

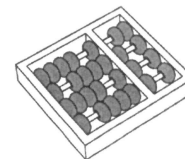


Rodolfo Guilherme Wottrich

**“Loop Parallelization in the Cloud
Using OpenMP and MapReduce”**

***“Paralelização de Laços na Nuvem
Usando OpenMP e MapReduce”***

**CAMPINAS
2014**



University of Campinas
Institute of Computing

Universidade Estadual de Campinas
Instituto de Computação

Rodolfo Guilherme Wottrich

“Loop Parallelization in the Cloud Using OpenMP and MapReduce”

Supervisor: Prof. Dr. Guido Costa Souza de Araújo
Orientador(a):

Co-Supervisor: Prof. Dr. Rodolfo Jardim de Azevedo
Co-orientador(a):

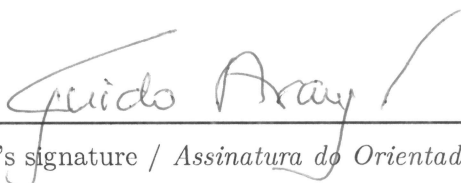
“Paralelização de Laços na Nuvem Usando OpenMP e MapReduce”

MSc Dissertation presented to the Post Graduate Program of the Institute of Computing of the University of Campinas to obtain a Master degree in Computer Science.

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Computação da Universidade Estadual de Campinas para obtenção do título de Mestre em Ciência da Computação.

THIS VOLUME CORRESPONDS TO THE FINAL VERSION OF THE DISSERTATION DEFENDED BY RODOLFO GUILHERME WOTTRICH, UNDER THE SUPERVISION OF PROF. DR. GUIDO COSTA SOUZA DE ARAÚJO.

ESTE EXEMPLAR CORRESPONDE À VERSÃO FINAL DA DISSERTAÇÃO DEFENDIDA POR RODOLFO GUILHERME WOTTRICH, SOB ORIENTAÇÃO DE PROF. DR. GUIDO COSTA SOUZA DE ARAÚJO.


Supervisor's signature / Assinatura do Orientador(a)

CAMPINAS
2014

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Maria Fabiana Bezerra Muller - CRB 8/6162

W914L Wottrich, Rodolfo Guilherme, 1990-
Loop parallelization in the cloud using OpenMP and MapReduce / Rodolfo
Guilherme Wottrich. – Campinas, SP : [s.n.], 2014.

Orientador: Guido Costa Souza de Araújo.

Coorientador: Rodolfo Jardim de Azevedo.

Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de
Computação.

1. Programação paralela (Computação). 2. Computação em nuvem. 3.
OpenMP (Programação paralela). 4. MapReduce (Programação paralela
distribuída). I. Araújo, Guido Costa Souza de, 1962-. II. Azevedo, Rodolfo Jardim
de, 1974-. III. Universidade Estadual de Campinas. Instituto de Computação. IV.
Título.

Informações para Biblioteca Digital

Título em outro idioma: Paralelização de laços na nuvem usando OpenMP e MapReduce

Palavras-chave em inglês:

Parallel programming (Computer science)

Cloud computing

OpenMP (Parallel programming)

MapReduce (Distributed parallel programming)

Área de concentração: Ciência da Computação

Titulação: Mestre em Ciência da Computação

Banca examinadora:

Guido Costa Souza de Araújo [Orientador]

Alexandro José Baldassin

Luiz Fernando Bittencourt

Data de defesa: 09-04-2014

Programa de Pós-Graduação: Ciência da Computação

TERMO DE APROVAÇÃO

Defesa de Dissertação de Mestrado em Ciência da Computação, apresentada pelo(a) Mestrando(a) **Rodolfo Guilherme Wottrich**, aprovado(a) em **09 de abril de 2014**, pela Banca examinadora composta pelos Professores Doutores:



Prof(a). Dr(a). **Alexandre José Baldassin**
Titular



Prof(a). Dr(a). **Luiz Fernando Bittencourt**
Titular



Prof(a). Dr(a). **Guido Costa Souza de Araújo**
Presidente

Loop Parallelization in the Cloud Using OpenMP and MapReduce

Rodolfo Guilherme Wottrich¹

April 09, 2014

Examiner Board/*Banca Examinadora*:

- Prof. Dr. Guido Costa Souza de Araújo (Supervisor/*Orientador*)
- Prof. Dr. Alexandro José Baldassin
Department of Statistics, Applied Mathematics and Computing - UNESP
- Prof. Dr. Luiz Fernando Bittencourt
Institute of Computing - UNICAMP
- Prof. Dr. Edmundo Roberto Mauro Madeira - Substitute
Institute of Computing - UNICAMP
- Prof. Dr. Bruno de Carvalho Albertini - Substitute
Polytechnic School - USP

¹Financial support: CAPES scholarship 03/2012 - 01/2013 and FAPESP scholarship (process 2012/17278-9) 02/2013 - 02/2014

Abstract

The pursuit of parallelism has always been an important goal in the design of computer systems, driven mainly by the constant interest in reducing program execution time. Parallel programming is an active research area, which has grown in interest due to the emergence of multicore architectures. On the other hand, harnessing the large computing and storage capabilities of the cloud and its desirable flexibility and scaling features offers a number of interesting opportunities to address some relevant research problems in scientific computing. Unfortunately, in many cases the implementation of applications on the cloud demands specific knowledge of parallel programming interfaces and APIs, which may become a burden when programming complex applications. To overcome such limitations, in this work we propose OpenMR, an execution model based on the syntax and principles of the OpenMP API which eases the task of programming distributed systems (i.e. local clusters or remote cloud). Specifically, this work addresses the problem of performing loop parallelization, using OpenMR, in a distributed environment, through the mapping of loop iterations to MapReduce nodes. By doing so, the cloud programming interface becomes the programming language itself, freeing the developer from the task of worrying about the details of distributing workload and data. To assess the validity of the proposal, we modified benchmarks from the SPEC OMP2012 suite to fit the proposed model, developed other I/O-bound toy benchmarks and executed them in two settings: (a) a computer cluster locally available through a standard LAN; and (b) clusters remotely available through the Amazon AWS services. We compare the results to the execution using OpenMP in an SMP architecture and show that the proposed parallelization technique is feasible and demonstrates good scalability.

Resumo

A busca por paralelismo sempre foi um importante objetivo no projeto de sistemas computacionais, conduzida principalmente pelo constante interesse na redução de tempos de execução de aplicações. Programação paralela é uma área de pesquisa ativa, na qual o interesse tem crescido devido à emergência de arquiteturas *multicore*. Por outro lado, aproveitar as grandes capacidades de computação e armazenamento da nuvem e suas características desejáveis de flexibilidade e escalabilidade oferece várias oportunidades interessantes para abordar problemas de pesquisa relevantes em computação científica. Infelizmente, em muitos casos a implementação de aplicações na nuvem demanda conhecimento específico de interfaces de programação paralela e APIs, o que pode se tornar um fardo na programação de aplicações complexas. Para superar tais limitações, neste trabalho propomos OpenMR, um modelo de execução baseado na sintaxe e nos princípios da API OpenMP que facilita a tarefa de programar sistemas distribuídos (isto é, *clusters* locais ou a nuvem remota). Especificamente, este trabalho aborda o problema de executar a paralelização de laços, usando OpenMR, em um ambiente distribuído, através do mapeamento de iterações do laço para nós MapReduce. Assim, a interface de programação para a nuvem se torna a própria linguagem, livrando o desenvolvedor da tarefa de se preocupar com detalhes da distribuição de cargas de trabalho e dados. Para avaliar a validade da proposta, modificamos *benchmarks* da suite SPEC OMP2012 para se encaixarem no modelo proposto, desenvolvemos outros *toy benchmarks* que são *I/O-bound* e executamos-os em duas configurações: (a) um *cluster* de computadores disponível localmente através de uma LAN padrão; e (b) *clusters* disponