conjunction of literals, also known as Horn clauses. This type of logical construction is a subgroup of first-order logic, and as such it is not only resoluble and complete given a set of axioms but it is also closed: the resolvent of two Horn clauses is also a Horn clause. This fact makes it possible and convenient to recursively solve these clauses using resolution methods based on SLD [15].

In 1983, David Warren introduced a memory architecture and an instruction set, later named the Warren Abstract Machine [16], meant to efficiently translate Prolog instructions to lower level code, then to be resolved. The WAM still presently sets a relevant standard amongst Prolog compilers [17]. It is important to note that whilst the order of the terms in a clause is mathematically indifferent, it can be computationally taxing. More recent additions have been made to WAM and other abstract machines in order to decrease the side-effects caused by parallel term computation, deriving from the use of Or and And parallelism, to be described later.

Prolog is then a language composed of *rules* and *terms*, and their mutual interaction. It has been argued that the logic programming paradigm should have been named the relational programming paradigm [9] since that terminology better describes the nature of the language. A term is the basic Prolog language entity, and it can be an *atom* (starts with lower-case letters or is enclosed in single quotation marks), a number (float, integer), or a compound term (also named a functor). An example of the latter would be a Prolog list such as `[L1, L2, ..., LN]`. A term can also be a free variable (its name starts with an uppercase letter or an underscore) which is type-less until it is *unified*, meaning that a value is then assigned to the variable. Since Prolog has no destructive assignment of variables, unification for each variable can occur only once. However, *backtracking* allows for unbinding already unified variables, since Prolog stores *choicepoints* and can restore a previous program state so as to explore all possible solutions.

A rule in Prolog is necessarily a Horn clause, composed by a head and a body, and follows the structure presented in Equation 2.1.

$$head :- body\_clause_1, body\_clause_2, ...body\_clause_N. \tag{2.1}$$

A rule's head and body are related by the operator `:-`, which is an implication: for the head to be true, the body must also be true. A rule can have no body – the equivalent to 2.2.

$$head : -true. \tag{2.2}$$

In these cases the rule is named a *fact* and represents a logical tautology in the program's scope. The set of rules and facts of a Prolog program is called its *clauses*. The rule names in a program are also called *predicates*, and a predicate can have several clauses with different *arity* (number of predicate arguments). The body of a rule is composed of a sequence of *goals*, interacting with one another through *connectives*, or operators; in this case `,/2` corresponds to the AND connective. Each goal represents a call to a predicate, which is then determined to be true or false. It is thus evident that the execution of a Prolog program requires both a *goal selection rule* to determine which goal is to be called next, and a *search rule* to choose which alternative of a goal to explore, if several exist. Prolog's resolution employs left-to-right goal selection and an depth-first search strategy, and each resolution step taken is called *reduction* or *logical inference*. Since Prolog is a programming language and thus not purely logical, it requires meta and extra-logical predicates such as input/output operations, arithmetic operations or the *cut* operator. The latter must be used under some circumstances for program correction, and it can also be helpful in expediting execution by *pruning* the search tree and thus set aside unexploited alternatives. This operator is an example of a non-logic predicate since it is sensitive to the order in which the goals are exploited.

Figure 2.1 gives an example of a basic Prolog program illustrating most of these concepts. In

```
cat(tom).
mouse(jerry).
cheese(roquefort).
cheese(emmental).

eats(X,Y):-cat(X),mouse(Y).
eats(X,Y):-mouse(X),cheese(Y).
```

Figure 2.1: Example of basic Prolog program

this program there are four assertions, or facts: `tom` is a cat, `jerry` is a mouse and `roquefort` and `emmental` are both cheese. There is also a rule, or predicate, composed of a head `eats(X,Y)` and a body containing the definition of eating. This rule expresses the fact that either cats eat mice (first clause of `eats/2`) or mice eat cheese (second clause of `eats/2`). Figure 2.2 contains some queries one could now pose regarding the program above (see Figure 2.1).

When analysing Figures 2.1 and 2.2 it becomes evident that the two parts - or clauses - of the predicate `eats(X,Y)` are independent from each other in the sense that they do not have side effects on one another. The fact that `tom` is a cat is detached from the fact that `emmental` is a type of cheese, and so it follows that these calculations could be made simultaneously and that would not alter the final answer of the query. This simple example serves to demonstrate the fitness of Prolog languages to the application of implicit parallel execution, which will be discussed in more

```
?- eats(Anything,tom).
        no
?- eats(tom,Anything).
        Anything=jerry?
        ;
        no
?- eats(jerry,Anything).
        Anything=roquefort?
        ;
        Anything=emmental.
```

Figure 2.2: Example of basic Prolog queries and answers.

detail in the following section.

2.1.2 Parallelism in Prolog

Parallelism in the Computer Science domain means to split a program in concurrent parts and execute them simultaneously. This if often done with *multi-threading*, using only one machine, but it can also support many machines running the same program at once. Also, parallelism can be divided in two categories, depending of how aware the user is of the parallelization mechanism. Implicit parallelism takes place when the programmer writes the code as if it were going to be executed sequentially and the system is responsible for executing the code concurrently, for faster and more efficient execution when compared to the sequential case. Explicit parallelism typically obtains even better results in terms of efficiency, since the user can tune the system for optimum performance. This, however, requires systems to provide a framework in which the user can decide how he/she wants to run the program in a parallel way. These two forms of parallelism have radically different applications and implementations, and the focus of this document is on an explicit parallel model; however, we explore it implicitly by default, hiding the parallelization details from the user. Different levels of explicit parallelism can easily be incorporated in the MapReduce for Prolog construct presented in this document because it provides different levels of customization, rendering it possible for more experienced users to explicitly call parallel predicates in the system. According to [18] there are three ways in which implicit parallelism can be explored in Prolog languages: And-parallelism, Or-parallelism and unification parallelism. A brief description of each follows, with reference to Figs 2.1 and 2.2.

And-parallelism can be used when there is more than one subgoal in a resolvent, meaning that the parallel executions either compete or cooperate to find a solution. This type of parallelism can be dependent or independent, depending on whether there are variables common to the branches which have not been unified prior to the query. Independent and-parallelism could be applied given the following query: `?-eats(Something,Anything).`, since it can be broken down as two clauses: `cat(Something),mouse(Anything).`, which could both be executed simultaneously. An example of dependent and-parallelism would be `?-eats(Something,Something).`, using the same break-down structure.

Or-parallelism can be applied when more than one rule head unifies with a query. Thus, the solution space is searched concurrently, and in effect each search can lead to different valid solutions. This form of parallelism applies when a query such as `eats(Something,Anything).` is called, since there are two different clauses corresponding to the `eats/2` predicate.

Unification parallelism can exist when a term with arity greater than one needs to be unified. In such a case, the unification of its arguments can be done in parallel: `eats(jerry,cheddar)=eats(Mouse,Cheese).`. In this case, the variables `Mouse` and `Cheese` can be bound to `jerry` and `emmental` values concurrently.

Most Prolog systems currenlty available do not support parallelism [18]. The Yap and the SICStus Prolog are two examples of systems that support implicit or-parallelism. Also, some systems such as Yap or SWI Prolog maintain explicit parallel constructs (for instance, thread support).

2.1.3 The Yap System

Yet Another Prolog system [1] first appeared in 1984 in University of Porto and presented a WAM based design with some improvements, namely a very fast emulator written in assembly [1]. However, in the mid 90's some portability issues regarding the system emulator assembly code raised, forcing the Yap developers to revert to a C-based emulator, which at first proved to be much slower, but whose performance has increased greatly over the past years [19]. Also, at this point, some additions were made to the Yap system so as to support parallelism [20] and tabling mechanisms [21]. Because Yap is meant to provide support not only for small applications, but also for applications which require the manipulation of large and complex databases, in the past few years three very important additions have been made to the system. From [1] a short summary of these features is presented below.

The Just-In-Time Indexer (JITI) allows for indexation of both multiple arguments, compound terms and multiple modes of usage. Even though JITI can have a cost in terms of memory usage, it is generally thought that the advantages in runtime outweigh it [22].

The Sequential Tabling Engine provides runtime support for sequential and dynamic mixed-strategy tabling. This mechanism has proved yield good result when compared to other Prolog systems in the literature [23], [24].

The Or-Parallel Tabling Engine uses incremental stack copying to increase runtime speeds, and it has been shown in [24] that this methodology is successful for systems with medium parallelism.

Yap is one of the fastest Prolog systems in existence, being highly portable due to its C source code and very complete, integrating several modules of I/O operations, threads and databases. The work described in this document is implemented on top of the Yap system.

2.2 The MapReduce Framework

MapReduce is a programming model developed by Google in the early 2000's [7] aimed at processing large amounts of data. As the name suggests, it is composed of two elementary operations: *map* and *reduce*, which are based on primitives originally introduced in functional programming languages such as Lisp. The map operation applies a transformation to a set of key/value pairs, resulting in another set of the same size consisting of pairs with the same key but a *mapped* value. The reduce operation groups all the mapped pairs with the same key and aggregates their values, usually into one - or no - result. The pseudocode in Figure 2.3 illustrates the functions described before. The `aux_aggregator` operation is independent of both the data being processed and the map and reduce operations, rendering it autonomous from the remaining program; this operation allows the user to run different kinds of data on the same MapReduce call and group then using a `key`. This feature is specific to Google's MapReduce implementation and it is not included in the MapReduce for Prolog construct because it was found to be unnecessary. The size of the datasets our construct is aimed at does not justify burdening the framework with another mandatory operation and since MapReduce for Prolog can be used iteratively, the user can simply make one call for each data type in a loop.

Figure 2.4 depicts a very simple MapReduce operation. In that case, the inputs are squares, triangles and circles, either black or white. The colour of the shapes represents their `key` and the shape itself is the `value`. The mapping process transforms each shape into the first letter of its name, thus mapping a square to an S, a triangle to a T, and so on. The mapped values are then sorted by colour, corresponding to the `aux_aggregator` operation, and finally the reduce function is called. This function consists of counting how many T's there are. Thus, the result of the operation per key is found to be 2 white T's and 1 black T.

The MapReduce model was primarily developed to be applied onto a large set of machines linked together - also known as a *cluster* - with the purpose of drastically reducing data processing times by taking advantage of the parallel architecture of this system. Most MapReduce frameworks described in the literature [7, 25, 26, 27, 28, 29], if not all, use a master-slave architecture, similar to the one presented in Figure 2.5. Figure 2.5 depicts the flow of data in a generic MapReduce application. The data flows from left to right and is controlled by the Master. The initial data is assigned to one of the Mappers, and through computation is transformed into Intermediary data. The Master then assignes Reducers with some of the Intermediary data and after the reduce operations take place, the result is determined. Given the sometimes huge size of the clusters in which MapReduce frameworks are applied (consider the architecture in [7]), they must be highly fault-tolerant and robust. Amongst other precautions mentioned in the literature, the master node

```
map_operation(key, value) -> (key, mapped_value) {
   mapped_value = perform_map_operation(value);
}

reduce_operation(key, set_of(mapped_value)) -> (key,
   reduced_value) {
   reduced_value = perform_reduce_operation(set_of(mapped_value)
      );
}

aux_aggregator(set_of(key, mapped_value)) -> set_of(key, set_of(
   mapped_value)) {
   for each key compute
      set_of(mapped_value) = aggregate_by_key(key, set_of(key,
         mapped_value));
}
```

Figure 2.3: Pseudocode for map and reduce operations

is usually responsible for pinging the slave nodes, as well as backing up the processed data and rescheduling work in case of slave failure.

So far, the features of the MapReduce paradigm have been superficially described, but nothing has been said regarding its capability to meet real-world data processing requirements. The relevance of this model lies in the fact that the map and reduce operations are suitable for expressing a number of classic processing algorithms under a *summation* form [8]. This form allows for a direct conversion to map and reduce operations, and it has been shown by [8] that algorithms such as locally weighted linear regression, expectation maximization and neural networks, amongst others, can be applied successfully to a MapReduce framework.

Whilst these algorithms can be useful, the MapReduce model is by no means limited to them, as many possible map and reduce operations can be defined for this framework. One needs only to ensure that the operations have no collateral effects on data other than that being used in the operation. Furthermore, it is necessary to guarantee that the operations on the data are associative and commutative, so that they can be executed in parallel and thus benefit from the inherent speeding up of the process. This speed-up is a pertinent indicator to evaluate the performance of a MapReduce framework running in parallel, and in this document the following metrics for system speed-up will be adopted:

$$S_u = \frac{T_s}{T_u} \tag{2.3}$$

where

S_u is the system speed-up for u processing units. If S_u is greater than 1, the system is faster than a sequential execution.

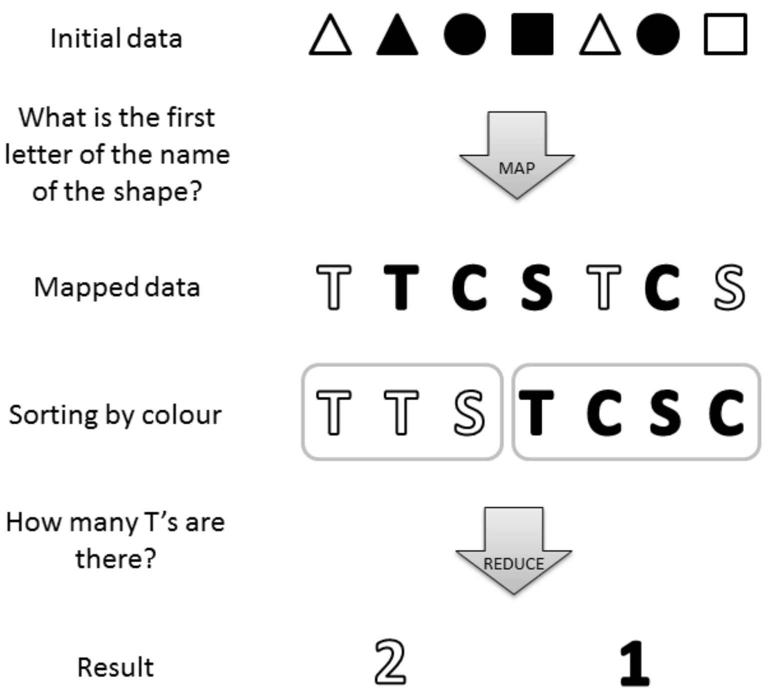


Figure 2.4: Graphic example of a MapReduce operation.

T_s is the time the system takes to run a sequential execution of the problem.

T_u is the time the system takes to run with u processing units.

The number of processing units in a system is considered to be the number of workers running simultaneously during a given call. The ideal and maximum number of processing units for a system can then be calculated as:

$$U = \sum_{m=1}^{M} \sum_{p=1}^{P_m} C_p \tag{2.4}$$

where

U is the total number of processing units in the system.

M is the number of Machines in the system.

P_m is the number of Processors in machine m.

C_p is the number of Cores in processor p.

2.2.1 MapReduce Implementations

There are presently several MapReduce implementations described in the literature [7, 25, 26, 27, 28, 29, 30], and in this document the most relevant to our work will be briefly introduced.

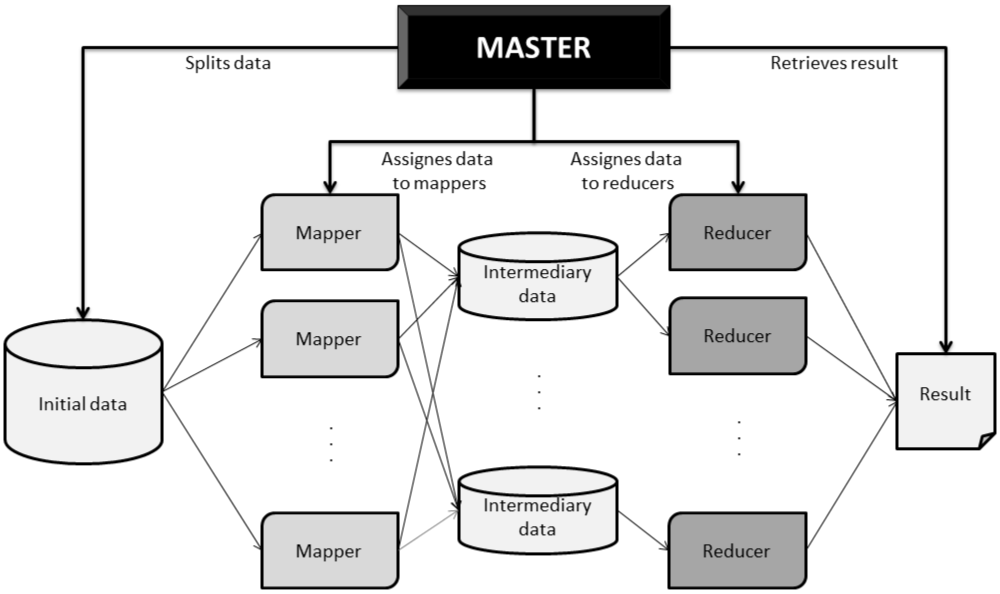


Figure 2.5: Example of MapReduce master-slave architecture

1. The HDFS or Hadoop Distributed File System [31] is a fault-tolerant distributed file system, which is designed to run on low-cost hardware. Its purpose is to meet the requirements of applications which need to manipulate large datasets and it was designed with a batch processing methodology in mind, as opposed to iterative data processing. This system uses data replication for higher reliability but also with the purpose of improving network traffic and data accessibility. In [27] Hadoop is compared to other approaches of large-scale data analysis, and whilst its setup time is negligible compared to others, the overall task processing time was found to be 3.2 times slower than the second slower approach tested (an SQL Database Management System) [27]. This highlights that there is still much work to be done if the MapReduce framework is to become a dominant approach in large-scale data-analysis.

2. Twister [29] presents an architecture different from other MapReduce frameworks since it provides efficient support for iterative MapReduce calls. Unlike most systems in the literature, it uses a publish/subscribe messaging protocol and attempts to reduce the amount of communication data to a minimum by increasing the granularity of the map operation. However, it does not feature any form of load balancing, nor is it highly fault tolerant. The only safeguards Twister implements are to back data up at the end of each iteration and to re-send work to slave nodes in case of failure. This approach presents slightly faster results than Hadoop in the situations described in [29].

3. SAGA [26] is a high level API which executes operations on distributed systems, supporting various architectures like clusters, clouds or grids. Unlike the two previous frameworks, SAGA is implemented natively in C++, as opposed to Java, and the MapReduce model was recently introduced into it. This approach is slower than most others due to its portability; the fact that it is not optimized for one distributed system only has a cost in terms of efficiency. Still, it provides a simple interface for programmers to use the MapReduce model in distributed systems with less conventional architectures.

2.2.2 MapReduce Applied to Prolog

One might wonder about the relevance of creating a MapReduce framework for Prolog, since there are already several portable and flexible implementations for other programming languages in the literature, as described in the previous sections. However, Prolog provides support for features which would be difficult to implement in functional, imperative or object-oriented languages, such as natural language analysis, machine learning and, of course, inductive logic programming. ILP is the preferred Prolog application in this work because it requires intensive and iterative processing of large amounts of data so as to infere rules applicable to it. As such, a MapReduce construct would be a valuable tool to make this process simpler and more efficient. An example of such an application is then briefly described below.

In [32], Ashwin Srinivasan *et al.* introduce an approach combining Hadoop's MapReduce framework [33] and the inductive logic programming system Aleph [34]. Their aim was to investigate whether the ILP engine could be applicable to very large datasets, seen as the amount of data available for processing has become so large as not to fit into one machine's memory. MapReduce was the selected framework for this task, due to its abstraction level and the fact that several machine learning algorithms can successfully be implemented on this model [8]. The approach used in this work consisted of two distinct engines, one for running ILP and the other for running the actual MapReduce using the Hadoop framework [31]. Two different sets of map and reduce functions were developed for this system, with different aims. The first of these sets was meant to distribute the background knowledge across the MapReduce cluster, so as to ensure that the second set of functions - which actually perform the relevant calculations for the given examples - had all the necessary clauses to be able to use a greedy algorithm. The Map Reduce and ILP engines communicate and the latter transforms examples not yet covered in MapReduce queries. When the last reduce operation is finished, the minimum cost clause determined is then returned to the ILP engine.

The authors have used both synthetic and real-world datasets, with sizes ranging from tens of thousands up to millions, and their results demonstrated that the MapReduce framework can be efficiently applied in this context. Still, the size of the dataset must be significant (greater than 500,000) in order to obtain some speed-up using this methodology. Also, the speed-ups are not nearly linear until datasets of size 5 million, and for datasets smaller than 500,000 the data processing time actually slows down when compared to sequential time due to the cost of data communication and disk access in the cluster, amongst other factors.

To the best of our knowledge, there is no MapReduce framework native to Prolog, and so the aim of this document is to describe a fast and versatile implementation of this framework in Yap. The motivation for this lies in the need for a tool for transparent distributed computing in Prolog, whose results present speed-ups even for small datasets, and whose interface would be available as predicates in a Yap library. We believe this would contribute towards more and simpler data processing support in Yap, and find it particularly relevant at an age when multi-core processors are increasingly common and inexpensive.

Chapter 3

MapReduce in Prolog

In this chapter we describe our high-level MapReduce parallel construct for Prolog and present the most relevant implementation details.

3.1 Architecture

The model's architecture is loosely based on the architecture described in [7] in the sense that it supports clusters of machines, but it innovates by taking advantage of the parallelism within each machine. Figure 3.1 shows how our framework can apply to a generic distributed architecture.

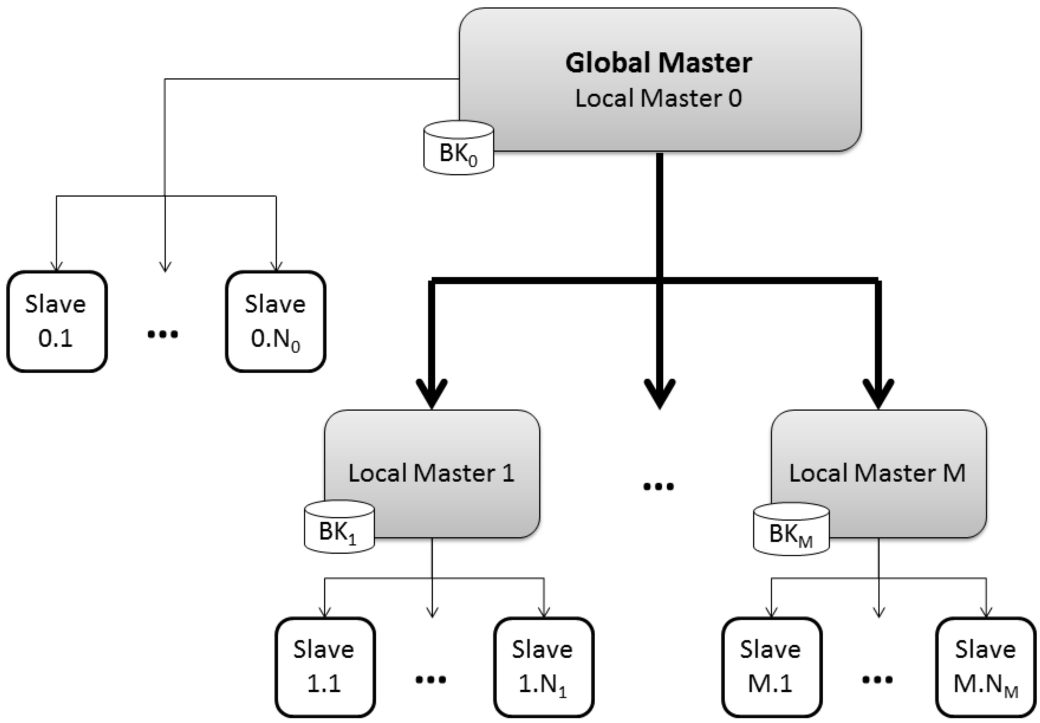


Figure 3.1: Framework architecture

There are three hierarchical levels in this architecture: the *Global Master* (GM), the *Local Masters* (LMs) and the *Slaves* (SLs). The GM controls the flow of communications and first-level scheduling, dispatching data to the LMs. There are as many LMs as machines in the cluster and each LM is responsible for local data scheduling and dispatching among the SLs running on that machine. The SLs execute both map and reduce predicates on their data and return the reduced value to the respective LM. Each LM then performs a reduce operation on all its SLs' reduced values, and similarly the GM executes the last reduce operation of the call. This architecture applies to distributed memory systems composed of multi-core machines.

For shared memory architectures (SMA), our MapReduce for Prolog uses multi-threading while for distributed memory architectures (DMA), it uses MPI [35]. In the SMA implementation, the first thread – LM0 – starts as many threads as the number of machine cores. Each thread runs a slave interface, which waits for thread messages from LM0 and carries out the work. In the DMA implementation, processes are started for each machine core or for each distributed computer node. The SLs can be thought of as resources that LMs manage according to different scheduling methods; the SLs do not keep track of how many operations they have executed, and they do not self terminate. Instead, LMs are responsible for their creation, task assignment and termination.

The system requires a set-up time, in which each LM loads any files that may have been requested by the user, so as to have the necessary information to carry out queries. This information is named *background knowledge*; in the case of different LMs, each one can have its own background knowledge. The set-up time is only spent once for each LM and each background knowledge file requested, for the SMA implementation. For the DMA implementation, files need to be read by all LMs. Since the data files are only loaded on LMs during the initialization of the program, this model allows for no communication overheads during runtime. Note that the user is responsible for having a copy of the program source code in each machine, as well as the map and reduce predicates and any other data required to complete the queries.

The MapReduce predicates are user-defined but follow a specific pre-defined signature. The map predicate has two arguments, the first being an element from the list of values to be mapped and the second the *mapped result*. The reduce predicate also has two arguments, the first being a list of Prolog terms to be reduced and the second the *reduced result*.

Each MapReduce call receives as arguments the names of predicates to be used to map and reduce data. As such, the user can specify several different predicates and use them indiscriminately in different MapReduce calls without having to re-initialize the system. The MapReduce predicate also requires a data array as input. This array can be created by the user, or it can be loaded from a file automatically. Our framework includes predicates capable of creating an array of data from a given file. The positions in the array contain the respective line of the file, in the form of a generic Prolog term. We consider this to be a flexible approach, since the user can use data from any other source he/she requires, as long as he/she makes it available to the system under this structure.

3.2 File System

One of the main goals of this implementation is to provide a flexible system, which supports both heavy computations across several machines and lighter iterative runs of MapReduce possibly executing on one machine alone. We have designed a transparent architecture divided in three functional modules as follows:

Initializer Creates a communication grid encompassing the LMs and the SLs, and loads the data for each LM.

MapReduce This module is composed of the master and the slave files. Only one of these files is used at any given time, according to the entity's hierarchical level. The slave version executes the map and reduce predicates, while the master version performs reduce operations and implements communication protocols.

Terminator Terminates the communication grid created by the Initializer and frees the allocated memory.

Additionally, user-defined files are required in order to specify the several map and reduce predicates to be used. The fact that this information is specified as Prolog predicates allows the user to easily reconfigure them – including system architecture and map and reduce predicates; it is also possible to run distinct MapReduce calls simultaneously.

3.3 Scheduling Methods

Most parallel and distributed MapReduce systems are not very concerned with the efficiency of scheduling strategies, rather with their redundancy and fault-tolerance strategies. Conversely, and since MapReduce for Prolog is an implementation for more modest computing capabilities, we concern ourselves with the speedup that this construct achieves, when compared to executing the MapReduce call sequentially. It is then crucial to have a scheduling method which allows for good performance in parallel, and bearing this in mind we developed four scheduling methods: (i) *single-step scheduling*; (ii) *static scheduling*; (iii) *dynamic scheduling* and (iv) *workpool scheduling*.

Figures 3.3, 3.4, 3.5 and 3.2 depict the interaction between LMs and SLs on each type of scheduling. This interaction can obviously extrapolate to GMs and LMs, respectively. All figures depict three stages of the scheduling algorithm, temporally from left to right, and the explanatory text is presented below.

Single-step scheduling is used as a base case. It takes the total number of items and distributes them evenly across slaves in just one step. One block of items goes to one slave, another to the second slave and so on, ensuring every SL is assigned the same number of queries, approximately. In stage two of Figure 3.2, the method of dividing data is depicted, and in stage three the division is completed.

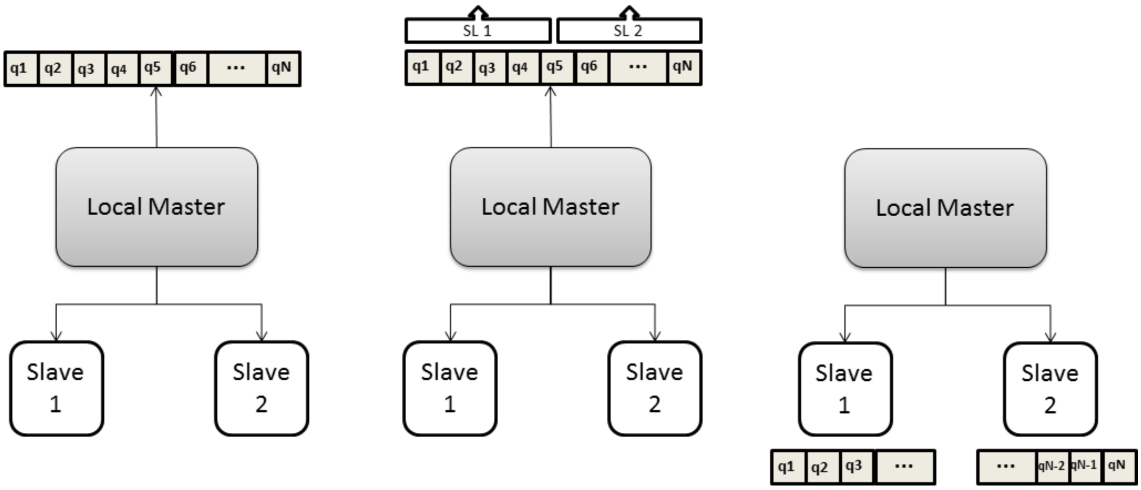


Figure 3.2: Single-step scheduling method.

Static scheduling consists of dividing the M data items in *chunks* of N elements and distributing them in a round-robin fashion by all the slaves. It differs from the single-step scheduling because the queries are distributed in several small chunks, in turns. Figure 3.3 shows that it first attributes a chunk to each slave and from then all the data is distributed alternately by the slaves.

Dynamic scheduling is more adaptive than the previous method, but also more demanding on the LM in terms of computation time. At first, it also attributes a chunk of data to each slave, in order, but then the LM waits for a reply from one of the SLs, informing that it is free. This algorithm behaves differently from static scheduling because, as shown on stage three of Figure 3.4, the LM waits for a reply from one of the SLs. The LM then attributes further work to the free SL and waits again. Ultimately, and if the data granularity is low, the dynamic scheduling converges towards static scheduling, since all SLs take the same time to complete the same number of queries.

Workpool scheduling is similar to the dynamic one, but implements a pool of work that is consumed on demand of idle slaves. As depicted in Figure 3.5, the SLs have access to a pool of work that is filled by the LM with chunks of data to be processed. The SLs remove one chunk of work when they are finished with their current one, until the pool is empty. The LM is not responsible for distributing the work between SLs, and this can be computationally less taxing on the LM entity. However, the access to the workpool is heavily competed for, and more so with a growing number of SLs.

Results and other considerations on the various scheduling methods are presented in further detail in Chapters 4 and 5, as well as some relevant future work, mentioned in Chapter 6.

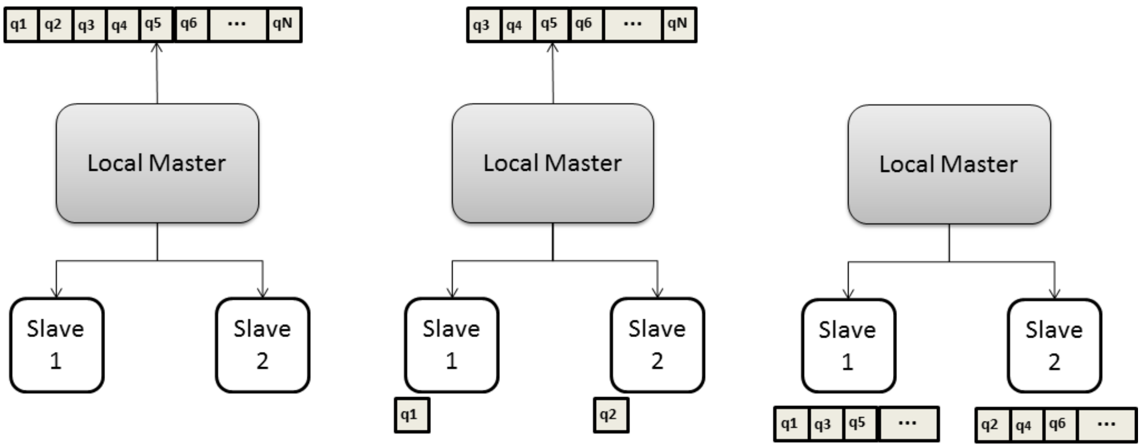


Figure 3.3: Static scheduling method.

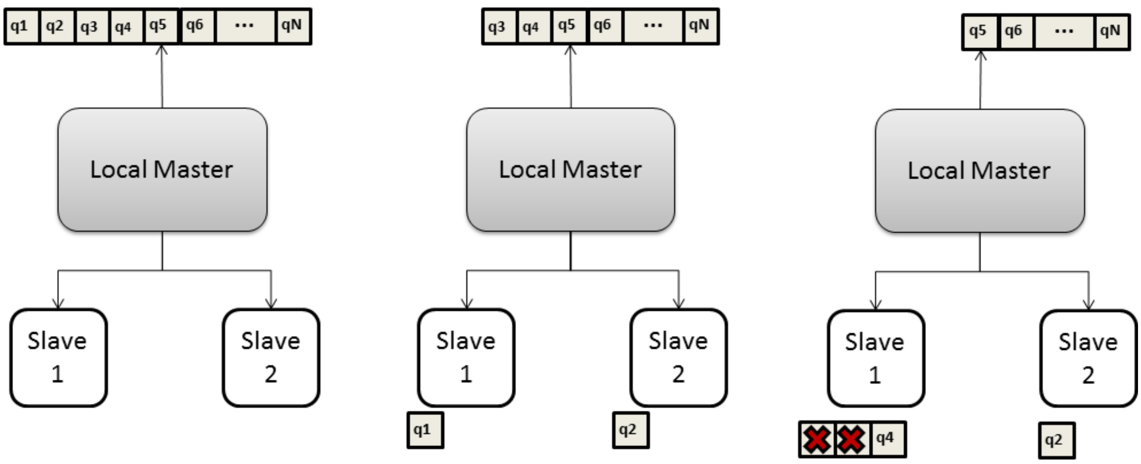


Figure 3.4: Dynamic scheduling method.

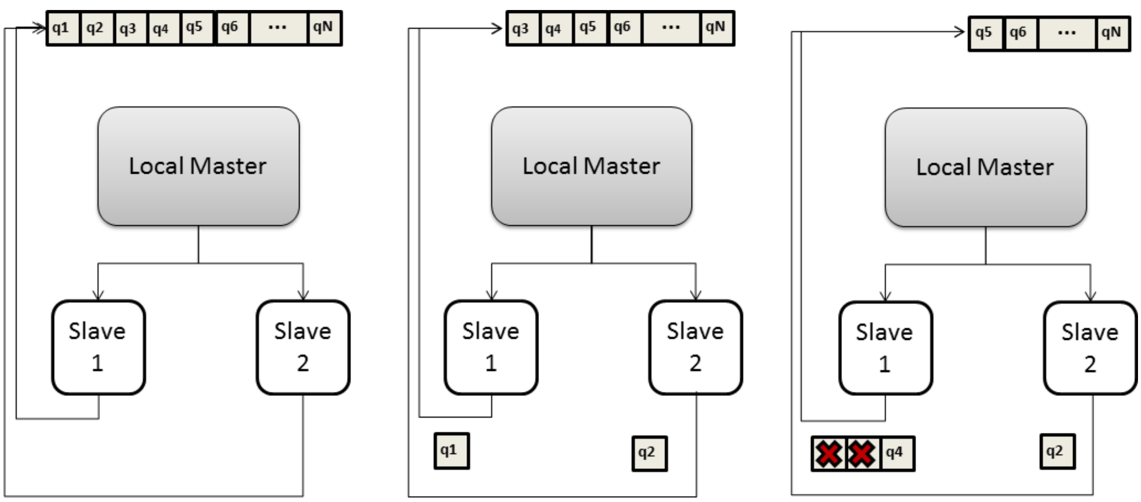


Figure 3.5: Workpool scheduling method.

3.4 User Interface

The MapReduce for Prolog user interface is composed of six predicates, as illustrated in Figure 3.6.

```
init_communicator(-Comm).
init_communicator(-Comm,+NoCores).
end_communicator(+Comm).

data_from_file(+Filename,-DataArray).

map_reduce(+Comm,+MapPred,+ReducePred,+DataArray,-Result).
map_reduce(+Comm,+MapPred,+ReducePred,+DataArray,-Result,+
    Scheduling).
map_reduce(+Comm,+MapPred,+ReducePred,+DataArray,-Result,+
    Scheduling,+NoElements).

map(+Value,-MappedValue).
reduce(+ListOfValues,-ReducedValue).
```

Figure 3.6: MapReduce for Prolog predicates in shared memory architectures.

The `init_communicator/1` and `init_communicator/2` predicates initialize the system: if no `NoCores` argument is provided, the MapReduce for Prolog determines the number of cores in the machine and starts the corresponding number of slaves. The predicate then returns the slave's information in the `Comm` argument. The `end_communicator/1` predicate should be used to terminate the communication grid and free memory.

The `data_from_file/2` predicate can be used to consult a file and load its lines, as Prolog terms, into an array. The use of this predicate is optional, since the user may build an array from other sources and pass it as argument to the *map_reduce()* call. This predicate supports three levels of customization. The most basic form – `map_reduce/5` – uses the standard scheduling options. The `map_reduce/6` and `map_reduce/7` allow the user to select a scheduling method and the number of elements per chunk for that method, if applicable. These predicates can be called iteratively and with different map and reduce operations, and they return only the final result.

Finally, the `map/2` and `reduce/2` are not part of the interface *per se*, but they are included in the description for completeness and also because even though they are user-defined, their signature must match the one in Figure 3.6. These predicates define the specific map and reduce operations and their names are passed as arguments to the `map_reduce/5` predicate. This allows for great flexibility, since the user can define several predicates prior to execution, as well as, for instance, specify different behaviours according to the machine the predicates are running in.

Due to the MPI communication protocol usage, the interface differs between shared memory and distributed memory architectures. The predicates for the distributed memory version do not contain the `Comm` argument, since the program is run as an MPI executable, meaning that the communication grid must be configured in the MPI protocol, outside the MapReduce for Prolog interface. For distributed memory systems, it is assumed that the grid has been configured and is running, and that a copy of the relevant files has been placed in every machine in the cluster. It is also not possible to change the scheduling method to workpool, since the SLs behaviour is radically different from the one exhibited in the other three scheduling methods. Other than that, the interface is very similar in both cases, and the configuration options are common to both cases. Note that the user can abstract from the details of the parallel implementation and machine architecture as we provide interfaces with different levels of transparency.

3.4.1 Usage Examples

Two usage examples are now presented in Figures 3.7 and 3.8.

```
map(V,1):-call(V),!.
map(_,0).

reduce([],0):-!.
reduce([H|T],RV):-reduce(T,Aux),RV is Aux+H.

example(Result):-
   init_communicator(8,Comm),
   data_from_file('queries.pl',MyArray),
   map_reduce(Comm,map,reduce,MyArray,Result1),
   do_something(Result1,MyArray,MyNewArray),
   map_reduce(Comm,map,reduce,MyNewArray,Result2),
   do_something(Result1,Result2,Result),
   end_communicator(Comm).
```

Figure 3.7: MapReduce for Prolog usage example for shared memory architecture

```
map(V,MV):-MV is V mod 2.

reduce([],0):-!.
reduce([H|T],RV):-reduce(T,Aux),RV is Aux+H.

example(Result):-
   data_from_file('queries.pl',MyArray),
   map_reduce(Comm, map, reduce,MyArray,Result),
   do_something(Result),
   end_communicator.
```

Figure 3.8: MapReduce for Prolog usage example for distributed memory architecture

The `map/2` predicate introduced in Figure 3.7 verifies whether a given call is true and the `reduce/2` predicate applied in this example sums all the numbers in a list, which calculates how many terms are true for `map/2`. This example is intended to be illustrative of a map operation native to Prolog, but there are many other possible applications for the simple but powerful framework we provide, such as run `map_reduce/5` calls in a loop, or define map and reduce operations so as to apply the Naïve Bayes algorithm on a dataset, as described in [8], amongst other.

In Figure 3.8, the `map/2` predicate is an example of a generic computation, and the purpose of that MapReduce call is to determine the number of odd numbers in the `queries.pl` file. This illustrates the high adaptability of MapReduce for Prolog, and its ease-of-use.

Chapter 4

Methodology

This chapter contains a thorough description of the software used to complete this work, such as the Yap System, the Intel VTune Amplifier tool and openMPI. Also, modifications to the Yap system source code and some known issues are also mentioned. Finally, the datasets used in the experiments and the respective map and reduce operations are presented.

4.1 The Yap System in Detail

Even though the MapReduce for Prolog construct was implemented *on* the Yap system (version 6.3), some research about its internal structure was made. This was necessary in order to perform some fine tuning required to improve the efficiency of MapReduce for Prolog; its initial results were not satisfactory. As such, slight modifications to the system have been made, and are described in further detail in Section 4.3. The Yap system is then depicted in Figure 4.1

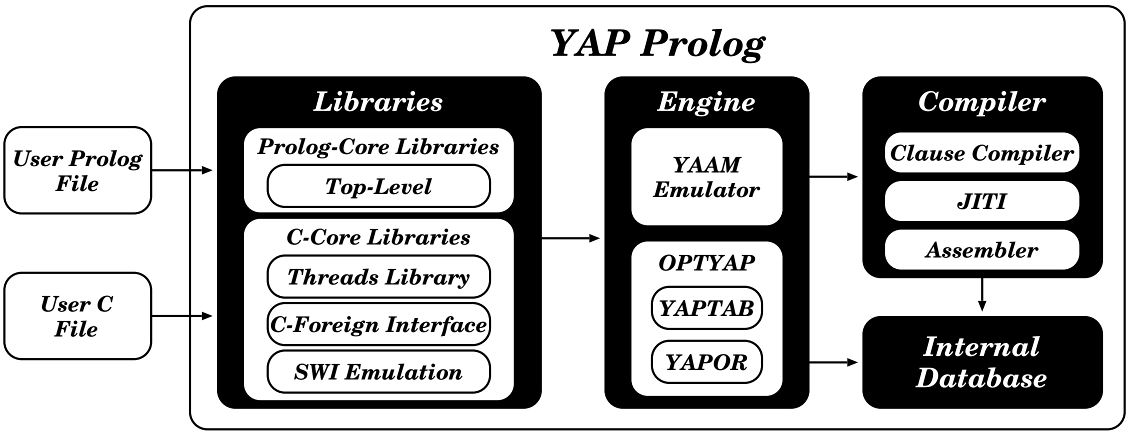


Figure 4.1: Organization of the Yap system (courtesy of Ricardo Rocha, from [1]).

In this system, there are four main data structures:

Libraries are composed of user-level Prolog libraries and core Prolog and C libraries. In this work, some changes have been made to the core libraries (see Section 4.1.1). MapReduce for Prolog's aim is to eventually integrate the user-level Prolog libraries of the Yap system.

Engine executes Yet Another Abstract Machine instructions and can use a number of strategies to improve execution.

Compiler compiles Prolog clauses and converts the data to be stored in the internal database by means of an assembler. Both the Engine and the Compiler are out of this work's scope.

Internal Database is composed of a Global and a Local memory space, as depicted by Figure 4.2. It contains Atom and Predicate Tables, which are shared between all the threads of the program, even though each thread has its own Local WAM Registers.

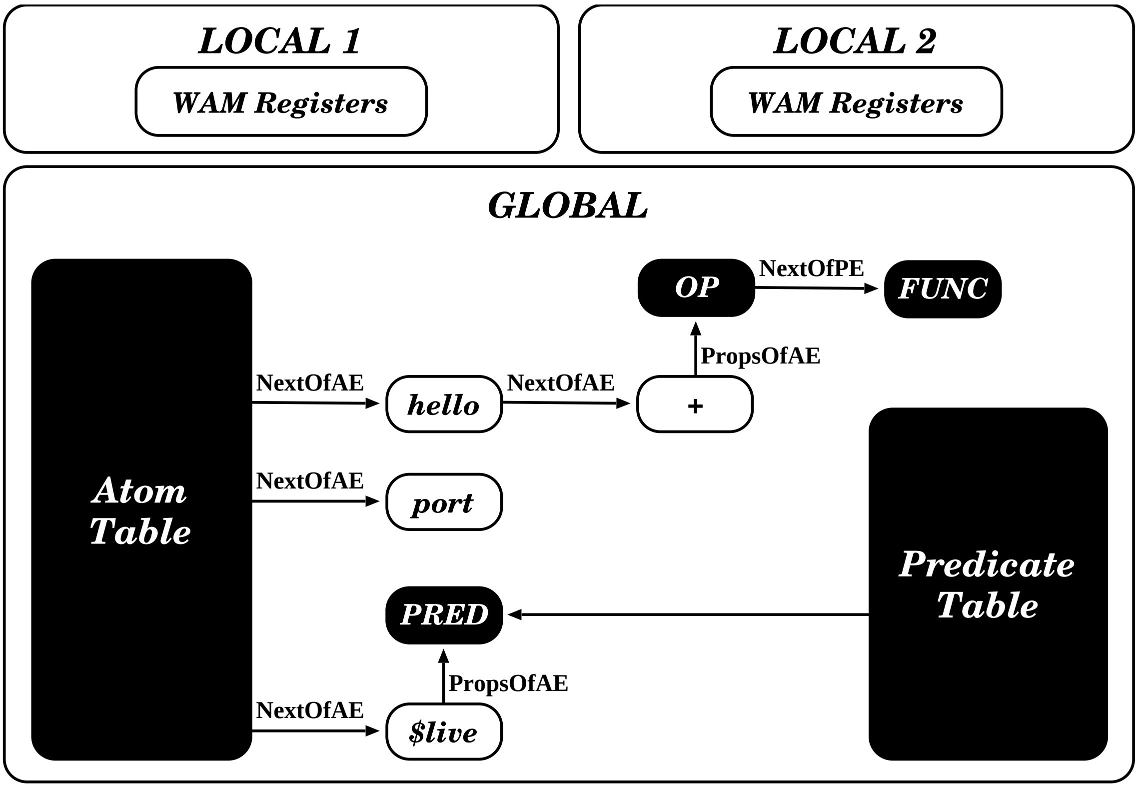


Figure 4.2: Organization of the Yap database (courtesy of Ricardo Rocha, from [1]).

Both the Atom Table and the Predicate Table are hash-based, and their entries are saved as a linked-list which contains all the atom's or the predicate's properties. Again, note that every thread saves its atoms in the joined Atom Table for Yap, and this is a factor hindering the shared memory MapReduce for Prolog's performance.

4.1.1 Yap Threads

The Yap supports standard POSIX threads, compatible with the SWI-Prolog multi-threaded library [36]. There are standard creating and termination predicates, and each thread is assigned a local memory space to save all backtrackable data. The Yap system also supports thread communication by means of thread queues. There is a queue for each thread, and they share the same name, which in this case works as an identifier of both the thread and the queue; the queue and the thread are created and destroyed in the same operation. In addition, independent queues can be created by the user, for other purposes. The queues have intrinsic associated condition variables, so as to regulate access to their data, and there are predicates to send and get messages from the queues which are signalled when new data is available on that queue.

There was some room for improvement in the current Yap implementation of the `thread_get_message/2` in terms of managing the locks efficiently, and the predicate was adjusted accordingly. Also, a non-blocking version for this predicate – `thread_get_message_non blocking/2` – was developed. Figures 4.3 and 4.4 present the implementation of the blocking and non-blocking versions of these predicates, respectively. The difference between these two implementations lies in the third clause of `thread_get_message/2` and `thread_get_message_non blocking/2`; the latter case fails when attempting to retrieve a message from an empty queue, whilst the first waits on the condition variable `Cond`. Once it is signalled, the predicate acquires the respective lock and proceeds to the message retrieval predicate `thread_get_message_loop/4`; note that a second check for messages in the queue is performed then.

```
thread_get_message(Term):-
        '$thread_self'(Id),
        thread_get_message(Id,Term).
thread_get_message(Queue, Term):-
        var(Queue),!,
        '$do_error'(instantiation_error,thread_get_message(Queue
           ,Term)).
thread_get_message(Queue,Term) :-
        recorded('$thread_alias',[Id|Queue],_),!,
        thread_get_message(Id, Term).
thread_get_message(Queue,Term):-
        recorded('$queue',q(Queue,Mutex,Cond,_,Key),_),
        '$db_is_dequeue_empty'(Key),!,
        '$cond_wait'(Cond,Mutex),
        '$lock_mutex'(Mutex),
        '$thread_get_message_loop'(Key,Term,Mutex,Cond).
thread_get_message(Queue,Term):-
        recorded('$queue',q(Queue,Mutex,Cond,_,Key),_),!,
        '$lock_mutex'(Mutex),
        '$thread_get_message_loop'(Key,Term,Mutex,Cond).
thread_get_message(Queue,Term):-
        '$do_error'(existence_error(message_queue,Queue),
           thread_get_message(Queue,Term)).
```

Figure 4.3: Blocking predicate to receive a message from a queue.

```
thread_get_message_nonblocking(Term):-
        '$thread_self'(Id),
        thread_get_message_nonblocking(Id,Term).
thread_get_message_nonblocking(Queue,Term):-
        var(Queue),!,
        '$do_error'(instantiation_error,
           thread_get_message_nonblocking(Queue,Term)).
thread_get_message_nonblocking(Queue,Term):-
        recorded('$thread_alias',[Id|Queue],_),!,
        thread_get_message_nonblocking(Id,Term).
thread_get_message_nonblocking(Queue,Term):-
        recorded('$queue',q(Queue,Mutex,Cond,_,Key),_),
        '$db_is_dequeue_empty'(Key),!,
        fail.
thread_get_message_nonblocking(Queue,Term) :-
        recorded('$queue',q(Queue,Mutex,Cond,_,Key),_),!,
        '$lock_mutex'(Mutex),
        '$thread_get_message_loop'(Key,Term,Mutex,Cond).
thread_get_message_nonblocking(Queue,Term) :-
        '$do_error'(existence_error(message_queue,Queue),
           thread_get_message_nonblocking(Queue,Term)).
```

Figure 4.4: Non-blocking predicate to receive a message from a queue.

From Figures 4.3 and 4.4 it can be inferred that there is some competitiveness between threads when retrieving a message from a queue. This proved to be an issue during implementation, and so it was decided to make use of the threads' individual message queues whenever possible, as opposed to having a shared work queue, even though the results from that approach – presented in Chapter 5 – were still good. This code can be found in 'pl/threads.yap', in the Yap source code.

4.1.2 Yap Statistics

Since speed-up is one of the considerations for the validation of MapReduce for Prolog, it was considered essential to have accurate time measuring operations. The Yap system makes available a statistics built-in predicate `statistics/2` which takes as argument a parameter such as `walltime` or `cputime` and returns its value at a given time. However, as this predicate's precision was found to be insufficient for taking the necessary measurements, two new predicates were developed and made available by Miguel Areias; they are presented in Figure 4.5.

```
statistics(thread_cputime_stime,KernelTime).
statistics(thread_cputime_utime,UserTime).
```

Figure 4.5: Time measuring predicates kindly made available by Miguel Areias.

These predicates measure time in miliseconds and are compatible with multi-threading applications. When the Yap system is started, a system time value is stored:`YapStartOfTimes`. The times returned by the `statistics/2` predicate are then measured by making system calls and finding the difference from that time. Also, some modifications were made to the `statistics/2` predicate regarding the measurement of the `walltime` parameter. These changes required altering the files 'pl/statistics.yap' in the core Prolog library, as well as the 'C/threads.c' and 'C/sysbits.c' in the core C library of Yap, and were mainly concerned with enhancing the precision of the `statistics/2` predicate. Namely, signed integers were converted to unsigned ones, since times are always positive in this scope.

4.1.3 Yap Message Passing Interface

Message Passing Interface is a communications protocol for parallel programming [37]. Its source code is available in C, C++ and Fortran, and since the MPI Forum [35] has standardized the system in 1994 and again in 1996, many hardware designers and vendors have adopted it. MPI is then a portable, highly efficient means of both broadcasting and messaging point-to-point, which can be efficiently used for MapReduce support [38, 30]. Even though it was initially designed to support distributed memory systems only, MPI2 and MPI3 have expanded scope to feature a somewhat limited thread support. The MPI Forum has made an effort to integrate the best features in various systems, and this has led to the discontinuation of some implementations, such as lamMPI [39]. However, implementations such as openMPI [40] or MPICH [41] are still maintained and supported.

The Yap system supports both lamMPI [39] and openMPI [40]. The module must be included in the Prolog code using the following command: `use_module(library(lam_mpi))`. In addition, when running Yap, the system command `mpirun` ou `mpiexec` must be invoked. A basic example of a program in Yap using MPI is depicted in Figure 4.6.

```
:-use_module(library(lam_mpi)).

example:-mpi_init,
         mpi_comm_rank(Rank),
         Rank=\=0,!,
         write('I am Slave'),write(Rank),nl,
         mpi_finalize.
example:-write('I am Master'), nl,
         mpi_finalize.
```

Figure 4.6: Example of an MPI program in Yap.

The `mpi_init/0` and `mpi_finalize/0` predicates must be invoked since this is required by the MPI protocol. In MapReduce for Prolog as well as in the example, the master is the MPI node with rank 0, and all the other processes are considered slaves. MPI in Yap also provides an interface to MPI messages, similar to the one described in Section 4.1.1 for threads. The messages require the Rank of the process, but there is a broadcast option available as well.

4.2 MapReduce for Prolog Implementation

This section elaborates on some implementation details of the MapReduce for Prolog construct presented in this document, namely the map_reduce/5 predicate. Figure 4.7 presents the lower level implementation details of that predicate. Note that all map_reduce calls are subsets of map_reduce/7.

```
map_reduce(Comm,MapPred,ReducePred,DataArray,Result):-
  Scheduling = 'dynamic',
  NoElements = 1000,
  map_reduce(Comm,MapPred,ReducePred,DataArray,Result,Scheduling
     ,NoElements).

map_reduce(Comm,MapPred,ReducePred,DataArray,Result,Scheduling)
   :-
  NoElements = 1000,
  map_reduce(Comm,MapPred,ReducePred,DataArray,Result,Scheduling
     ,NoElements).

map_reduce(Comm,MapPred,ReducePred,DataArray,Result,Scheduling,
   NoElements):-
  (
    Scheduling == 'dynamic' ->
    distribute_work_dynamic(Comm, MapPred, ReducePred, DataArray
       , NoElements, Result)
  ;
    Scheduling == 'static' ->
    distribute_work_static(Comm, MapPred, ReducePred, DataArray,
        NoElements, Result)
  ;
    error('Please use static or dynamic as the scheduling method
       .')
  ).
```

Figure 4.7: map_reduce/5 implementation details.

Figure 4.8 details the implementation of the auxiliary predicates distribute_work_static/6 and distribute_work_dynamic/6. Note that the dynamic scheduling is composed of three work dispatching stages, whilst static scheduling only sends work and receives results, thus only having two stages.

Figure 4.9 presents the algorithm used to schedule and dispatch the data to slaves, for the dynamic scheduling method, using the MPI protocol. The send_work_init_dynamic/7 predicate

```
distribute_work_dynamic(Comm, MapPred, ReducePred, DataArray,
   NoElements, Result):-
  length(DataArray, Size),
  StartPosition = 0,
  send_work_init_dynamic(Comm,MapPred,ReducePred,DataArray,
     NoElements,StartPosition, NewStartPosition),
  send_chunk_per_result_dynamic(MapPred,ReducePred,DataArray,
     NewStartPosition,NoElements,Size,FirstResults),
  get_results_dynamic(Comm,LastResults),
  append(FirstResults,LastResults,ResultList),
  call(ReducePred,ReduceList,Result).

distribute_work_static(Comm, MapPred, ReducePred, DataArray,
   NoElements, Result):-
  length(DataArray, Size),
  StartPosition = 0,
  send_work_init_static(Comm,MapPred,ReducePred,DataArray,
     NoElements, Size, StartPosition),
  get_results_static(Comm,ResultList),
  call(ReducePred,ReduceList,Result).
```

Figure 4.8: Auxiliar predicates in work distribution.

distributes a *chunk* of work to each slave, in order. Then, the `send_chunk_per_result_dynamic/7` predicate waits for a result and sends another *chunk* of work to the slave that produces the result. `send_chunk_per_result_dynamic/7` does this until it reaches the end of the array. Finally, the `get_results_dynamic/2` predicate gathers the remaining results.

The details of the static scheduling method are similar, though slightly less complex. Since the positions of an array in Yap start at 0, the `Size` argument is actually the last position of the array, or `Size - 1`. On `get_results_dynamic/2` the algorithm is not actually collecting a piece of work form each slave, it is only making sure that as many pieces as there are slaves are collected. These chunks of work correspond to the ones sent initially, and in the DMA case an auxiliary predicate to determine the number of slaves in the grid must be used both in `send_work_init_dynamic/5` and `get_results_dynamic/2`.

```
send_work_init_dynamic(0,_,_,_,_,LastPosition,LastPosition):-!.
send_work_init_dynamic(Slave,MapPred,ReducePred,DataArray,
   NoElements,StartPosition,_):-
  EndPosition is StartPosition + NoElements,
  send_to(Slave, [MapPred,ReducePred,StartPosition,EndPosition])
     ,
  NewStartPosition is EndPosition + 1,
  NewSlave is Slave - 1,
  send_work_init_dynamic(NewSlave,MapPred,ReducePred,DataArray,
     NoElements,NewStartPosition,_).

send_chunk_per_result_dynamic(MapPred,ReducePred,StartPosition,
   EndPosition,NoElements,Size,[ResultH|ResultT]):-
  EndPosition is StartPosition + NoElements,
  EndPosition > Size,!,
  receive_from(Slave,ResultH),
  send_to(Slave,[MapPred,ReducePred,StartPosition,EndPosition]),
  NewStartPosition is EndPosition + 1,
  send_chunk_per_result_dynamic(MapPred,ReducePred,
     NewStartPosition,EndPosition,NoElements,Size,ResultT).
send_chunk_per_result_dynamic(MapPred,ReducePred,StartPosition,
   EndPosition,NoElements,Size,[ResultH|[]]):-
  receive_from(Slave,ResultH),
  send_to(Slave,[MapPred,ReducePred,StartPosition,Size]).

get_results_dynamic(0,[]):-!.
get_results_dynamic(Slave,[ResultH|ResultT]):-
  receive_from(_,ResultH),
  NewSlave is Slave - 1,
  get_results_dynamic(Slave,ResultT).
```

Figure 4.9: Dynamic scheduling implementation details.

4.3 Intel VTune Amplifier

Intel VTune Amplifier [42] is a performance profiler for serial and parallel performance analysis, made available by Intel for a 30-day trial [43]. This program is meant to analyse performance of applications using Intel processor's data, and it presents numerous statistics regarding the performance of the application. Since the processors of the test machines are not Intel, this part of the MapReduce for Prolog development took place in a machine with the following characteristics:

- One Intel Core2 Quad processor, 2.83 GHz (totalling 4 cores).
- 8 GB RAM and 500 GB Sata2 Hard Drive.
- Running Ubuntu 12.10 in 64-bit mode.

Intel VTune Amplifier comes with a pre-defined set of analysis, and in this work the Hotspot and Concurrency analysis were the most used. Figure 4.10 shows the general aspect of one such analysis.

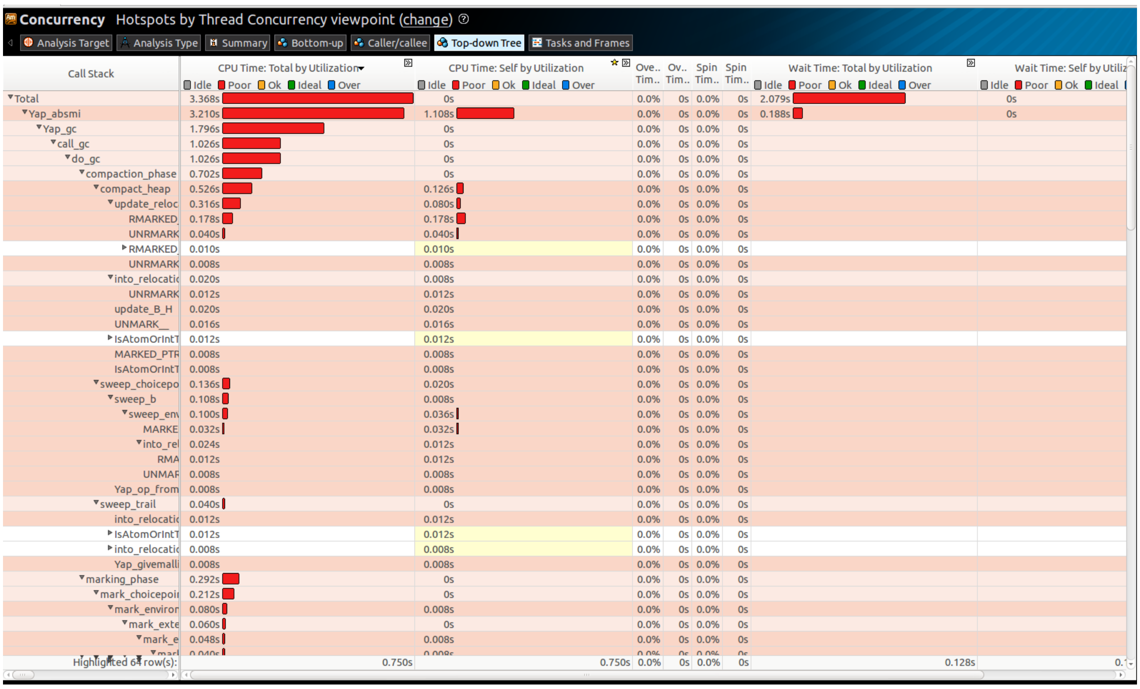


Figure 4.10: Intel VTune Amplifier Hotspot analysis.

The use of this tool was motivated by poor results in the first MapReduce for Prolog implementations. Once it was determined that the higher-level code was not responsible for the non-linear speed-ups, this option was used to track down the source of the problem. As such, several different analysis were run and proved that many more locks were being called in the program than should have been. Figure 4.11 depicts this situation.

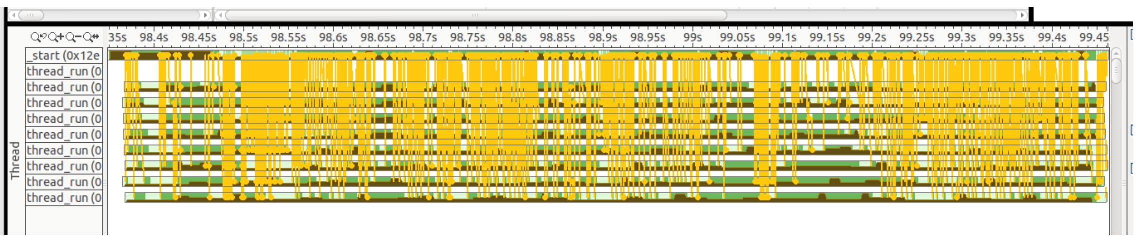


Figure 4.11: Intel VTune Amplifier lock detection.

The yellow lines represent locks, and the white spaces in the bar represent waits. Intel VTune Amplifier also allows for determining which part of the source code is causing a given lock, and so this methodology was adopted as a form to address the poor performance problem. From Intel VTune Amplifier, it was gathered that three different forms of locks were causing synchronization overheads in MapReduce for Prolog. They are as follows:

READ_LOCK is a *pthread* based lock, and it regulates access to protected structures, in read-only mode.

WRITE_LOCK is similar to READ_LOCK, but in this case the access is required to make changes in the protected data structures. Both these locks are used when accessing or creating entries in the Atom or Predicate tables.

PELOCK is a lock implemented by the Yap system and it is associated to the process of initializing Yap and to the data indexing on startup.

In order to resolve this situation, modifications were made to the files 'C/adtdefs.c', 'H/Yatom.h' and 'C/absmi.c', in the Yap core libraries. These modifications consisted only of commenting sections of outdated code, where possible, and they produced the desired result. However, there is one locking situation that could not be resolved until the present date. The locks regulating access to the Atom Table are inefficient and cripple performance in applications that require intensive access to it. This is the case of the BLOG dataset, which will be described in more detail in Sec 4.5 and whose results are presented and commented on Chapter 5.

4.4 Materials

Our testing environment consisted of two shared memory machines, used both independently and as a cluster. Their technical specifications are the same:

- Four six-core AMD Opteron 8425 processors, 2.1 GHz (totalling 24 cores).
- 64 GB RAM and 1.5 TB Hard Drive.
- Running Red Hat Enterprise Linux in 64-bit mode.

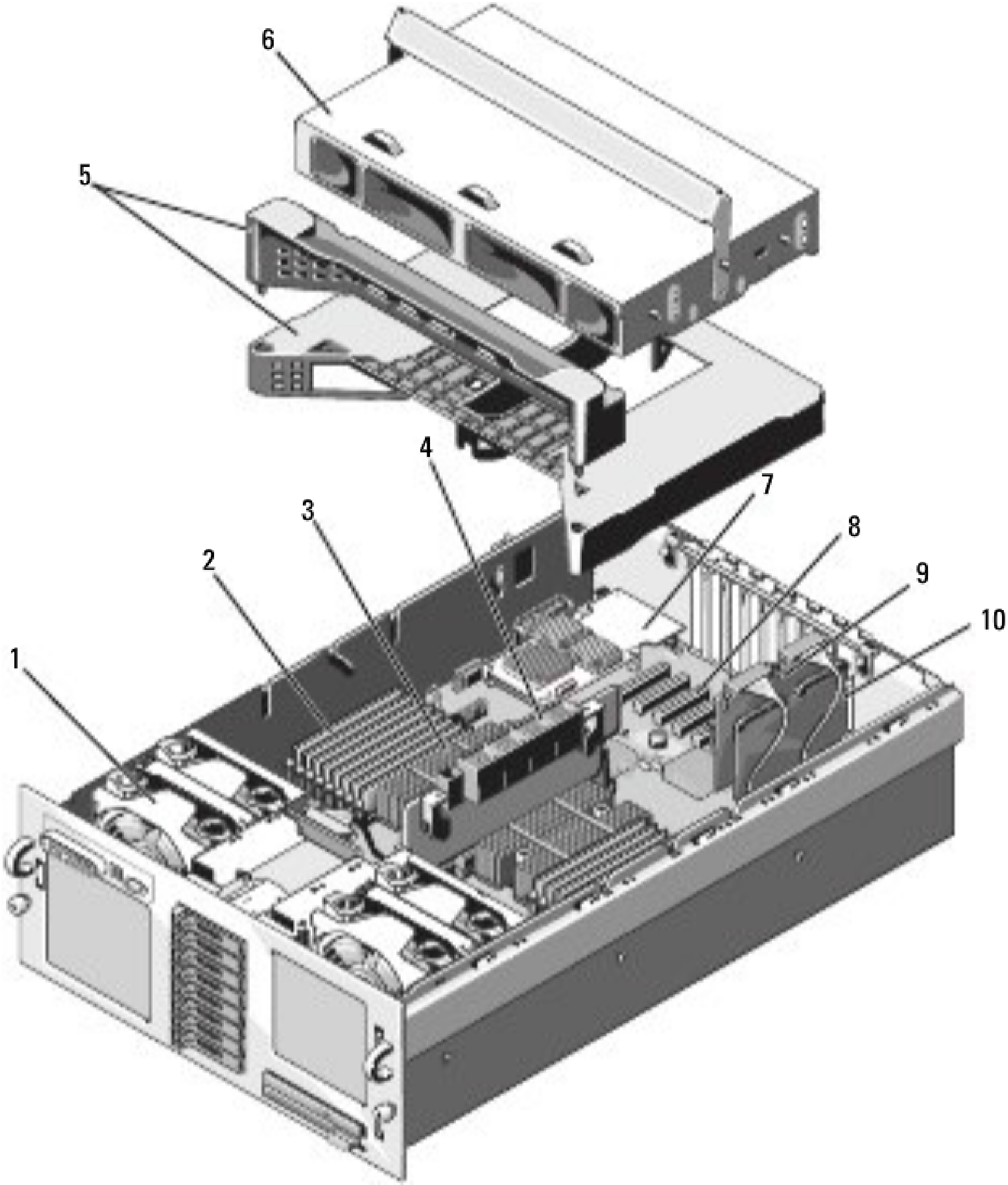
Figures 4.12, 4.13 and 4.14 show the physical location where these machines are running, as well as a view of the front panel.

The four processors in each machine are mounted as depicted in Figure 4.15. These machines are set up on a Dell™ PowerEdge™ R905 motherboard [2] each. Each group of two processors in one machine shares cache memory registers, and there is a high-speed connection module named riser board (item 4 in Figure 4.15) joining groups of two processors in the motherboard (above and below). Both machines use the processor expansion module (item 6 in Figure 4.15), so as to incorporate four processors on the same motherboard.

Figure 4.12: Machines' storage facility, located in DCC.

Figure 4.13: Machines' front view.

Figure 4.14: Machines' front view - detailed.



1	fan modules (4)	2	memory modules (16)
3	heatsink/processor (2)	4	riser board
5	cooling shroud assembly	6	processor expansion module
7	NIC daughter card	8	expansion card slots (7)
9	SAS controller card	10	optional RAC

Figure 4.15: Dell™ PowerEdge™ R905 Architecture, from [2]

4.5 Datasets

Four datasets of different characteristics were selected to validate the MapReduce for Prolog implementation. Two of them are composed of data native to Prolog, as well as background knowledge files (data files specified by the user) which must be consulted during execution. The other two consist of integers, and simple operations are performed on them. Table 4.1 summarises this information.

Table 4.1: Data type and background knowledge file size

Dataset	Data type	Background knowledge size
ODD	Arithmetic	–
PROB	Probabilistic	–
MAMMO	Prolog facts	91.2 MB
BLOG	Prolog facts	1.5 GB

We next describe the map and reduce operations applied to these datasets:

ODD the map operation verifies whether a number is odd and the reduce operation counts how many odd numbers there are in the dataset. Code implementing these operations can be found in Figure 4.16.

PROB the map operation assigns a partition of the probabilistic space to an occurrence and the reduce operation counts the total number of occurrences in each partition. This can be used to calculate conditional probabilities so as to implement a step of a Bayesian network, for instance. The map and reduce operations for this dataset can be found in Figure 4.17

MAMMO and BLOG the map and reduce operations applied to these datasets are similar and are reported in Figure 4.18 and Figure 4.19, respectively. The map operation verifies whether a term is true, based on rules specified in the background knowledge files (which differ according to the dataset) and the reduce operation counts how many terms were covered by that rule.

```
map(Number,Rest):-
  Rest is Number mod 2.
map(_,0).

reduce([],0):-!.
reduce([H|Xs],Out1):-reduce(Xs,Out),
  Out1 is Out+H.
```

Figure 4.16: Map and reduce operations for dataset ODD.

```
map(Term,[CWillow,CMissing,CAspen]):-
  arg(1,Term,Elevation),Elevation > 2600,
  arg(2,Term,Aspect),Aspect > 90,
  arg(3,Term,Slope),Slope > 5,
  arg(4,Term,HzDist),HzDist > 1200,
  arg(5,Term,VtDist),VtDist > 230,
  arg(11,Term,WilArea),WilArea = 0,
  arg(12,Term,SoilType),SoilType = 0,
  !,
  arg(55,Term,Class),
  (Class = 4 ->
    (CWillow = 1, CMissing = 0, CAspen = 0)
  ;
    (CWillow = 0, CMissing = 0, CAspen = 1)
  ).
map(_Term,[0,1,0]).

reduce([],[0,0,0]).
reduce([[CWillow,CMissing,CAspen]|Tail],[W1,M1,A1]):-
  reduce(Tail,[W,M,A]),
  W1 is W+CWillow,
  M1 is M+CMissing,
  A1 is A+CAspen.
```

Figure 4.17: Map and reduce operations for dataset PROB.

Tests were run for both the shared memory and the distributed memory implementations, across the two machines in the cluster, using different numbers of queries (300,000, 600,000 or 1,200,000 queries were posed for each test). We also performed experiments with the four different scheduling strategies for a fixed number of queries (dataset size) and fixed number of items sent to each slave (chunk size). Experiments varying the dataset and chunk sizes were performed for 1, 2, 4, 8, 16 and 24 slaves.

```
map(Term, 1) :-
  is_malignant(Term),!.
map(_,0).

reduce([],0):-!.
reduce([H|Xs],Out1):-reduce(Xs,Out),
  Out1 is Out+H.

is_malignant(A):-
  same_study(A,B),
  'HO_BreastCA'(B,hxDCorLC),
  'MassPAO'(B,present),
  'ArchDistortion'(A,notPresent),
  'Sp_AsymmetricDensity'(A,notPresent),
  'Calc_Round'(A,notPresent),
  'SkinRetraction'(B,notPresent),
  'Calc_Popcorn'(A,notPresent),
  'FH_DCNOS'(B,none).
```

Figure 4.18: Map and reduce operations for dataset MAMMO.

```
map(Term,1):-
  item(Term),!.
map(_,0).

reduce([],0):-!.
reduce([H|Xs],Out1):-reduce(Xs,Out),
  Out1 is Out+H.

item(A):-
  blogname(A,energetica),
  tk2(B,A,C,'VER:infi',fare),
  tk2(D,A,E,'VER:pres',potere).

tk2(A,B,C,D,E) :-
  (var(E) ->
    tk(A,B,C,D,E),
    D \= 'ADV',
    D \= 'PON',
    D \= 'CON',
    D \= 'PRE',
    D \= 'PRE:det',
    D \= 'DET:def',
    D \= 'DET:indef',
    E \= '<unknown>',
    E \= '@card@'
  ;
    tk(A,B,C,D,E)
  ).
```

Figure 4.19: Map and reduce operations for dataset BLOG.

4.6 Known Issues

This section enumerates the known issues regarding the MapReduce for Prolog implementation. Note that most of these are mentioned again in Chapter 6, under future work.

- At this point, the Yap system does not yet support MPI protocol in multi-threaded applications, and this renders the use of a two-level scheduling method impossible at present.
- There are issues concerning reading and writing from files using the Yap system I/O interface. When two threads or processes attempt to open the same file simultaneously, an error occurs and they fail silently.
- As mentioned earlier, there is a syncronization point when accessing the Atom Table in Yap, and this effect is more evident in applications which use words or strings frequently.
- It is presently necessary to run the `mpi_init` program before Yap runtime. This can be done by executing the command `-z "mpi_init"` after calling yap in the command line, for instance.
- The usage of different background knowledge files in each machine may be difficult, as MapReduce for Prolog does not provide predicates to split the data itself. This implies the user must split the files according to the needs of each slave, and that may not be trivial.

Chapter 5

Results

This chapter analyses the data obtained from MapReduce for Prolog thorough testing and presents several plots, regarding the scheduling methods, the load balancing and the effect of varying the number of elements per *chunk*. An effort is made throughout this chapter to maintain consistency between notation and line colours on the plots. Finally, there is a discussion on both quantitative and qualitative result aspects.

5.1 Initial Measurements

This section is concerned with measurements that are a basis for further testing, such as the time for data loading and the sequential execution times for each dataset. Tests were performed using the four datasets mentioned in Chapter 4, a varying number of queries for each (300,000, 600,000 or 1,200,000), different scheduling methods as described in Chap 3, and different *chunk* sizes for those scheduling methods, when applicable.

Sections 5.1.1 and 5.1.2 below present some relevant data regarding measurements that are used across all this chapter.

5.1.1 Loading Data Files

Table 5.1 contains the set-up time spent loading the queries files and the background knowledge, when applicable, for each dataset and query number. This time is only spent on the first run of the MapReduce for Prolog and it was recorded in seconds. In shared memory, the time of thread creation and termination is not taken into account, since it is negligible. For distributed memory, the termination time is also negligible. Note that the set-up time for distributed memory is highly dependent on the number of running slaves and on the machines' hard drive: if the files being loaded are shared between several processes, the set-up time could be slightly increased.

Table 5.1: Set-up times (in seconds) for varying dataset sizes

Dataset	300,000	600,000	1,200,000
ODD	2.4	4.2	7.8
PROB	24.0	47.5	95.5
MAMMO	30.2	34.0	41.8
BLOG	377.5	381.9	387.1

5.1.2 Sequential Execution Times

Tables 5.2 and 5.3 show the overall time (*walltime*), in milliseconds, of a MapReduce call for each dataset. Note that the corresponding times between SMA and DMA vary significantly. This can be justified by the fact that MPI runs processes (and not threads), which are managed at kernel level, and thus more efficiently. In addition, MapReduce for Prolog is implemented using the Yap Prolog system, which is not yet finely tuned for thread support. In particular, Yap's sequential version run in the MPI implementation uses simpler data structures and does not share global data structures; thus their manipulation becomes simpler and faster.

Table 5.2: Sequential execution times (in milliseconds) for SMA and varying dataset sizes

Dataset	300,000	600,000	1,200,000
ODD	240	485	956
PROB	479	968	2,016
MAMMO	1,238	2,194	4,623
BLOG	824	1,872	3,783

Table 5.3: Sequential execution times (in milliseconds) for DMA and varying dataset sizes

Dataset	300,000	600,000	1,200,000
ODD	226	453	905
PROB	376	733	1,447
MAMMO	707	1,413	2,829
BLOG	573	1,148	2,294

5.2 Scheduling Methods Evaluation

This section thoroughly tests and assesses the MapReduce for Prolog scheduling methods. Firstly, there is a comparison of the seven different scheduling possibilities, followed by an analysis of the load balancing for each case. In addition, an assessment of the variation of the *chunk* size, when applicable, is also made available. Finally, a qualitative discussion is presented and some comments on performance are included. All the relevant raw data to aid in this section's analysis is included in Appendices A, B and C; these data include all the walltimes used to calculate the speed-ups presented below, as well as a full account of each slave's time for load balancing assessment and the effect of *chunk* size variation.

5.2.1 Varying Scheduling Strategies

Figures 5.1, 5.2, 5.3 and 5.4 plot the seven scheduling methods made available by MapReduce for Prolog for each dataset. The results presented here do not take into account the set-up times described in Table 5.1. The aim of these plots is to demonstrate the variation of the performance of the scheduling methods according to the type of data and also with the implementation used. The data used to plot these graphs was obtained by running five trials of each MapReduce call and calculating their average. Finally, the data from dataset BLOG is incomplete in the distributed memory instances because memory constraints did not allow for running sixteen instances of this application on the cluster. Note that the colours are fixed for each scheduling method and that shared memory instances are marked with a cross, whilst distributed memory ones with a dot. For further data refer to Appendix A.

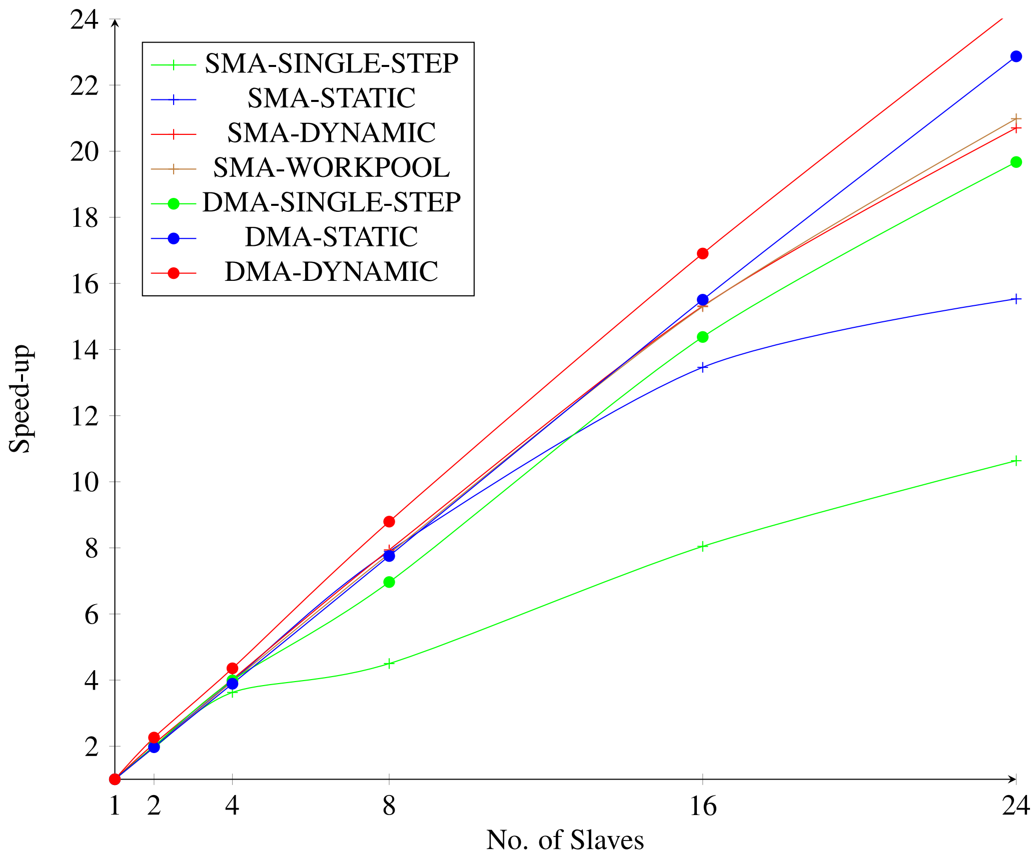


Figure 5.1: Comparison of scheduling methods for ODD dataset (600,000 queries and 1,000 elements per chunk)

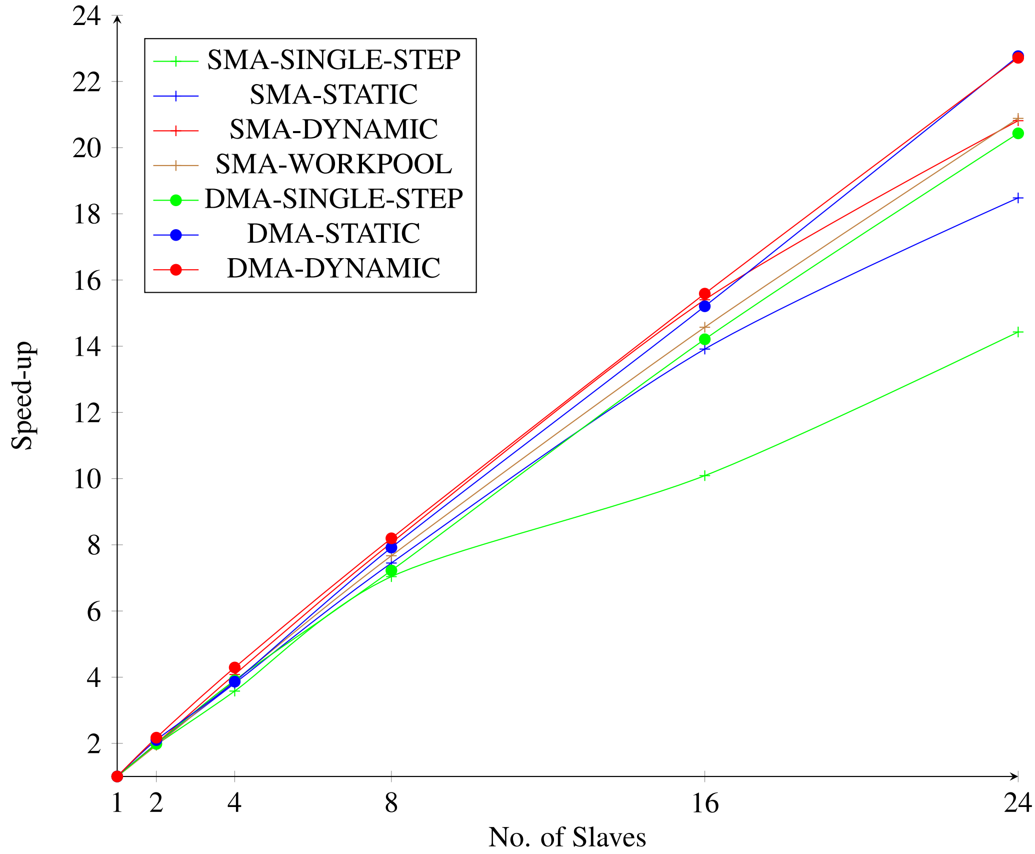


Figure 5.2: Comparison of scheduling methods for PROB dataset (600,000 queries and 1,000 elements per chunk)

These results show that MapReduce for Prolog achieves nearly linear speed-ups, for both shared and distributed memory, and for all the different datasets tested. The distributed memory implementation has proved to be consistently faster than the shared memory one. This is to be expected since MPI runs processes and Yap is not yet finely tuned for thread support. In fact, this could explain the somewhat under achieving results for the dataset BLOG in shared memory. The BLOG dataset requires intensive use of the Yap atom table, whose synchronization is centralized. Since this table is shared between all slaves in a process, it can cause a significant overhead.

From Figures 5.1, 5.2, 5.3 and 5.4, we can also observe that globally the most efficient scheduling methods are the workpool (SMA-POOL) and the dynamic scheduling (SMA-DYNAMIC or DMA-DYNAMIC). If the data's granularity was negligible, the dynamic algorithm would tend to static scheduling, with slightly worse performance due to the small wait caused by the master only sending work when the slave is already free. In the workpool strategy, the slaves are responsible for their own work management, thus making it even more efficient than the dynamic scheduling. However, and to ensure compatibility between both MapReduce for Prolog versions, we will adopt the dynamic scheduling method as the default strategy, since it displays the best behaviour for distributed memory and a close second for shared memory.

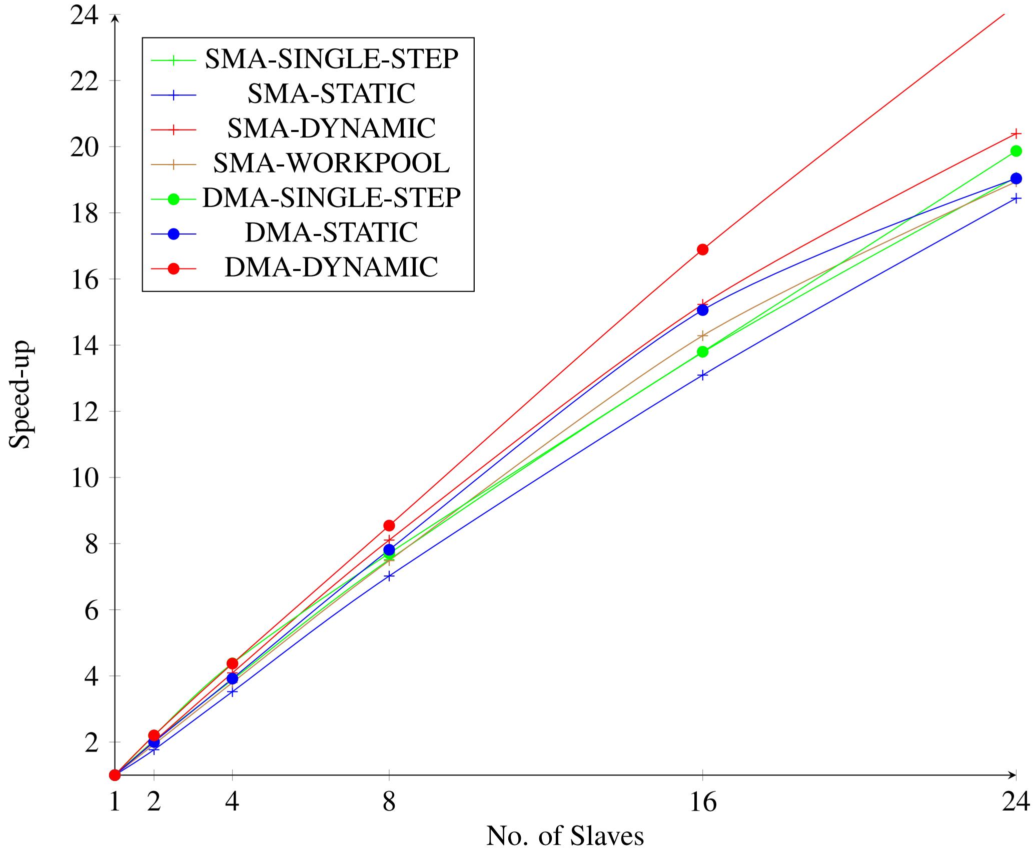


Figure 5.3: Comparison of scheduling methods for MAMMO dataset (600,000 queries and 1,000 elements per chunk)

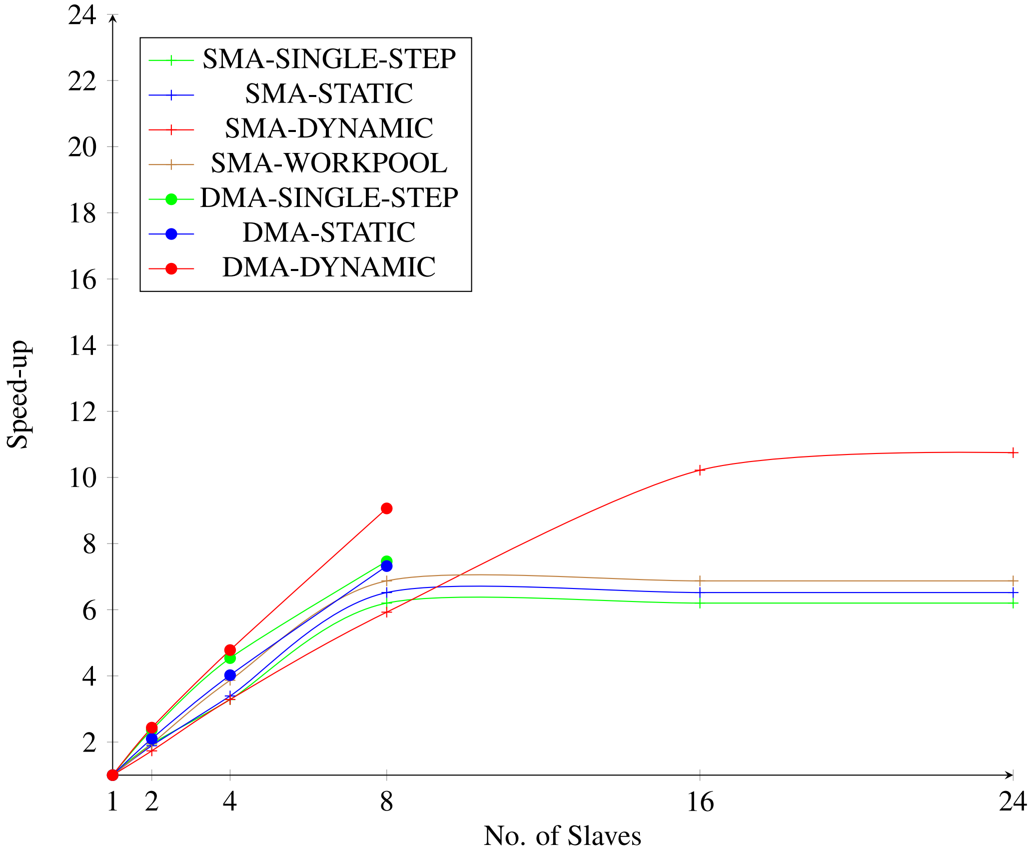


Figure 5.4: Comparison of scheduling methods for BLOG dataset (600,000 queries and 1,000 elements per chunk)

5.2.2 Load Balancing

In order to assess load balancing in the different scheduling methods, the CPU time of each slave was measured and plotted in Figure 5.5. This test was run for 1.2 million queries and for sixteen slaves, with the exception of DMA-BLOG, in which case it was only possible to use eight slaves due to memory constraints. The y-axis of Figure 5.5 denotes the maximum deviation between slaves, as a percentage of the average walltime of the respective run. As before, each MapReduce call was run 5 times and all values presented are calculated from the averages of those runs. For further data refer to Appendix B.

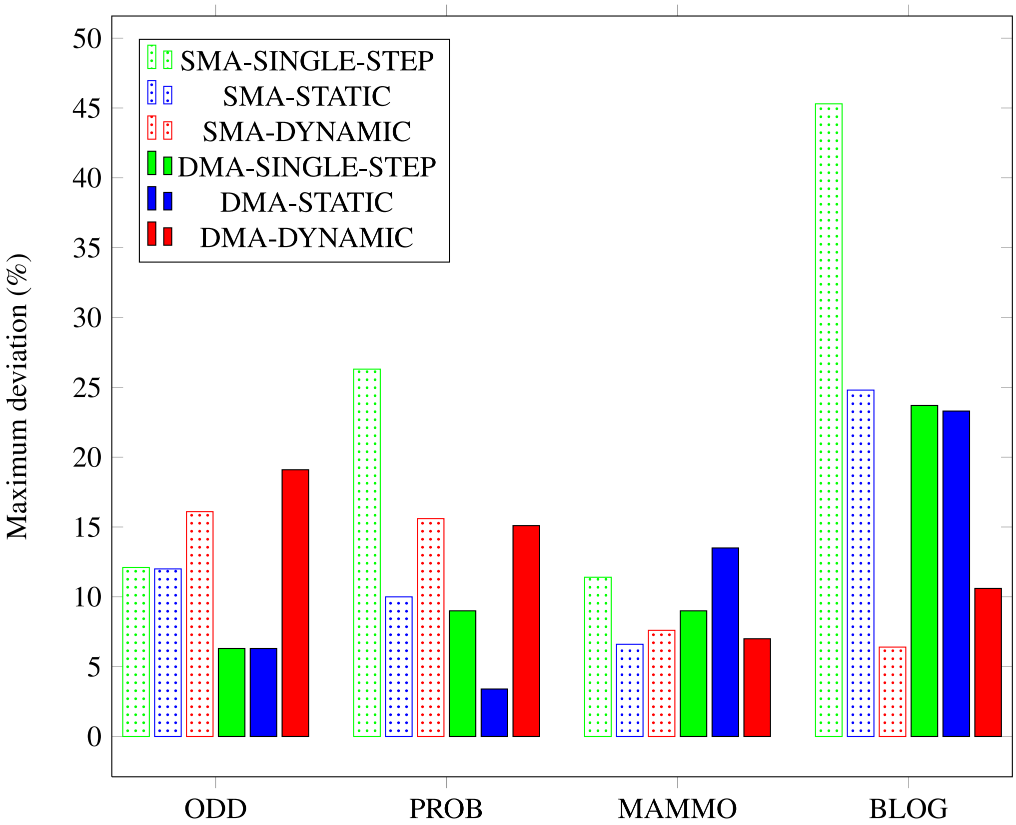


Figure 5.5: Load balancing for different scheduling methods (1,200,000 queries and 1,000 elements per chunk)

From Figure 5.5 it becomes evident that static scheduling is generally more efficient for datasets PROB and ODD and dynamic scheduling for datasets MAMMO and PROB. This is caused by the data granularity of the datasets native to Prolog; queries can take variable times to succeed or fail, which can contribute to load imbalance. The fact that the SMA is consistently slower than DMA, and more so for single-step scheduling, can be justified by the fact that the communication between threads is slower than between MPI nodes due to synchronization issues in the Yap Prolog system; this would cause a significant detachment between the reception of the first data in each slave. This effect becomes more evident when the slaves are only processing a large block of data, at once.

5.2.3 Varying Chunk Sizes

Figure 5.6 and 5.7 depicts the effect of varying the size of the chunks in the two best performing scheduling methods. The time is given in milliseconds and it is an average of five consecutive and equal MapReduce runs. For further data refer to Appendix C.

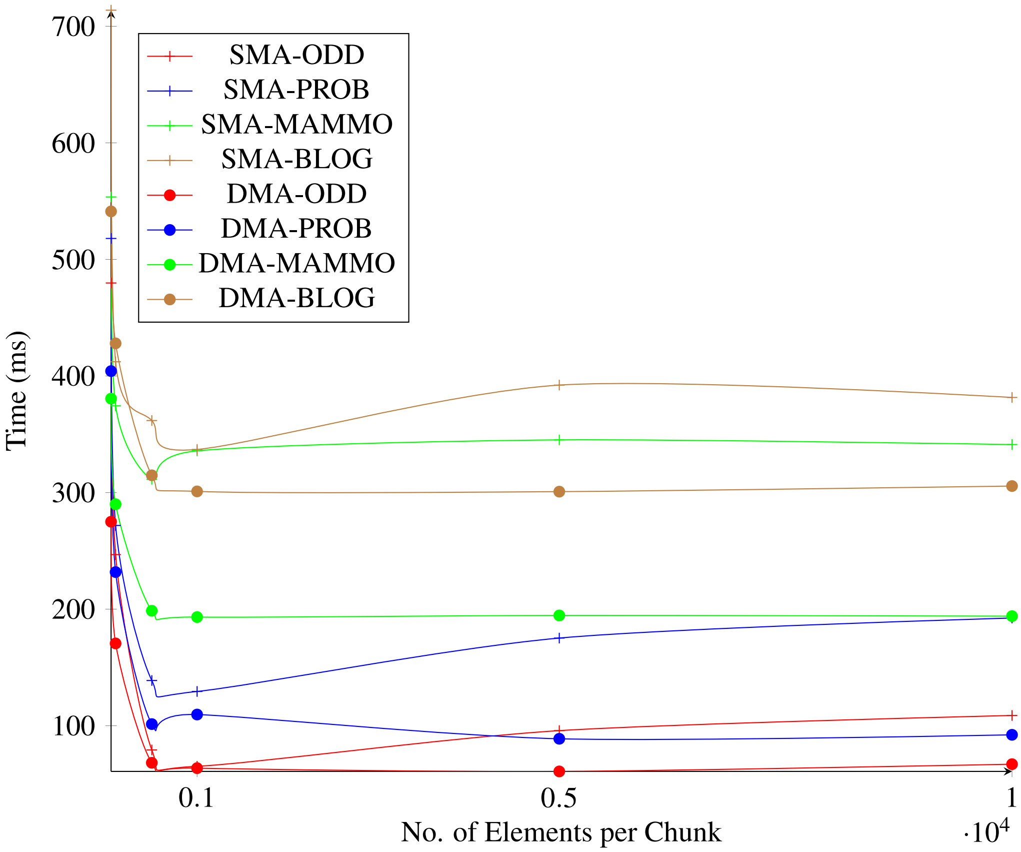


Figure 5.6: Effect of chunk size variation in dynamic scheduling (1,200,000 queries)

For all four datasets used for testing, there appears to be an optimum number of queries to minimize execution time. In our methodology, when testing scheduling methods using chunks, we have used queries of size 1,000 precisely to obtain the fastest result possible when assessing other parameters. 1,000 elements per chunks is a somewhat empirical choice, however, because even though the curves all demonstrate a tendency towards a minimum around that point, it would require testing every single value to ensure that 1,000 is in fact the best choice.

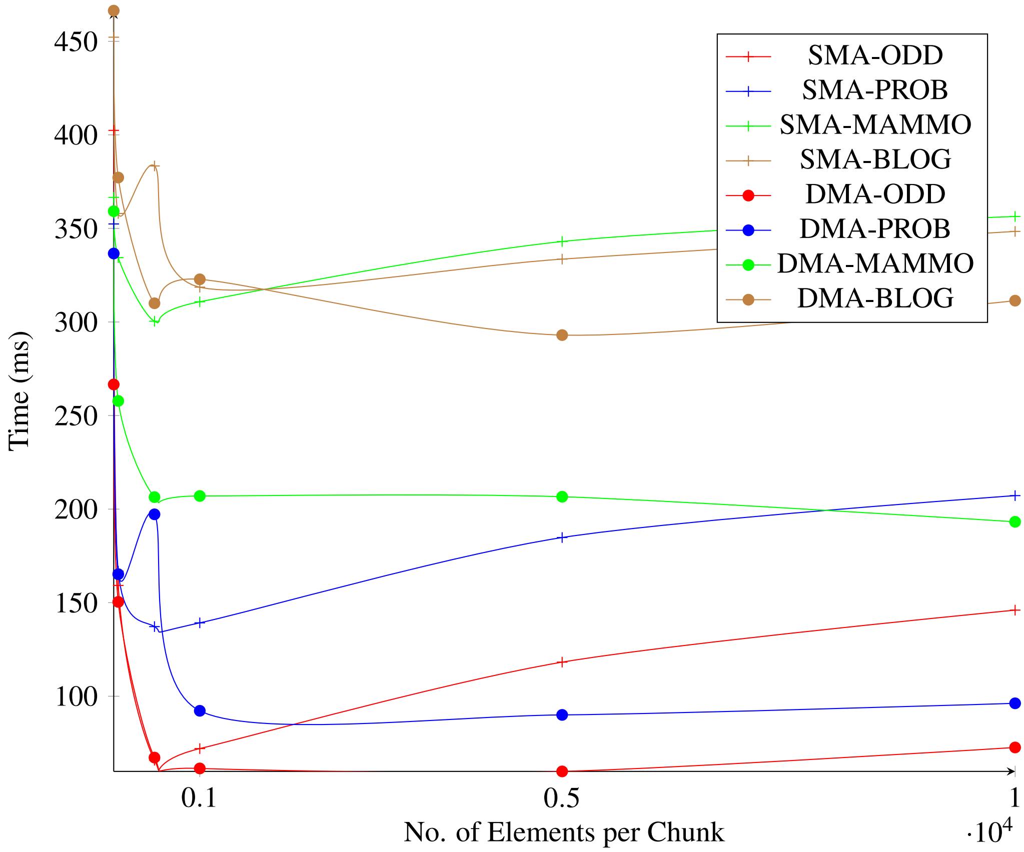


Figure 5.7: Effect of chunk size variation in static scheduling (1,200,000 queries)

5.3 Varying Data Sizes

This section introduces the speed-up plots for varying data sizes and for dynamic and static scheduling, since these methods were found to be the best performing ones in the previous section. Again, consistency is maintained in the plots below by fixing colours for the same size and using different markers for shared and distributed memory. For further data refer to Appendix A.

Figures 5.8, 5.9, 5.10 and 5.11 depict the behaviour of dynamic scheduling, for each dataset, with varying queries size and 1,000 elements per chunk.

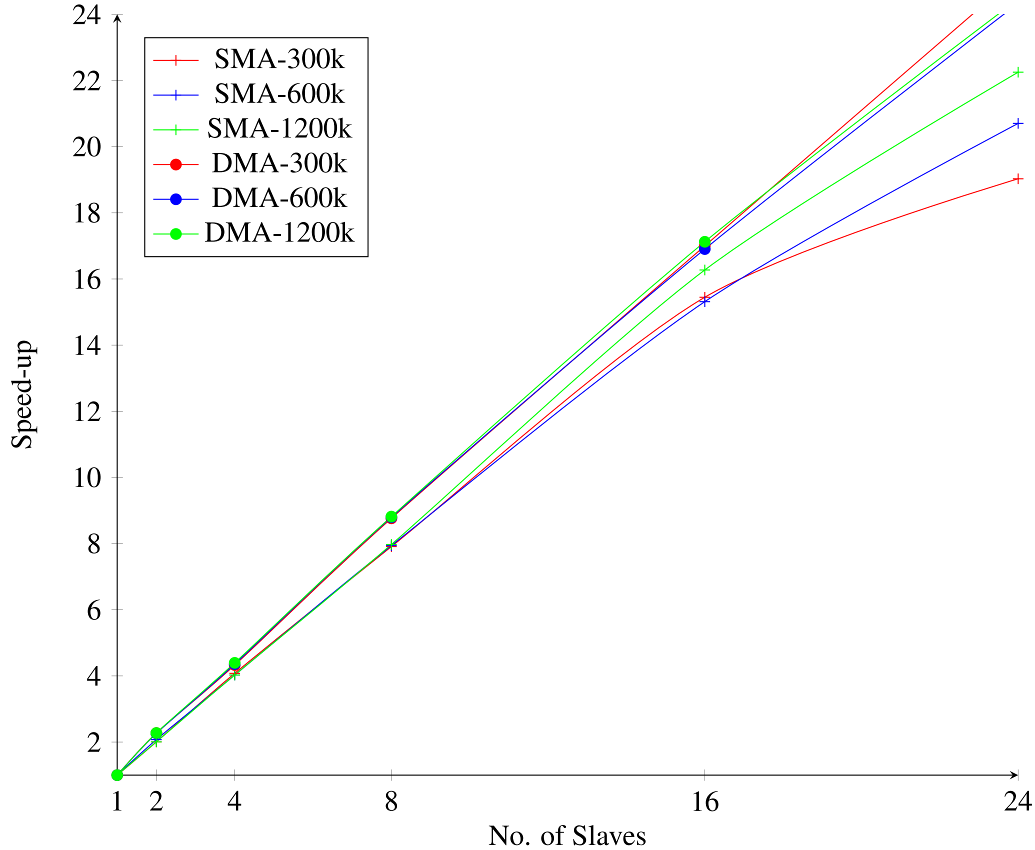


Figure 5.8: Effect of variation of queries size with dynamic scheduling in ODD dataset (1,000 elements per chunk)

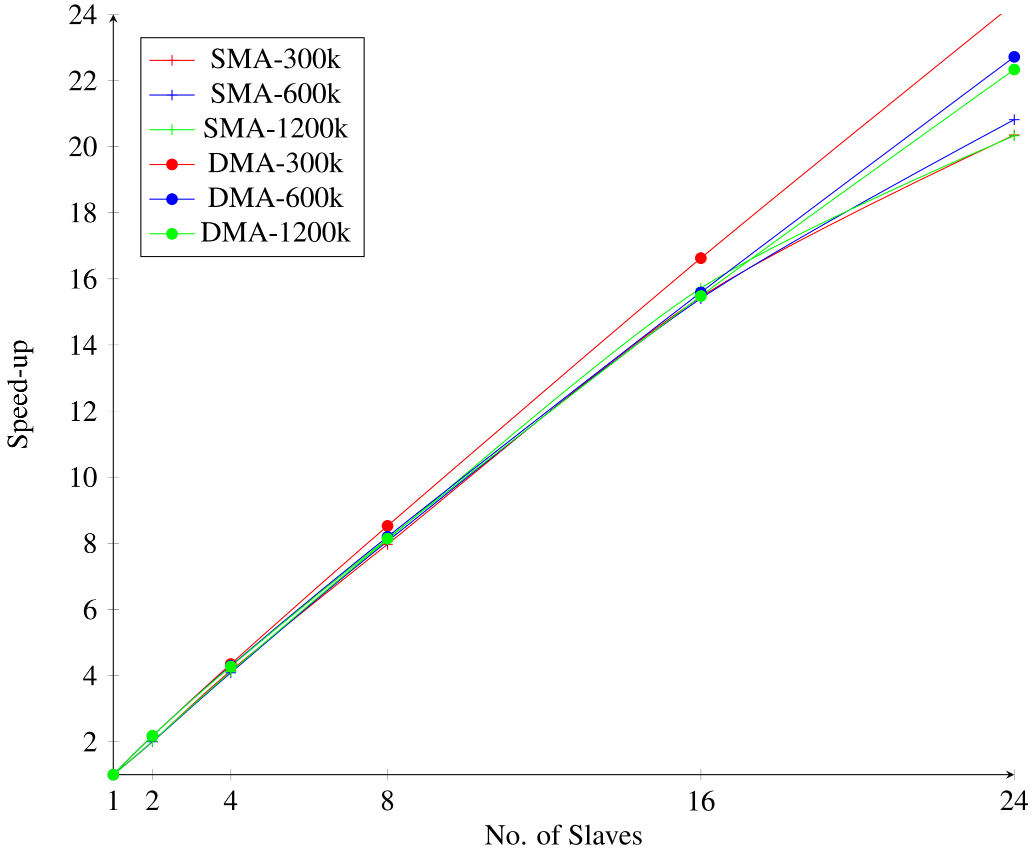


Figure 5.9: Effect of variation of queries size with dynamic scheduling in PROB dataset (1,000 elements per chunk)

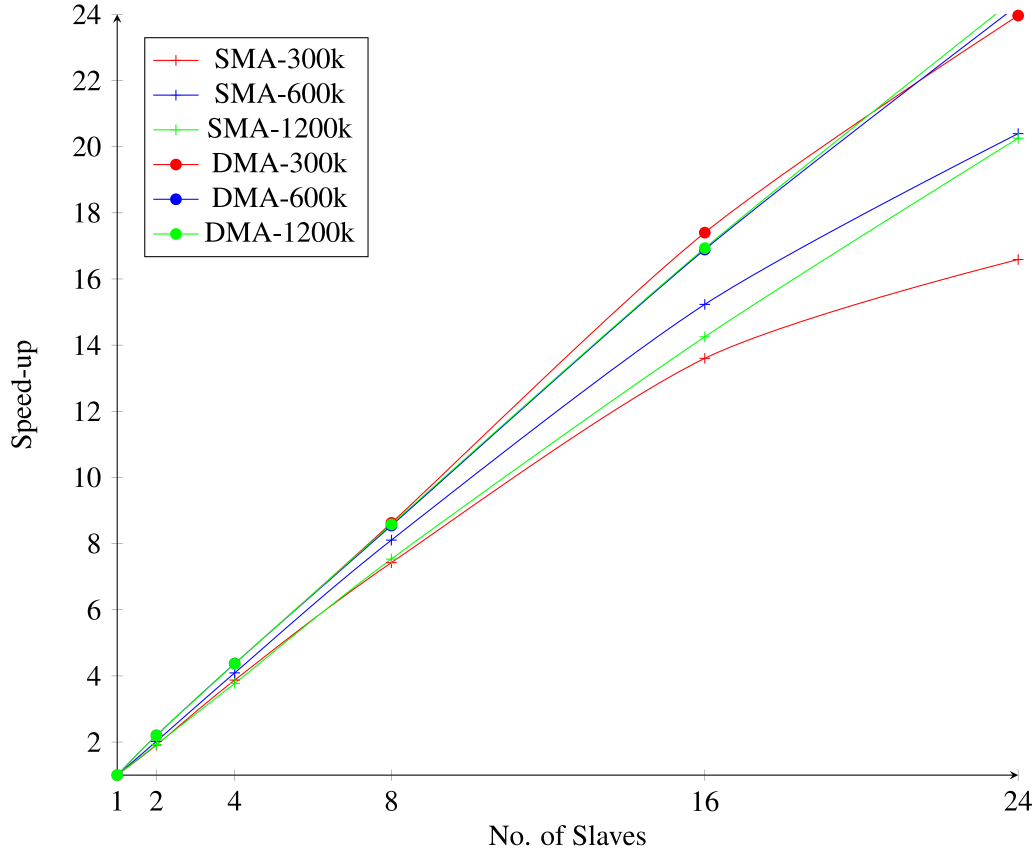


Figure 5.10: Effect of variation of queries size with dynamic scheduling in MAMMO dataset (1,000 elements per chunk)

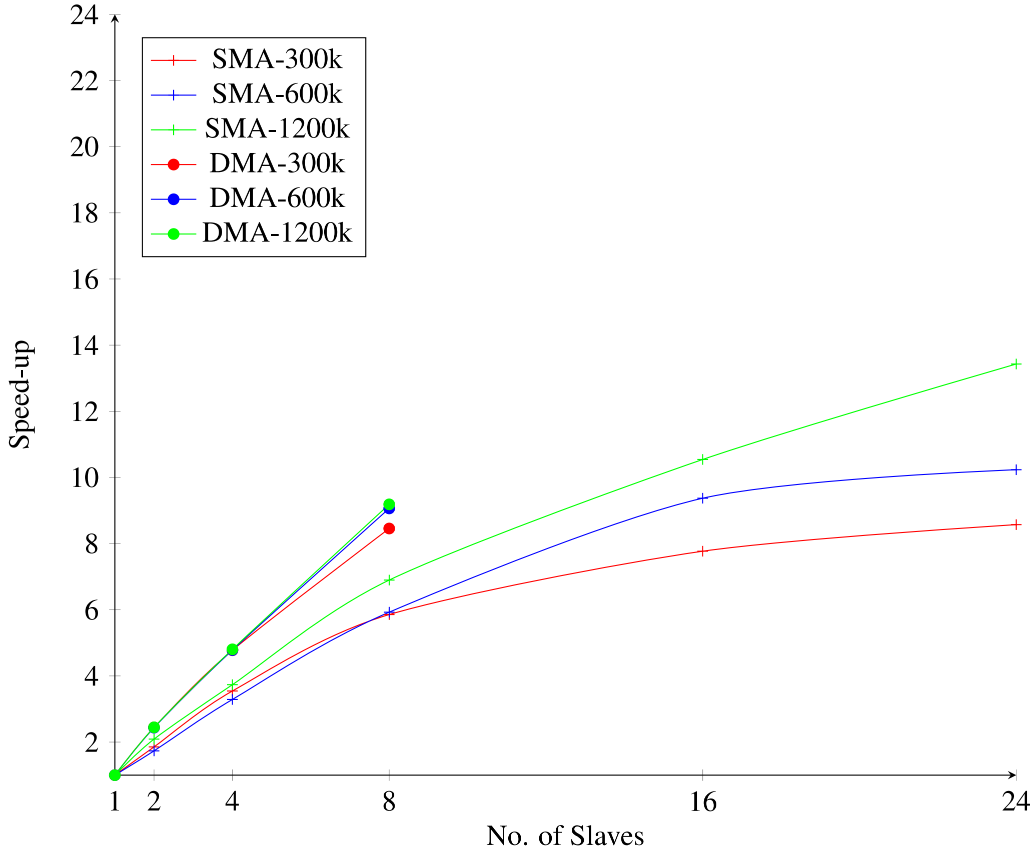


Figure 5.11: Effect of variation of queries size with dynamic scheduling in BLOG dataset (1,000 elements per chunk)

Figure 5.12, 5.13, 5.14 and 5.15 depict the behaviour of static scheduling, for each dataset, with varying queries size and 1,000 elements per chunk.

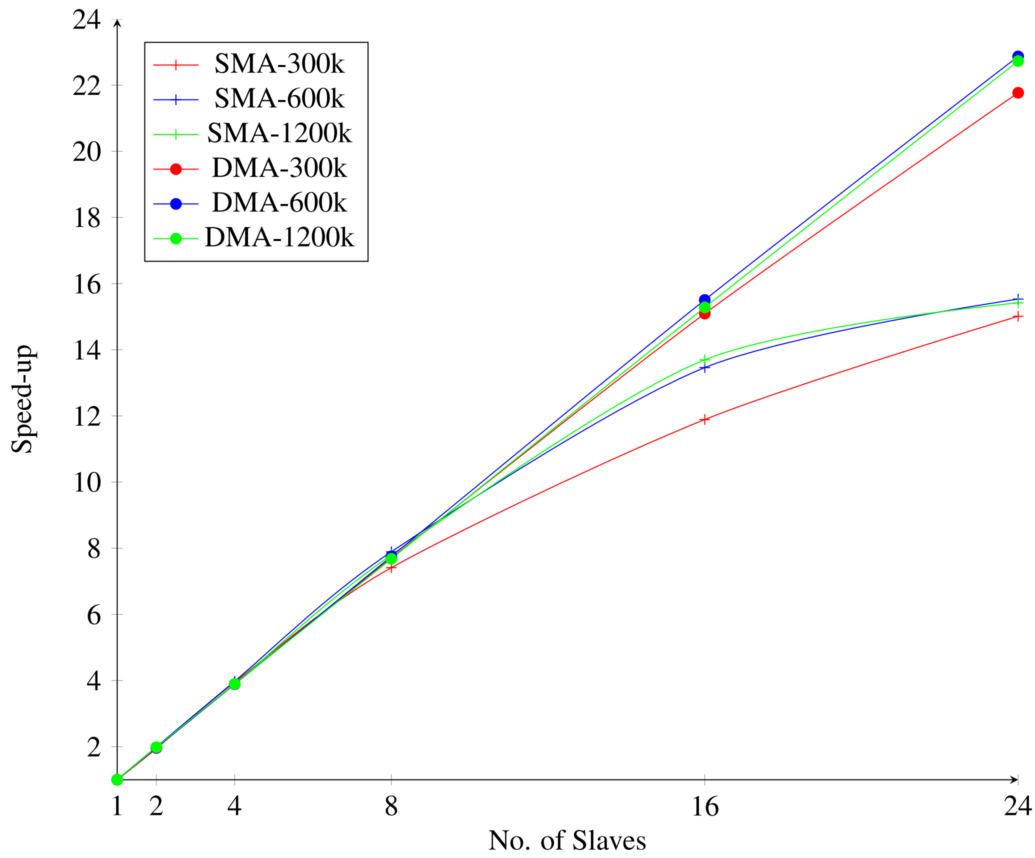


Figure 5.12: Effect of variation of queries size with static scheduling in ODD dataset (1,000 elements per chunk)

In general, these results show that DMA seems to be immune to variations on the dataset size. On the other hand, for SMA, these results show a generic tendency to obtain better speedups as we increase the dataset size and the number of slaves, which confirms the good scalability of our MapReduce for Prolog framework.

We believe all these tests consider and evaluate the most relevant features of MapReduce for Prolog. They demonstrate that our construct can scale efficiently, and that it can manage data with different granularity. We provide a flexible user interface, which allows for adapting the scheduling method to the data type, should the user wish to do so. The results are good for both shared and distributed memory implementations, making MapReduce for Prolog a flexible and agile MapReduce implementation for modest computing capabilities, whose focus is data native to Prolog.

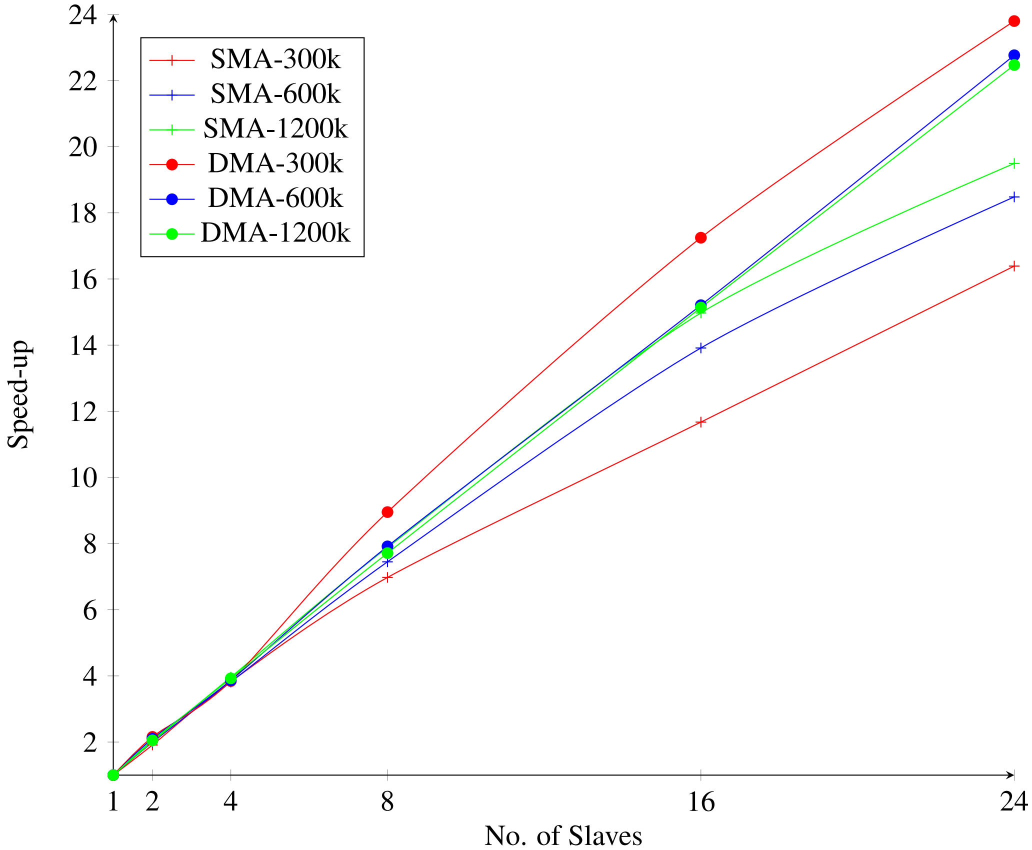


Figure 5.13: Effect of variation of queries size with static scheduling in PROB dataset (1,000 elements per chunk)

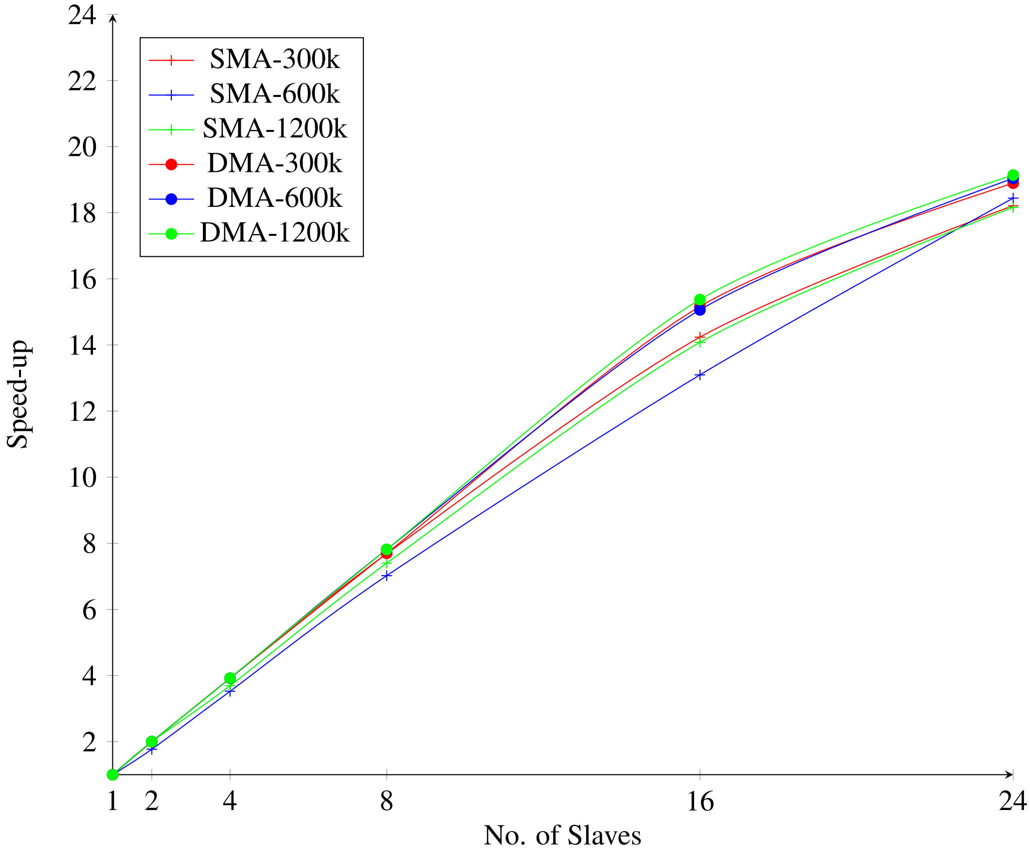


Figure 5.14: Effect of variation of queries size with static scheduling in MAMMO dataset (1,000 elements per chunk)

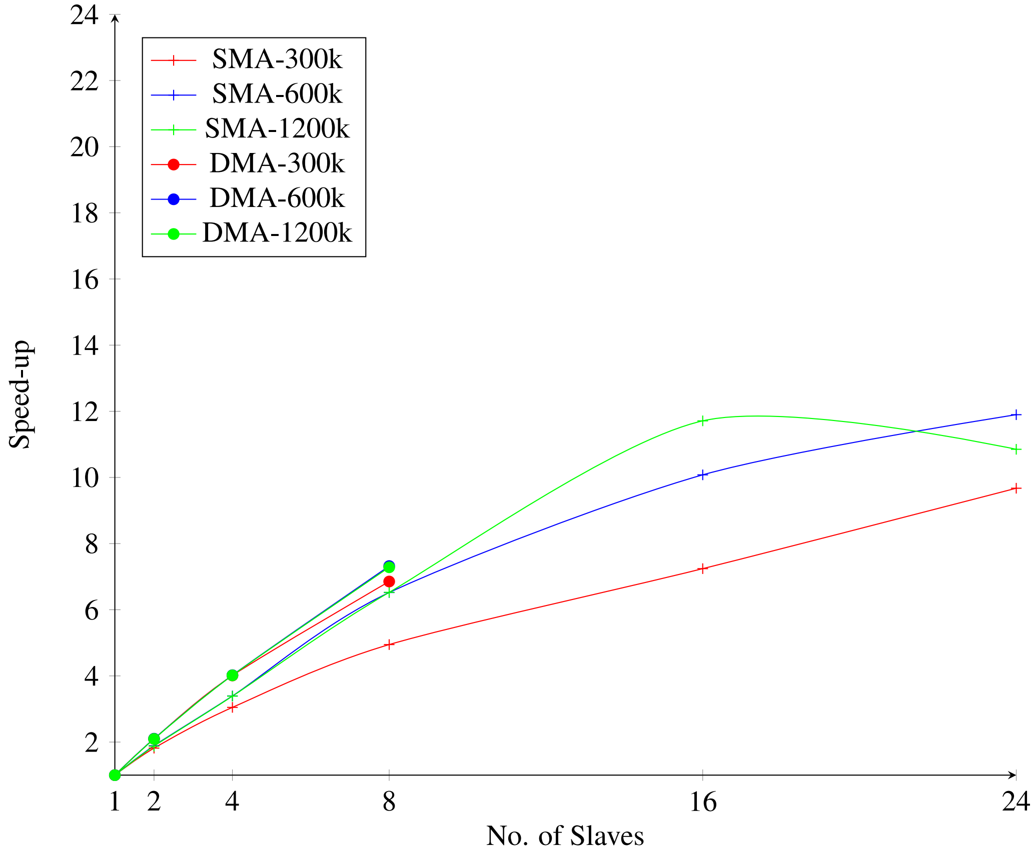


Figure 5.15: Effect of variation of queries size with static scheduling in BLOG dataset (1,000 elements per chunk)

Chapter 6

Conclusions and Future Work

In this last chapter, a summary of the main contributions of this work is made, and some directions for further work are provided. This thesis is then wraped up by some relevant final remarks.

6.1 Main Contributions

The work included in this thesis can be described as the design, implementation and testing process for a MapReduce for Prolog construct. Even though MapReduce for Prolog's architecture is standalone, the construct was developed, assessed and tuned for the Yap system. Usage examples and extensive documentation are also provided both in this work and in the code files.

This work can be divided into three main contributions:

MapReduce for Prolog – SMA is a version of the application that can be run in one machine alone, taking advantage of parallel processors, which are now more than ever common. This multi-threaded implementation presents nearly linear speed-ups until 24 cores as demonstrated by the tests. However, its performance is slightly worse than that of the MapReduce for Prolog – DMA, and this will be discussed in further detail in Section 6.2 below. To the best of our knowledge, there is no MapReduce implementation for shared memory alone, and so this work presents the novel opportunity of a transparent MapReduce for multicore shared memory architectures.

MapReduce for Prolog – DMA presents the same functionalities as MapReduce for Prolog – SMA, but has the advantage of running on a previously set up MPI grid, and thus provide cluster support. This implementation can be thought of a lightweight, agile MapReduce construct, as it is not redundant or fault tolerant, but rather aimed at smaller datasets and relatively modest computing capabilities; in these cases, the MapReduce for Prolog – DMA proves to have linear speed-ups and an overall good performance.

Scheduling technique assessment has demonstrated that MapReduce for Prolog can have very good speed-ups by using the adequate scheduling method for each data type. Testing in Chapter 5 has shown that the static scheduling algorithm performs better for numeric datasets, whilst the dynamic method proves to be a better choice for datasets native to Prolog. Other scheduling methods have been developed and evaluated, and have been found dispensable; those methods will not be made available in the final version of the MapReduce for Prolog code.

6.2 Further Work

We hope that the work resulting from this thesis has opened some new research opportunities, and that by making MapReduce for Prolog available to Yap users we can gather feedback and improve on this implementation. Currently, there are some points which still have room for improvement, and they are mentioned below. Together with those points, we have placed some suggestions, aimed mainly towards validating the implementation.

Improve BLOG dataset results Using Intel VTune Amplifier, it has been determined that the Yap Atom Table is not yet parallelized. As such, and since the BLOG dataset accesses that table often, its results were not in line with the remaining work. It would be revelant to develop a version of the Yap Atom Table for multi-threaded applications, and this would enable a subsequent improvement on the BLOG dataset performance for shared memory.

Develop a single MapReduce for Prolog Once Yap is finely tuned for thread support, it would be pertinent to develop a hybrid version, with two scheduling levels, as originally described in Chapter 3. We believe this would yield even better results, if not speed-up wise, quite possibly in terms of overall time. At any rate, the fact that the user does not have to choose between the shared and distributed memory versions would always be an improvement.

Distribute Data Across Cluster A relevant upgrade would also be to either set a network shared memory space for machines on the cluster or develop a predicate to separate data according to slaves and distribute it without user intervention. This would be interesting because each slave should only read the data it will require, thus making the setup time much shorter.

Test MapReduce for Prolog with a large dataset Even though MapReduce for Prolog's main target is not demanding data processing, it would be pertinent to see how our construct handles a more lengthy dataset; it would be interesting to determine if the speed-ups are still linear for that case.

MPI Guide The authors would like to develop and include a practical MPI configuration guide with the code. Whilst we are aware there is extensive documentation on message passing protocol, we would like to include a fully functional example using this interface for MapReduce for Prolog, so as to ensure easy and fast configuration for even more basic users.

Compare MapReduce for Prolog with other frameworks There is a vast number of MapReduce frameworks described in the literature, and it would be relevant to evaluate MapReduce for Prolog's performance against that of Hadoop or Twister, for instance. The set-up times, as well as the speed-ups should be considered.

6.3 Final Remark

A MapReduce parallel construct was designed and implemented in the Yap system. This construct provides an elegant way of implementing many applications in the summation form in Prolog [8], with the advantage of being intrinsically parallelizable. Two parallel implementations of the MapReduce are provided: a multithreaded and a message passing. In contrast to the Google's MapReduce implementation [7], whose focus is on distributed processing of data stored in disk, our implementation focuses on parallelization of the map and reduce operations where the data is already in memory.

This implementation has been tested using four applications and an evaluation of how different scheduling strategies and *chunk* sizes can affect performance concluded that: (i) our MapReduce construct can have linear speedups up to 24 processors; (ii) a dynamic distributed scheduling strategy, in general, performs better than centralized or static strategies; (iii) the performance varies signifficantly with the number of items being sent to each processor at a time; and (iv) our MapReduce model is a good alternative for taking advantage of the currently available low cost multi-core architectures.

One of the limitations of performance is related to the data synchronization used in the Yap implementation. Work is in progress to decentralize the access to data structures in order to further improve performance. We have also been studying best ways of executing MapReduce in the hybrid distributed shared-memory multi-core architectures.

Appendix A

Walltime Data

This chapter presents the raw data concerning the variation of MapReduce calls' walltimes with the number of queries. All times are given in milliseconds and the chapter is divided in sections according to the scheduling method and subsections according to the dataset. On the tables, `Slaves` denotes the number of slaves used to process the call, the `Average` is the average of five runs using that number of slaves and `Speedup` represents the speed-up calculated from the time for one slave on that table. This appendix includes the data used to plot Figures 5.1, 5.2, 5.3, 5.4, 5.8, 5.9, 5.10, 5.11, 5.12, 5.13, 5.14, 5.15 in Chapter 5.

A.1 Dynamic Scheduling

A.1.1 MAMMO

Slaves	1	2	4	8	16	24
	801	364	185	94	44	33
	801	363	183	92	45	34
	800	365	183	93	46	34
	801	363	183	92	47	33
	799	363	182	93	48	33
Average	800	364	183	93	46	33
Speedup	1,00	2,20	4,37	8,62	17,40	23,96

Table A.1: MAMMO DMA 300k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	1118	586	288	151	84	64
	1116	587	290	151	81	67
	1119	585	289	151	82	68
	1119	587	288	150	82	69
	1118	585	288	150	82	69
Average	1118	586	289	151	82	67
Speedup	1,00	1,91	3,87	7,42	13,60	16,59

Table A.2: MAMMO SMA 300k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	1598	727	364	186	94	65
	1598	726	364	186	95	65
	1598	727	369	191	94	69
	1597	726	365	186	97	65
	1598	725	365	186	93	65
Average	1598	726	365	187	95	66
Speedup	1,00	2,20	4,37	8,54	16,89	24,28

Table A.3: MAMMO DMA 600k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	2418	1197	593	296	157	119
	2420	1198	590	296	156	117
	2419	1198	591	301	161	117
	2419	1195	591	299	159	121
	2418	1197	591	300	161	119
Average	2419	1197	591	298	159	119
Speedup	1,00	2,02	4,09	8,11	15,23	20,39

Table A.4: MAMMO SMA 600k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	3194	1451	730	372	189	130
	3198	1454	736	375	186	133
	3188	1451	729	371	192	128
	3184	1450	727	371	187	129
	3186	1450	731	370	188	130
Average	3190	1451	731	372	188	130
Speedup	1,00	2,20	4,37	8,58	16,93	24,54

Table A.5: MAMMO DMA 1200k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	4483	2332	1187	595	317	219
	4485	2326	1189	593	314	220
	4477	2330	1188	595	312	218
	4480	2331	1189	596	315	229
	4478	2326	1185	595	314	220
Average	4481	2329	1188	595	314	221
Speedup	1,00	1,92	3,77	7,53	14,25	20,26

Table A.6: MAMMO SMA 1200k (1000 elems/chunk)

A.1.2 BLOG

Slaves	1	2	4	8
	715	292	150	81
	717	293	150	80
	715	294	149	80
	715	292	150	90
	715	293	150	92
Average	715	293	150	85
Speedup	1,00	2,44	4,78	8,46

Table A.7: BLOG DMA 300k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	856	492	252	160	121	112
	854	493	260	160	122	93
	891	491	261	160	125	113
	982	492	255	148	114	113
	978	493	259	151	105	101
Average	912	492	257	156	117	106
Speedup	1,00	1,85	3,54	5,85	7,77	8,57

Table A.8: BLOG SMA 300k (1000 elems/chunk)

Slaves	1	2	4	8
	1428	586	299	157
	1428	587	299	157
	1428	589	299	161
	1427	584	298	157
	1431	587	299	156
Average	1428	587	299	158
Speedup	1,00	2,44	4,78	9,06

Table A.9: BLOG DMA 600k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	1692	976	506	281	175	167
	1696	977	512	287	176	167
	1694	975	524	280	185	172
	1692	975	517	295	178	152
	1689	979	515	285	189	169
Average	1693	976	515	286	181	165
Speedup	1,00	1,73	3,29	5,93	9,37	10,23

Table A.10: BLOG SMA 600k (1000 elems/chunk)

Slaves	1	2	4	8
	2867	1173	598	312
	2869	1176	601	314
	2869	1173	597	312
	2873	1174	595	312
	2871	1172	596	312
Average	2870	1174	597	312
Speedup	1,00	2,45	4,80	9,19

Table A.11: BLOG DMA 1200k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	3802	1828	1012	551	352	286
	3796	1814	1022	549	364	289
	3796	1818	1013	564	360	274
	3803	1813	1015	545	361	281
	3792	1815	1022	544	364	284
Average	3798	1818	1017	551	360	283
Speedup	1,00	2,09	3,74	6,90	10,54	13,43

Table A.12: BLOG SMA 1200k (1000 elems/chunk)

A.1.3 PROB

Slaves	1	2	4	8	16	24
	400	184	93	48	24	16
	399	183	92	46	24	17
	399	184	91	47	24	16
	399	184	91	46	24	17
	398	184	92	47	24	16
Average	399	184	92	47	24	16
Speedup	1,00	2,17	4,35	8,53	16,62	24,33

Table A.13: PROB DMA 300k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	501	250	121	64	32	26
	500	250	122	62	34	24
	501	249	120	62	32	25
	501	251	120	64	32	24
	500	250	121	62	32	24
Average	501	250	121	63	32	25
Speedup	1,00	2,00	4,14	7,97	15,45	20,35

Table A.14: PROB SMA 300k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	799	370	188	100	54	39
	793	363	182	95	49	32
	797	370	189	100	53	37
	794	365	184	96	50	34
	792	362	183	94	49	33
Average	795	366	185	97	51	35
Speedup	1,00	2,17	4,29	8,20	15,59	22,71

Table A.15: PROB DMA 600k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	1006	506	247	124	64	48
	1006	503	247	124	69	48
	1006	505	248	125	66	47
	1008	504	245	126	64	47
	1011	505	247	126	64	52
Average	1007	505	247	125	65	48
Speedup	1,00	2,00	4,08	8,06	15,40	20,81

Table A.16: PROB SMA 600k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	1591	726	367	189	97	66
	1604	737	380	201	109	78
	1587	724	365	189	95	63
	1586	728	368	192	98	67
	1604	745	385	208	116	83
Average	1594	732	373	196	103	71
Speedup	1,00	2,18	4,27	8,14	15,48	22,33

Table A.17: PROB DMA 1200k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	2008	998	489	245	127	97
	2010	1003	487	253	126	95
	2005	996	489	247	130	104
	2013	999	489	247	127	96
	2006	997	488	246	129	102
Average	2008	999	488	248	128	99
Speedup	1,00	2,01	4,11	8,11	15,72	20,33

Table A.18: PROB SMA 1200k (1000 elems/chunk)

A.1.4 ODD

Slaves	1	2	4	8	16	24
	282	124	65	32	17	11
	284	124	65	33	17	11
	282	124	65	32	16	11
	283	124	65	32	17	12
	280	125	66	32	16	11
Average	282	124	65	32	17	11
Speedup	1,00	2,27	4,33	8,76	17,00	25,20

Table A.19: ODD DMA 300k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	264	132	64	32	18	16
	262	132	64	33	17	16
	263	131	63	33	16	12
	263	130	66	34	18	12
	261	130	65	34	16	13
Average	263	131	64	33	17	14
Speedup	1,00	2,00	4,08	7,91	15,45	19,03

Table A.20: ODD SMA 300k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	562	248	129	64	33	22
	563	248	129	63	32	22
	566	252	133	67	36	27
	566	251	129	64	33	22
	566	249	128	63	33	23
Average	565	250	130	64	33	23
Speedup	1,00	2,26	4,36	8,79	16,90	24,34

Table A.21: ODD DMA 600k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	515	246	124	64	37	25
	519	247	125	65	34	25
	517	249	132	65	33	25
	519	247	131	65	33	25
	518	259	131	67	32	25
Average	518	250	129	65	34	25
Speedup	1,00	2,07	4,02	7,94	15,31	20,70

Table A.22: ODD SMA 600k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	1130	497	257	127	65	46
	1133	499	261	132	70	49
	1129	496	257	127	64	44
	1129	495	256	128	65	46
	1130	498	255	127	66	46
Average	1130	497	257	128	66	46
Speedup	1,00	2,27	4,39	8,82	17,12	24,46

Table A.23: ODD DMA 1200k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	1044	523	260	133	64	46
	1039	523	260	130	64	47
	1040	517	257	130	63	46
	1040	517	258	130	65	49
	1044	515	258	130	64	46
Average	1041	519	259	131	64	47
Speedup	1,00	2,01	4,03	7,97	16,27	22,25

Table A.24: ODD SMA 1200k (1000 elems/chunk)

A.2 Static Scheduling

A.2.1 MAMMO

Slaves	1	2	4	8	16	24
	708	354	181	91	47	38
	706	354	180	93	46	37
	707	355	180	92	46	37
	706	353	180	91	47	38
	706	353	181	92	47	37
Average	707	354	180	92	47	37
Speedup	1.00	2.00	3.92	7.70	15.16	18.89

Table A.25: MAMMO DMA 300k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	1239	619	315	161	90	77
	1238	623	318	161	88	69
	1239	622	314	161	86	67
	1239	621	316	161	85	63
	1237	623	317	162	86	64
Average	1238	622	316	161	87	68
Speedup	1.00	1.99	3.92	7.68	14.23	18.21

Table A.26: MAMMO SMA 300k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	1416	706	360	181	95	74
	1410	705	363	182	93	74
	1413	706	360	180	94	74
	1412	706	360	180	93	75
	1412	706	360	181	94	74
Average	1413	706	361	181	94	74
Speedup	1.00	2.00	3.92	7.81	15.06	19.04

Table A.27: MAMMO DMA 600k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	2186	1247	624	313	166	119
	2198	1250	623	315	168	117
	2189	1236	623	314	168	121
	2200	1235	623	310	168	121
	2199	1234	621	311	168	117
Average	2194	1240	623	313	168	119
Speedup	1.00	1.77	3.52	7.02	13.09	18.44

Table A.28: MAMMO SMA 600k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	2832	1413	720	363	183	149
	2825	1417	721	360	184	148
	2827	1411	725	367	184	147
	2829	1413	720	360	186	147
	2830	1420	720	360	183	148
Average	2829	1415	721	362	184	148
Speedup	1.00	2.00	3.92	7.81	15.37	19.14

Table A.29: MAMMO DMA 1200k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	4625	2312	1256	626	331	256
	4622	2324	1247	625	328	254
	4624	2310	1254	624	330	254
	4624	2311	1248	625	327	255
	4622	2317	1255	624	326	254
Average	4623	2315	1252	625	328	255
Speedup	1.00	2.00	3.69	7.40	14.08	18.16

Table A.30: MAMMO SMA 1200k (1000 elems/chunk)

A.2.2 BLOG

Slaves	1	2	4	8
	574	274	142	84
	572	274	143	84
	574	273	143	83
	573	272	143	83
	572	273	143	84
Average	573	273	143	84
Speedup	1.00	2.10	4.01	6.85

Table A.31: BLOG DMA 300k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	825	453	271	162	113	84
	826	456	271	167	114	84
	825	452	266	167	114	84
	822	454	273	168	115	87
	823	454	271	169	113	87
Average	824	454	270	167	114	85
Speedup	1.00	1.82	3.05	4.95	7.24	9.67

Table A.32: BLOG SMA 300k (1000 elems/chunk)

Slaves	1	2	4	8
	1158	545	284	156
	1145	545	290	158
	1145	548	286	156
	1147	545	283	157
	1144	549	284	157
Average	1148	546	285	157
Speedup	1.00	2.10	4.02	7.32

Table A.33: BLOG DMA 600k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	1869	991	551	288	184	157
	1871	994	549	286	184	158
	1871	991	554	288	185	158
	1882	988	552	287	189	157
	1870	992	553	287	187	157
Average	1873	991	552	287	186	157
Speedup	1.00	1.89	3.39	6.52	10.08	11.90

Table A.34: BLOG SMA 600k (1000 elems/chunk)

Slaves	1	2	4	8
	2299	1097	568	316
	2293	1095	570	315
	2294	1093	578	319
	2293	1094	569	313
	2293	1101	571	312
Average	2294	1096	571	315
Speedup	1.00	2.09	4.02	7.28

Table A.35: BLOG DMA 1200k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	3784	2012	1119	584	320	349
	3773	2015	1120	578	322	350
	3789	2019	1112	583	322	351
	3787	2018	1113	579	327	351
	3780	2015	1119	574	324	342
Average	3783	2016	1117	580	323	349
Speedup	1.00	1.88	3.39	6.53	11.71	10.85

Table A.36: BLOG SMA 1200k (1000 elems/chunk)

A.2.3 PROB

Slaves	1	2	4	8	16	24
	377	174	98	43	22	15
	376	175	99	42	22	15
	377	175	98	42	22	20
	375	175	97	42	22	15
	375	173	97	41	21	14
Average	376	174	98	42	22	16
Speedup	1.00	2.16	3.84	8.95	17.25	23.80

Table A.37: PROB DMA 300k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	480	251	130	77	47	29
	480	251	126	74	47	27
	477	251	123	66	37	28
	478	251	123	63	37	30
	478	248	123	63	37	32
Average	479	250	125	69	41	29
Speedup	1.00	1.91	3.83	6.98	11.67	16.39

Table A.38: PROB SMA 300k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	734	347	190	91	47	32
	731	351	190	91	46	31
	731	348	189	98	54	32
	739	351	189	91	47	33
	730	348	190	92	47	33
Average	733	349	190	93	48	32
Speedup	1.00	2.10	3.87	7.92	15.21	22.76

Table A.39: PROB DMA 600k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	965	486	250	134	70	55
	968	486	254	129	74	51
	969	487	255	129	70	52
	969	487	256	129	67	50
	971	486	254	129	67	54
Average	968	486	254	130	70	52
Speedup	1.00	1.99	3.82	7.45	13.91	18.48

Table A.40: PROB SMA 600k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	1443	698	365	184	93	63
	1447	705	367	186	93	64
	1458	704	381	197	107	63
	1441	715	364	188	92	63
	1446	705	366	183	93	69
Average	1447	705	369	188	96	64
Speedup	1.00	2.05	3.93	7.71	15.14	22.47

Table A.41: PROB DMA 1200k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	2022	1003	503	256	136	104
	2011	1003	510	255	131	106
	2017	1003	512	257	143	102
	2014	1004	508	254	131	100
	2014	1005	511	256	132	105
Average	2016	1004	509	256	135	103
Speedup	1.00	2.01	3.96	7.89	14.97	19.49

Table A.42: PROB SMA 1200k (1000 elems/chunk)

A.2.4 ODD

Slaves	1	2	4	8	16	24
	227	116	58	29	15	11
	229	116	59	30	15	10
	225	115	58	29	15	11
	227	115	58	30	15	10
	224	115	58	29	15	10
Average	226	115	58	29	15	10
Speedup	1.00	1.96	3.89	7.70	15.09	21.77

Table A.43: ODD DMA 300k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	242	132	61	34	22	18
	241	121	61	33	20	17
	238	124	61	33	21	15
	240	120	61	31	19	16
	240	121	60	31	19	14
Average	240	124	61	32	20	16
Speedup	1.00	1.94	3.95	7.41	11.89	15.01

Table A.44: ODD SMA 300k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	456	230	116	58	29	20
	450	231	120	59	29	20
	455	230	115	58	30	19
	451	228	115	58	30	20
	452	230	116	59	28	20
Average	453	230	116	58	29	20
Speedup	1.00	1.97	3.89	7.75	15.51	22.87

Table A.45: ODD DMA 600k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	484	243	126	62	41	34
	484	243	121	63	34	34
	485	243	121	62	33	28
	485	243	120	60	36	28
	485	242	121	60	36	32
Average	485	243	122	61	36	31
Speedup	1.00	2.00	3.98	7.89	13.46	15.53

Table A.46: ODD SMA 600k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	908	458	232	120	60	40
	901	455	232	117	59	41
	903	451	237	121	58	39
	906	453	231	116	61	40
	905	461	229	115	58	39
Average	905	456	232	118	59	40
Speedup	1.00	1.99	3.90	7.68	15.28	22.73

Table A.47: ODD DMA 1200k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	956	480	246	122	76	58
	956	483	244	121	66	71
	963	487	244	123	68	62
	946	482	246	123	66	64
	959	479	245	124	73	55
Average	956	482	245	123	70	62
Speedup	1.00	1.98	3.90	7.80	13.70	15.42

Table A.48: ODD SMA 1200k (1000 elems/chunk)

A.3 Single-step Scheduling

A.3.1 MAMMO

Slaves	1	2	4	8	16	24
	832	347	174	100	55	39
	693	345	177	100	55	37
	695	345	173	100	55	38
	697	346	172	101	55	37
	695	345	172	101	54	38
Average	722	346	174	100	55	38
Speedup	1.00	2.09	4.16	7.20	13.18	19.11

Table A.49: MAMMO DMA 300k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	1234	621	310	161	92	66
	1235	608	311	158	91	67
	1231	609	312	157	90	66
	1231	613	313	170	90	70
	1229	610	312	159	88	68
Average	1232	612	312	161	90	67
Speedup	1.00	2.01	3.95	7.65	13.66	18.28

Table A.50: MAMMO SMA 300k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	1518	691	344	196	110	77
	1518	690	347	197	109	76
	1519	690	348	198	111	77
	1518	693	347	198	110	76
	1517	690	347	196	110	76
Average	1518	691	347	197	110	76
Speedup	1.00	2.20	4.38	7.71	13.80	19.87

Table A.51: MAMMO DMA 600k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	2433	1190	626	323	177	130
	2422	1187	626	320	178	127
	2425	1190	619	323	175	126
	2421	1189	623	327	176	126
	2412	1199	625	319	173	126
Average	2423	1191	624	322	176	127
Speedup	1.00	2.03	3.88	7.51	13.78	19.08

Table A.52: MAMMO SMA 600k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	2978	1516	692	395	217	153
	2979	1517	698	394	216	154
	2985	1518	694	398	219	154
	2988	1513	694	393	225	154
	2979	1513	697	393	216	155
Average	2982	1515	695	395	219	154
Speedup	1.00	1.97	4.29	7.56	13.64	19.36

Table A.53: MAMMO DMA 1200k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	4943	2438	1225	628	342	251
	4907	2464	1226	629	346	253
	4894	2461	1224	622	342	249
	4914	2449	1224	622	341	254
	4905	2437	1224	627	345	249
Average	4913	2450	1225	626	343	251
Speedup	1.00	2.01	4.01	7.85	14.31	19.56

Table A.54: MAMMO SMA 1200k (1000 elems/chunk)

A.3.2 BLOG

Slaves	1	2	4	8
	744	286	174	89
	549	289	175	89
	552	290	176	90
	552	288	173	89
	552	290	174	89
Average	590	289	174	89
Speedup	1.00	2.04	3.38	6.61

Table A.55: BLOG DMA 300k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	874	480	274	211	136	129
	877	481	273	211	132	130
	883	478	273	212	132	128
	892	480	274	211	131	129
	891	476	276	213	131	128
Average	883	479	274	212	132	129
Speedup	1.00	1.84	3.22	4.17	6.67	6.86

Table A.56: BLOG SMA 300k (1000 elems/chunk)

Slaves	1	2	4	8
	1283	540	284	171
	1282	545	283	174
	1289	544	281	172
	1284	544	283	172
	1288	540	285	172
Average	1285	543	283	172
Speedup	1.00	2.37	4.54	7.46

Table A.57: BLOG SMA 600k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	1988	1038	607	324	226	235
	1994	1042	605	319	225	231
	1981	1043	604	320	226	235
	1988	1000	606	319	224	238
	1990	999	605	321	226	237
Average	1988	1024	605	321	225	235
Speedup	1.00	1.94	3.28	6.20	8.82	8.45

Table A.58: BLOG SMA 600k (1000 elems/chunk)

Slaves	1	2	4	8
	2461	1259	630	330
	2450	1258	635	333
	2458	1261	641	332
	2460	1258	642	334
	2464	1260	636	333
Average	2459	1259	637	332
Speedup	1.00	1.95	3.86	7.40

Table A.59: BLOG DMA 1200k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	3570	2047	1132	590	383	366
	3561	2042	1136	577	387	363
	3554	2038	1132	576	384	367
	3562	2041	1136	581	387	368
	3563	2047	1135	581	385	364
Average	3562	2043	1134	581	385	366
Speedup	1.00	1.74	3.14	6.13	9.25	9.74

Table A.60: BLOG SMA 1200k (1000 elems/chunk)

A.3.3 PROB

Slaves	1	2	4	8	16	24
	341	179	89	47	25	18
	346	173	91	48	25	18
	348	177	89	46	25	17
	343	175	91	47	25	18
	346	177	90	47	26	18
Average	345	176	90	47	25	18
Speedup	1.00	1.96	3.83	7.34	13.68	19.37

Table A.61: PROB DMA 300k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	528	269	139	78	54	41
	535	266	136	72	49	38
	532	269	137	73	49	38
	529	264	136	74	48	39
	532	266	136	74	55	43
Average	531	267	137	74	51	40
Speedup	1.00	1.99	3.88	7.16	10.42	13.35

Table A.62: PROB SMA 300k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	684	350	176	96	49	34
	695	347	178	95	48	34
	688	348	177	95	48	34
	689	346	177	97	49	34
	697	350	176	95	49	33
Average	691	348	177	96	49	34
Speedup	1.00	1.98	3.91	7.22	14.21	20.43

Table A.63: PROB DMA 600k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	1040	531	288	151	104	71
	1030	531	284	145	103	74
	1023	526	283	146	102	71
	1037	528	296	146	101	71
	1036	527	292	146	102	71
Average	1033	529	289	147	102	72
Speedup	1.00	1.95	3.58	7.04	10.09	14.43

Table A.64: PROB SMA 600k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	3190	699	352	201	98	66
	1369	709	354	203	98	67
	1385	709	352	201	98	67
	1376	715	356	201	98	67
	1360	712	357	202	98	67
Average	1736	709	354	202	98	67
Speedup	1.00	2.45	4.90	8.61	17.71	25.99

Table A.65: PROB DMA 1200k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	2064	1030	607	283	182	140
	2048	1034	601	282	181	138
	2041	1030	600	282	180	137
	2049	1025	598	283	174	163
	2039	1029	600	282	173	152
Average	2048	1030	601	282	178	146
Speedup	1.00	1.99	3.41	7.25	11.51	14.03

Table A.66: PROB SMA 1200k (1000 elems/chunk)

A.3.4 ODD

Slaves	1	2	4	8	16	24
	244	121	63	35	17	12
	246	122	63	35	18	13
	248	124	62	42	17	12
	248	123	62	35	18	13
	249	123	62	35	17	13
Average	247	123	62	36	17	13
Speedup	1.00	2.01	3.96	6.79	14.20	19.60

Table A.67: ODD DMA 300k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	283	149	78	65	44	30
	276	145	79	64	39	34
	286	147	79	65	40	30
	288	146	77	64	39	28
	276	146	77	64	40	28
Average	282	147	78	64	40	30
Speedup	1.00	1.92	3.61	4.38	6.98	9.39

Table A.68: ODD SMA 300k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	488	245	124	70	34	25
	494	245	123	71	34	25
	496	246	123	71	34	25
	491	245	123	70	34	25
	490	247	123	71	35	25
Average	492	246	123	71	34	25
Speedup	1.00	2.00	3.99	6.97	14.38	19.67

Table A.69: ODD DMA 600k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	571	295	162	127	79	55
	563	289	150	124	68	56
	584	293	153	124	67	53
	562	285	158	127	69	52
	560	290	160	129	70	51
Average	568	290	157	126	71	53
Speedup	1.00	1.96	3.63	4.50	8.05	10.64

Table A.70: ODD SMA 600k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	978	487	263	125	66	49
	979	487	265	126	68	49
	978	489	267	127	66	48
	972	486	265	126	68	48
	978	487	266	125	68	50
Average	977	487	265	126	67	49
Speedup	1.00	2.01	3.68	7.77	14.54	20.02

Table A.71: ODD DMA 1200k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	1123	603	308	204	151	98
	1111	581	314	200	140	98
	1108	611	318	203	141	105
	1150	608	310	201	141	99
	1112	607	318	197	144	105
Average	1121	602	314	201	143	101
Speedup	1.00	1.86	3.57	5.58	7.82	11.10

Table A.72: ODD SMA 1200k (1000 elems/chunk)

A.4 Workpool Scheduling

A.4.1 MAMMO

Slaves	1	2	4	8	16	24
	1086	558	288	148	80	62
	1088	557	291	147	80	59
	1087	558	287	147	83	59
	1087	554	294	148	78	61
	1088	555	287	147	78	60
Average	1087	556	289	147	80	60
Speedup	1.00	1.95	3.76	7.38	13.62	18.06

Table A.73: MAMMO SMA 300k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	2184	1139	575	293	153	115
	2184	1134	577	292	152	115
	2188	1134	575	290	154	116
	2185	1126	576	293	153	116
	2187	1134	574	292	153	115
Average	2186	1133	575	292	153	115
Speedup	1.00	1.93	3.80	7.48	14.28	18.94

Table A.74: MAMMO SMA 600k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	4606	2335	1154	593	308	221
	4619	2332	1158	592	306	221
	4620	2331	1155	591	307	221
	4619	2332	1153	590	309	222
	4618	2334	1156	582	306	220
Average	4616	2333	1155	590	307	221
Speedup	1.00	1.98	4.00	7.83	15.03	20.89

Table A.75: MAMMO SMA 1200k (1000 elems/chunk)

A.4.2 BLOG

Slaves	1	2	4	8	16	24
	828	445	245	157	106	101
	830	447	251	154	110	100
	827	449	252	156	104	100
	828	447	249	156	108	100
	829	446	249	153	111	100
Average	828	446	249	155	107	100
Speedup	1.00	1.85	3.32	5.34	7.68	8.27

Table A.76: BLOG SMA 300k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	1864	956	483	281	185	175
	1860	957	477	271	183	173
	1863	958	475	263	187	175
	1864	957	490	273	180	166
	1868	957	484	268	177	178
Average	1863	957	481	271	182	173
Speedup	1.00	1.95	3.87	6.87	10.22	10.75

Table A.77: BLOG SMA 600k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	3247	1784	954	524	327	276
	3247	1782	957	519	326	288
	3246	1785	942	513	318	278
	3251	1779	943	522	323	288
	3246	1787	940	520	322	291
Average	3247	1783	947	519	323	284
Speedup	1.00	1.82	3.43	6.25	10.05	11.43

Table A.78: BLOG SMA 1200k (1000 elems/chunk)

A.4.3 PROB

Slaves	1	2	4	8	16	24
	483	243	119	61	33	27
	483	243	119	63	32	24
	482	242	119	60	32	24
	481	235	120	61	33	24
	482	236	119	60	33	27
Average	482	240	119	61	33	25
Speedup	1.00	2.01	4.05	7.90	14.79	19.13

Table A.79: PROB SMA 300k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	937	491	241	122	65	45
	936	486	241	122	64	44
	934	478	241	123	63	44
	932	478	241	122	65	46
	939	478	240	121	64	45
Average	936	482	241	122	64	45
Speedup	1.00	1.94	3.89	7.67	14.57	20.88

Table A.80: PROB SMA 600k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	1932	966	474	246	124	88
	1931	968	476	240	133	91
	1925	971	473	242	130	92
	1937	965	476	243	124	94
	1927	971	475	240	124	90
Average	1930	968	475	242	127	91
Speedup	1.00	1.99	4.07	7.97	15.20	21.21

Table A.81: PROB SMA 1200k (1000 elems/chunk)

A.4.4 ODD

Slaves	1	2	4	8	16	24
	245	122	62	30	17	14
	243	121	60	31	16	13
	241	121	60	30	16	12
	243	121	61	31	15	15
	243	120	60	30	17	13
Average	243	121	61	30	16	13
Speedup	1.00	2.01	4.01	7.99	15.00	18.13

Table A.82: ODD SMA 300k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	478	240	121	61	31	23
	475	239	120	60	31	23
	475	242	119	61	31	22
	470	242	120	60	31	23
	473	242	119	61	31	22
Average	474	241	120	61	31	23
Speedup	1.00	1.97	3.96	7.83	15.30	20.98

Table A.83: ODD SMA 600k (1000 elems/chunk)

Slaves	1	2	4	8	16	24
	960	476	239	120	63	44
	955	477	240	121	64	45
	957	471	240	121	61	46
	956	476	240	122	64	46
	953	479	236	120	62	45
Average	956	476	239	121	63	45
Speedup	1.00	2.01	4.00	7.92	15.23	21.15

Table A.84: ODD SMA 1200k (1000 elems/chunk)

Appendix B

Load Balancing Data

This appendix includes raw data used to plot Figure 5.5 in Chapter 5, concerning the load balancing. The three sections in this chapter contain the data for each scheduling method. All times are given in miliseconds, `d max` is the difference between the fastest and the slowest slave and `d max %` is the ratio between that difference and the average walltime of the call.

B.1 Dynamic Scheduling

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
walltime	189	191	193	189	192	191
slaves	183	184	182	184	183	183
	173	177	175	174	175	175
	181	180	185	185	187	184
	175	172	171	172	179	174
	185	184	185	185	184	185
	174	177	169	170	170	172
	186	185	185	185	185	185
	168	174	175	174	175	173
	186	184	186	182	186	185
	175	177	175	168	180	175
	183	187	184	183	183	184
	172	174	170	170	173	172
	185	176	185	184	184	183
	172	174	172	174	171	173
	185	183	187	179	183	183
	175	178	170	174	171	174
					d max	13
					d max%	7.0%

Table B.1: MAMMO DMA 1200k 16 slaves (1000 elems/chunk)

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
walltime	100	101	103	95	105	101
slaves	91	92	92	92	91	92
	73	81	75	81	80	78
	83	84	84	87	89	85
	79	74	74	79	78	77
	93	89	92	87	92	91
	80	80	82	78	77	79
	91	90	88	90	90	90
	83	79	84	75	76	79
	92	91	92	91	90	91
	81	83	76	78	78	79
	91	92	93	92	92	92
	77	80	79	84	77	79
	91	88	90	93	91	91
	76	81	82	83	78	80
	91	88	92	91	91	91
	84	85	84	81	84	84
					d max	15
					d max %	15.1%

Table B.2: PROB DMA 1200k 16 slaves (1000 elems/chunk)

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
walltime	295	300	303	300	300	300
slaves	287	287	288	276	285	285
	259	270	267	258	268	264
	287	287	296	296	290	291
	260	268	264	268	263	265
	289	287	289	287	283	287
	261	262	262	257	255	259
	294	287	289	286	287	289
	256	269	262	262	267	263
					d max	32
					d max %	10.6%

Table B.3: BLOG DMA 1200k 8 slaves (1000 elems/chunk)

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
walltime	64	68	69	66	68	67
slaves	60	62	62	59	62	61
	53	55	47	51	45	50
	60	60	61	60	62	61
	52	48	49	49	52	50
	61	61	63	61	59	61
	55	53	50	52	47	51
	62	61	62	61	61	61
	53	54	49	52	53	52
	60	62	64	62	63	62
	54	54	47	52	52	52
	61	61	58	59	61	60
	53	50	47	50	50	50
	62	60	61	61	61	61
	49	48	48	48	54	49
	61	60	60	60	62	61
	51	48	48	47	54	50
					d max	13
					d max %	19.1%

Table B.4: ODD DMA 1200k 16 slaves (1000 elems/chunk)

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
walltime	316	310	307	313	310	311
slaves	268	272	292	291	276	280
	264	258	274	288	267	270
	272	264	265	257	269	265
	266	267	263	258	262	263
	288	270	266	274	265	273
	280	263	267	265	261	267
	276	273	292	272	270	277
	297	269	261	264	275	273
	268	256	267	263	260	263
	265	260	269	267	273	267
	277	287	291	284	275	283
	285	263	265	261	259	267
	245	261	263	263	264	259
	260	271	259	271	272	267
	264	270	265	291	289	276
	267	258	265	268	263	264
					d max	24
					d max %	7.6%

Table B.5: MAMMO SMA 1200k 16 slaves (1000 elems/chunk)

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
walltime	136	135	136	137	137	136
slaves	95	95	90	93	93	93
	95	89	91	89	98	92
	92	103	126	112	89	104
	96	90	96	94	97	95
	91	89	87	113	107	97
	100	91	92	94	93	94
	88	89	95	84	91	89
	100	85	111	122	98	103
	109	116	103	131	93	110
	95	88	91	88	84	89
	95	96	85	89	91	91
	120	110	93	106	96	105
	94	90	112	88	98	96
	98	94	90	98	96	95
	94	91	86	95	94	92
	93	111	104	113	91	102
					d max	21
					d max %	15.6%

Table B.6: PROB SMA 1200k 16 slaves (1000 elems/chunk)

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
walltime	349	368	359	359	348	357
slaves	271	279	289	322	276	287
	259	276	268	282	310	279
	274	282	276	266	277	275
	274	285	274	264	266	273
	323	296	273	270	282	289
	280	284	283	277	277	280
	280	309	306	276	278	290
	270	275	285	274	268	274
	270	275	279	288	278	278
	273	278	273	269	280	275
	274	269	270	275	275	273
	306	283	290	286	283	290
	287	291	318	268	270	287
	288	304	278	285	293	290
	291	330	284	281	292	296
	291	289	285	297	290	290
					d max	23
					d max %	6.4%

Table B.7: BLOG SMA 1200k 16 slaves (1000 elems/chunk)

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
walltime	65	66	67	68	69	67
slaves	23	27	24	19	28	24
	44	44	28	29	29	35
	33	17	29	54	31	33
	23	24	29	22	30	26
	37	29	28	24	28	29
	23	21	25	27	24	24
	36	49	24	27	26	32
	19	23	30	37	28	27
	29	26	47	26	32	32
	22	20	58	33	35	34
	19	24	29	24	29	25
	30	30	30	39	24	31
	22	30	28	38	46	33
	26	27	29	25	28	27
	22	29	27	24	30	26
	29	21	20	27	27	25
					d max	11
					d max %	16.1%

Table B.8: ODD SMA 1200k 16 slaves (1000 elems/chunk)

B.2 Static Scheduling

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
walltime	189	191	193	189	192	191
slaves	183	184	182	184	183	183
	173	177	175	174	175	175
	181	180	185	185	187	184
	175	172	171	172	179	174
	185	184	185	185	184	185
	174	177	169	170	170	172
	186	185	185	185	185	185
	168	174	175	174	175	173
	186	184	186	182	186	185
	175	177	175	168	180	175
	183	187	184	183	183	184
	172	174	170	170	173	172
	185	176	185	184	184	183
	172	174	172	174	171	173
	185	183	187	179	183	183
	175	178	170	174	171	174
					d max	13
					d max %	7.0%

Table B.9: MAMMO DMA 1200k 16 slaves (1000 elems/chunk)

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
walltime	100	101	103	95	105	101
slaves	91	92	92	92	91	92
	73	81	75	81	80	78
	83	84	84	87	89	85
	79	74	74	79	78	77
	93	89	92	87	92	91
	80	80	82	78	77	79
	91	90	88	90	90	90
	83	79	84	75	76	79
	92	91	92	91	90	91
	81	83	76	78	78	79
	91	92	93	92	92	92
	77	80	79	84	77	79
	91	88	90	93	91	91
	76	81	82	83	78	80
	91	88	92	91	91	91
	84	85	84	81	84	84
					d max	15
					d max %	15.1%

Table B.10: PROB DMA 1200k 16 slaves (1000 elems/chunk)

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
walltime	295	300	303	300	300	300
slaves	287	287	288	276	285	285
	259	270	267	258	268	264
	287	287	296	296	290	291
	260	268	264	268	263	265
	289	287	289	287	283	287
	261	262	262	257	255	259
	294	287	289	286	287	289
	256	269	262	262	267	263
					d max	32
					d max %	10.6%

Table B.11: BLOG DMA 1200k 8 slaves (1000 elems/chunk)

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
walltime	64	68	69	66	68	67
slaves	60	62	62	59	62	61
	53	55	47	51	45	50
	60	60	61	60	62	61
	52	48	49	49	52	50
	61	61	63	61	59	61
	55	53	50	52	47	51
	62	61	62	61	61	61
	53	54	49	52	53	52
	60	62	64	62	63	62
	54	54	47	52	52	52
	61	61	58	59	61	60
	53	50	47	50	50	50
	62	60	61	61	61	61
	49	48	48	48	54	49
	61	60	60	60	62	61
	51	48	48	47	54	50
					d max	13
					d max %	19.1%

Table B.12: ODD DMA 1200k 16 slaves (1000 elems/chunk)

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
walltime	334	324	323	325	325	326
slaves	300	296	298	300	302	299
	310	312	313	310	310	311
	308	295	306	302	307	304
	328	317	314	316	317	318
	311	309	313	304	311	310
	321	309	315	312	316	315
	319	310	312	315	317	315
	314	321	314	316	318	317
	322	312	318	313	316	316
	304	301	296	297	302	300
	326	320	318	317	318	320
	321	311	312	312	316	314
	299	297	296	299	301	298
	306	299	301	300	300	301
	318	318	317	320	318	318
	317	309	312	307	313	312
					d max	21
					d max %	6.6%

Table B.13: MAMMO SMA 1200k 16 slaves (1000 elems/chunk)

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
walltime	136	134	136	137	137	136
slaves	123	124	121	124	122	123
	118	115	114	123	122	118
	121	119	114	120	115	118
	126	126	131	129	124	127
	114	123	122	124	118	120
	119	121	111	122	116	118
	118	114	118	116	122	118
	116	123	116	112	112	116
	127	128	129	129	117	126
	126	126	128	132	128	128
	119	121	115	115	116	117
	128	123	130	116	127	125
	128	125	123	123	121	124
	135	127	127	127	131	129
	119	124	121	121	110	119
	111	120	118	121	121	118
					d max	14
					d max %	10.0%

Table B.14: PROB SMA 1200k 16 slaves (1000 elems/chunk)

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
walltime	380	379	380	379	380	380
slaves	288	288	291	292	287	289
	343	346	345	339	347	344
	293	300	295	296	297	296
	287	294	290	291	289	290
	301	312	317	319	314	313
	321	326	319	323	324	323
	282	288	284	289	288	286
	283	285	286	282	280	283
	277	282	283	278	281	280
	352	352	352	356	356	354
	314	313	315	316	317	315
	310	313	313	309	315	312
	309	312	310	307	308	309
	376	372	375	374	375	374
	301	300	304	303	301	302
	315	314	317	315	313	315
					d max	94
					d max %	24.8%

Table B.15: BLOG SMA 1200k 16 slaves (1000 elems/chunk)

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
walltime	68	71	70	68	74	70
slaves	55	58	59	47	55	55
	48	45	55	61	51	52
	54	58	57	63	62	59
	49	60	47	63	60	56
	56	46	48	59	59	54
	58	55	42	53	58	53
	55	59	64	48	63	58
	51	52	57	55	55	54
	64	50	59	51	54	56
	51	56	57	48	57	54
	53	59	52	62	62	58
	58	58	57	62	59	59
	57	59	61	60	65	60
	50	63	59	57	51	56
	59	62	48	61	44	55
	57	54	46	51	57	53
					d max	8
					d max %	12.0%

Table B.16: ODD SMA 1200k 16 slaves (1000 elems/chunk)

B.3 Single-step scheduling

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
walltime	187	188	189	189	189	188
slaves	177	176	177	176	177	177
	180	180	179	180	181	180
	179	180	181	180	180	180
	180	180	180	179	181	180
	176	177	177	178	177	177
	179	178	179	178	178	178
	177	177	177	180	178	178
	180	179	180	179	178	179
	170	170	171	171	172	171
	179	178	178	179	178	178
	182	182	182	183	184	183
	180	180	179	180	181	180
	181	181	181	182	181	181
	179	179	179	179	181	179
	187	188	189	188	187	188
	182	181	180	181	182	181
					d max	17
					d max %	9.0%

Table B.17: MAMMO DMA 1200k 16 slaves (1000 elems/chunk)

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
walltime	93	94	94	93	93	93
slaves	93	92	91	92	92	92
	90	90	89	89	90	90
	85	85	82	85	84	84
	91	93	94	93	92	93
	91	89	89	90	90	90
	88	90	89	90	89	89
	87	84	86	86	85	86
	89	90	89	89	89	89
	88	87	86	87	86	87
	92	92	91	93	92	92
	86	83	85	85	85	85
	87	87	87	86	87	87
	92	91	89	90	90	90
	91	90	89	89	90	90
	86	83	84	85	83	84
	89	90	89	90	89	89
					d max	8
					d max %	9.0%

Table B.18: PROB DMA 1200k 16 slaves (1000 elems/chunk)

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
walltime	328	327	328	328	331	328
slaves	283	286	285	287	289	286
	248	252	252	254	252	252
	282	285	285	287	287	285
	247	251	250	252	251	250
	328	325	329	328	330	328
	250	254	253	255	253	253
	282	284	283	287	286	284
	252	256	255	254	256	255
					d max	78
					d max %	23.7%

Table B.19: BLOG DMA 1200k 8 slaves (1000 elems/chunk)

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
walltime	72	66	66	66	66	67
slaves	63	63	64	64	63	63
	64	65	66	66	65	65
	62	61	60	62	61	61
	64	65	64	65	65	65
	63	63	64	64	64	64
	63	63	64	64	63	63
	63	63	63	64	63	63
	63	64	64	65	64	64
	62	61	61	61	61	61
	64	65	65	66	65	65
	66	63	62	64	63	64
	70	63	64	64	65	65
	62	64	63	64	64	63
	64	64	64	66	64	64
	65	63	63	63	63	63
	65	65	65	66	66	65
					d max	4
					d max %	6.3%

Table B.20: ODD DMA 1200k 16 slaves (1000 elems/chunk)

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
walltime	375	368	366	365	364	368
slaves	323	323	325	322	327	324
	367	365	362	362	364	364
	367	367	363	364	360	364
	339	334	337	331	338	336
	369	365	364	363	364	365
	367	366	364	364	364	365
	333	337	336	334	339	336
	361	364	361	362	362	362
	329	337	336	333	338	335
	332	338	338	334	340	336
	327	327	327	323	327	326
	338	333	336	329	337	335
	330	337	335	333	337	334
	347	344	346	343	346	345
	374	363	365	364	364	366
	324	325	325	324	326	325
					d max	42
					d max %	11.4%

Table B.21: MAMMO SMA 1200k 16 slaves (1000 elems/chunk)

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
walltime	246	216	214	213	213	220
slaves	212	215	213	212	212	213
	182	201	205	202	202	198
	186	189	191	188	187	188
	186	190	192	190	189	189
	184	187	189	187	185	186
	166	169	170	170	168	169
	185	190	190	189	187	188
	210	214	211	212	212	212
	196	198	198	196	196	197
	196	198	199	197	196	197
	196	200	200	198	197	198
	197	200	200	198	197	198
	164	167	168	167	165	166
	153	159	161	159	158	158
	165	168	169	170	167	168
	150	156	157	156	155	155
					d max	58
					d max %	26.3%

Table B.22: PROB SMA 1200k 16 slaves (1000 elems/chunk)

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
walltime	406	406	406	409	405	406
slaves	393	394	393	396	393	394
	266	269	270	271	267	269
	351	348	362	347	346	351
	348	347	346	356	346	349
	404	406	403	406	404	405
	227	232	234	235	231	232
	365	381	364	396	360	373
	347	343	341	351	343	345
	393	395	395	398	393	395
	231	234	234	237	232	234
	341	331	342	331	336	336
	352	348	348	352	349	350
	404	406	404	406	405	405
	217	221	222	224	220	221
	351	348	348	349	346	348
	384	376	377	348	381	373
					d max	184
					d max %	45.3%

Table B.23: BLOG SMA 1200k 16 slaves (1000 elems/chunk)

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
walltime	119	118	115	118	140	122
slaves	104	104	101	107	99	103
	113	114	114	110	135	117
	116	110	116	117	116	115
	100	104	104	103	114	105
	107	109	110	114	116	111
	103	103	105	101	104	103
	108	108	109	109	102	107
	112	115	114	116	116	115
	113	116	110	117	116	114
	109	108	112	109	107	109
	105	107	106	104	101	105
	107	107	108	108	102	106
	102	100	105	102	103	102
	103	105	105	105	101	104
	106	105	107	100	103	104
	103	101	105	101	104	103
					d max	15
					d max %	12.1%

Table B.24: ODD SMA 1200k 16 slaves (1000 elems/chunk)

Appendix C

Variation of Chunk Size

This appendix presents the raw data concerning the variation of chunk size, for the applicable scheduling methods. It is divided into sections, each corresponding to a form of data scheduling. On the tables, `Elems` respresents the number of query elements on a *chunk* and `Average` is calcluated using the data collected in five consecutive runs of the same call. Figures 5.6 and 5.7 in Chapter 5 were plotted using the data presented in this appendix.

C.1 Dynamic Scheduling

Elems	50	100	500	1000	5000	10000
	374	245	198	194	194	192
	362	248	199	192	196	196
	396	246	198	194	194	194
	312	463	199	193	194	194
	459	248	199	193	195	194
Average	381	290	199	193	195	194

Table C.1: MAMMO DMA 1200k (16 slaves)

Elems	50	100	500	1000	5000	10000
	462	372	314	336	345	341
	541	374	310	337	344	342
	523	358	310	334	346	342
	567	396	310	336	346	340
	675	372	313	336	345	341
Average	554	374	311	336	345	341

Table C.2: MAMMO SMA 1200k (16 slaves)

Elems	50	100	500	1000	5000	10000
	535	386	314	299	302	304
	527	388	315	303	300	308
	556	386	315	301	302	303
	476	593	315	301	301	308
	612	387	315	301	299	305
Average	541	428	315	301	301	306

Table C.3: BLOG DMA 1200k (8 slaves)

Elems	50	100	500	1000	5000	10000
	758	429	354	339	408	378
	742	400	366	339	400	398
	667	410	364	341	392	384
	692	399	361	328	383	365
	710	423	364	338	378	383
Average	714	412	362	337	392	382

Table C.4: BLOG SMA 1200k (16 slaves)

Elems	50	100	500	1000	5000	10000
	350	160	102	93	89	93
	384	157	102	93	88	93
	246	527	102	176	89	92
	476	157	100	94	89	91
	565	158	101	92	89	92
Average	404	232	101	110	89	92

Table C.5: PROB DMA 1200k (16 slaves)

Elems	50	100	500	1000	5000	10000
	519	229	141	128	176	188
	518	295	138	127	175	187
	521	280	137	131	177	192
	516	281	138	130	175	196
	516	274	140	131	173	199
Average	518	272	139	129	175	192

Table C.6: PROB SMA 1200k (16 slaves)

Elems	50	100	500	1000	5000	10000
	270	128	68	63	61	67
	259	128	68	64	60	67
	289	127	67	63	61	67
	207	342	69	63	61	68
	350	128	69	65	61	66
Average	275	171	68	64	61	67

Table C.7: ODD DMA 1200k (16 slaves)

Elems	50	100	500	1000	5000	10000
	475	245	80	66	93	108
	478	248	79	65	99	107
	479	247	82	65	95	111
	480	246	75	66	96	110
	487	248	80	64	96	108
Average	480	247	79	65	96	109

Table C.8: ODD SMA 1200k (16 slaves)

C.2 Static Scheduling

Elems	50	100	500	1000	5000	10000
	338	220	206	208	207	193
	361	219	206	207	207	194
	395	219	207	207	206	193
	278	411	207	206	206	193
	424	220	206	207	207	193
Average	359	258	206	207	207	193

Table C.9: MAMMO DMA 1200k (16 slaves)

Elems	50	100	500	1000	5000	10000
	367	335	303	311	343	358
	367	333	301	310	342	352
	368	330	299	309	342	350
	364	344	298	315	346	356
	367	330	301	309	342	366
Average	367	334	300	311	343	356

Table C.10: MAMMO SMA 1200k (16 slaves)

Elems	50	100	500	1000	5000	10000
	476	334	309	322	293	312
	450	335	311	323	294	311
	479	547	311	324	294	310
	373	334	308	322	292	311
	554	336	311	323	292	313
Average	466	377	310	323	293	311

Table C.11: BLOG DMA 1200k (8 slaves)

Elems	50	100	500	1000	5000	10000
	451	357	384	321	332	349
	455	360	383	318	335	348
	452	359	383	317	336	348
	454	355	386	319	332	348
	449	359	381	318	333	349
Average	452	358	383	319	334	348

Table C.12: BLOG SMA 1200k (16 slaves)

Elems	50	100	500	1000	5000	10000
	380	150	589	92	91	96
	353	150	99	92	90	96
	223	226	99	93	89	97
	506	150	99	92	90	96
	221	150	100	92	90	96
Average	337	165	197	92	90	96

Table C.13: PROB DMA 1200k (16 slaves)

Elems	50	100	500	1000	5000	10000
	356	164	140	138	186	204
	348	169	138	139	186	209
	354	175	137	139	185	206
	350	162	135	139	183	207
	354	164	136	141	184	210
Average	352	167	137	139	185	207

Table C.14: PROB SMA 1200k (16 slaves)

Elems	50	100	500	1000	5000	10000
	252	107	67	61	59	72
	243	108	67	62	60	71
	311	109	68	61	60	73
	190	320	66	62	60	74
	337	108	68	61	60	73
Average	267	150	67	61	60	73

Table C.15: ODD DMA 1200k (16 slaves)

Elems	50	100	500	1000	5000	10000
	403	159	66	66	124	141
	403	161	65	91	121	148
	403	167	66	70	115	146
	401	154	66	65	115	147
	402	155	66	68	116	148
Average	402	159	66	72	118	146

Table C.16: ODD SMA 1200k (16 slaves)

C.3 Workpool Scheduling

Elems	50	100	500	1000	5000	10000
	365	336	304	315	343	333
	379	333	302	316	340	331
	365	335	303	316	343	334
	365	334	304	317	340	338
	378	331	302	315	339	338
Average	370	334	303	316	341	335

Table C.17: MAMMO SMA 1200k (16 slaves)

Elems	50	100	500	1000	5000	10000
	359	331	324	329	380	344
	361	329	313	330	377	359
	359	331	320	329	380	358
	367	328	322	331	377	358
	383	338	318	321	376	356
Average	366	331	319	328	378	355

Table C.18: BLOG SMA 1200k (16 slaves)

Elems	50	100	500	1000	5000	10000
	756	157	128	128	171	180
	756	157	130	142	172	181
	751	155	129	134	171	179
	756	156	131	129	171	182
	745	158	129	127	170	180
Average	753	157	129	132	171	180

Table C.19: PROB SMA 1200k (16 slaves)

Elems	50	100	500	1000	5000	10000
	737	311	65	62	97	107
	727	272	62	64	96	110
	732	297	63	61	94	108
	739	290	63	62	95	109
	734	374	64	62	95	109
Average	734	309	63	62	95	109

Table C.20: ODD SMA 1200k (16 slaves)

References

[1] V. Santos Costa, R. Rocha, and L. Damas. The YAP Prolog System. *Journal of Theory and Practice of Logic Programming*, 12(1 & 2):5–34, 2012.

[2] DELL. Dell™ PowerEdge™ R905 – Hardware Owner's Manual, 2007. URL: http://www.manualowl.com/m/Dell/PowerEdge-R905/Manual/189693.

[3] D. Page and A. Srinivasan. ILP: A short look back and a longer look forward. *The Journal of Machine Learning Research*, 4:415–430, 2003.

[4] P. M. Nugues. *An Introduction to Language Processing with Perl and Prolog: An Outline of Theories, Implementation, and Application with Special Consideration of English, French, and German*. Springer, 2006.

[5] W. C. Benton and C. N. Fischer. Interactive, scalable, declarative program analysis: from prototype to implementation. In *Proceedings of the 9th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 13–24. ACM, 2007.

[6] D. Li and D. Liu. *A fuzzy Prolog database system*. John Wiley and Sons Inc, 1990.

[7] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[8] Cheng-Tao Chu, S. Kyun Kim, Yi-An Lin, Y. Yu, G. Bradski, A. Y. Ng, and K. Olukotun. Map-Reduce for Machine Learning on Multicore. In *Advances in Neural Information Processing Systems 19*, pages 281–288. MIT Press, 2007.

[9] J. W. Lloyd. Practical Advantages of Declarative Programming. In *Joint Conference on Declarative Programming, GULP-PRODE*, volume 1, pages 18–30, 1994.

[10] Stefano Ceri, Georg Gottlob, and Letizia Tanca. What You Always Wanted to Know About Datalog (And Never Dared to Ask). *Knowledge and Data Engineering, IEEE Transactions on*, 1(1):146–166, 1989.

[11] Patricia Hill and John Wylie Lloyd. *The Gödel programming language*. The MIT Press, 1994.

[12] Alain Colmerauer and Philippe Roussel. The birth of Prolog. In *History of programming languages—II*, pages 331–367. ACM, 1996.

[13] J. Wielemaker, T. Schrijvers, M. Triska, and T. Lager. SWI-Prolog. *Journal of Theory and Practice of Logic Programming*, 12(1 & 2):67–96, 2012.

[14] M. Carlsson and P. Mildner. SICStus Prolog - the first 25 years. *Theory and Practice of Logic Programming*, 12(1-2):35–66, 2012.

[15] C. Green. Application of theorem proving to problem solving. Technical report, Defense Technical Information Center Document, Stanford University, 1969.

[16] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, 1983.

[17] H. Aït-Kaci. *Warren's Abstract Machine – A Tutorial Reconstruction*. The MIT Press, 1991.

[18] G. Gupta, E. Pontelli, K. Ali, M. Carlsson, and M. V. Hermenegildo. Parallel Execution of Prolog Programs: A Survey. *ACM Transactions on Programming Languages and Systems*, 23(4):472–602, 2001.

[19] V. Santos Costa. Optimising Bytecode Emulation for Prolog. In *Principles and Practice of Declarative Programming*, number 1702 in LNCS, pages 261–267. Springer-Verlag, 1999.

[20] R. Rocha, F. Silva, and V. Santos Costa. YapOr: an Or-Parallel Prolog System Based on Environment Copying. In *Portuguese Conference on Artificial Intelligence*, number 1695 in LNAI, pages 178–192. Springer-Verlag, 1999.

[21] R. Rocha, F. Silva, and V. Santos Costa. On a Tabling Engine That Can Exploit Or-Parallelism. In *International Conference on Logic Programming*, number 2237 in LNCS, pages 43–58. Springer-Verlag, 2001.

[22] V. Santos Costa, K. Sagonas, and R. Lopes. Demand-Driven Indexing of Prolog Clauses. In *International Conference on Logic Programming*, number 4670 in LNCS, pages 395–409. Springer-Verlag, 2007.

[23] R. Rocha. On Improving the Efficiency and Robustness of Table Storage Mechanisms for Tabled Evaluation. In *International Symposium on Practical Aspects of Declarative Languages*, number 4354 in LNCS, pages 155–169. Springer-Verlag, 2007.

[24] R. Rocha, F. Silva, and V. Santos Costa. Dynamic Mixed-Strategy Evaluation of Tabled Logic Programs. In *International Conference on Logic Programming*, number 3668 in LNCS, pages 250–264. Springer-Verlag, 2005.

[25] S. Papadimitriou and J. Sun. DisCo: Distributed Co-clustering with Map-Reduce: A Case Study Towards Petabyte-Scale End-to-End Mining. In *International Conference on Data Mining*, pages 512–521. IEEE Computer Society, 2008.

[26] C. Miceli, M. Miceli, S. Jha, H. Kaiser, and A. Merzky. Programming Abstractions for Data Intensive Computing on Clouds and Grids. In *IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 478–483. IEEE Computer Society, 2009.

[27] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A Comparison of Approaches to Large-Scale Data Analysis. In *ACM International Conference on the Management of Data*, pages 165–178. ACM, 2009.

[28] S. Pallickara, J. Ekanayake, and G. Fox. Granules: A Lightweight, Streaming Runtime for Cloud Computing with Support, for Map-Reduce. In *International Conference on Cluster Computing and Workshops*, pages 1–10. IEEE Computer Society, 2009.

[29] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: A Runtime for Iterative MapReduce. In *ACM International Symposium on High Performance Distributed Computing*, pages 810–818. ACM, 2010.

[30] S. Plimpton and K. Devine. MapReduce in MPI for Large-scale Graph Algorithms. *Parallel Computing*, 37(9):610–632, 2011.

[31] D. Borthakur for Apache Software Foundation. The Hadoop Distributed File System: Architecture and Design, 2007. URL: http://hadoop.apache.org/docs/r0.18.0/hdfs_design.pdf.

[32] A. Srinivasan, T. A. Faruquie, and S. Joshi. Data and task parallelism in ILP using MapReduce. *Machine Learning*, 86(1):141–168, 2012.

[33] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Symposium on Mass Storage Systems and Technologies*, pages 1–10. IEEE Computer Society, 2010.

[34] A. Srinivasan. *The Aleph Manual*, 2004. URL: http://www.cs.ox.ac.uk/activities/machlearn/Aleph.

[35] Message Passing Interface Forum. URL: http://www.mpi-forum.org.

[36] J. Wielemaker. Native Preemptive Threads in SWI-Prolog. In *International Conference on Logic Programming*, number 2916 in LNCS, pages 331–345. Springer-Verlag, 2003.

[37] E. L. Lusk W. Gropp and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*, volume 1. MIT press, 1999.

[38] Torsten Hoefler, Andrew Lumsdaine, and Jack Dongarra. Towards Efficient MapReduce Using MPI. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 240–249. Springer, 2009.

[39] LAM/MPI Parallel Computing. URL: http://www.lam-mpi.org.

[40] G. Bosilca T. Angskun J. J. Dongarra J. M. Squyres V. Sahay P. Kambadur B. Barrett A. Lumsdaine R. H. Castain D. J. Daniel R. L. Graham T. S. Woodall E. Gabriel, G. E. Fagg. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 97–104. Springer, 2004.

[41] B. Toonen N. T. Karonis and I. Foster. MPICH-G2: a Grid-enabled implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing*, 63(5):551–563, 2003.

[42] R. K. Malladi. Using Intel® VTune™ Performance Analyzer Events/Ratios & Optimizing Applications, 2009.

[43] Intel Software. Intel® VTune™ Amplifier XE 2013, 2013. URL: http://software.intel.com/en-us/intel-vtune-amplifier-xe.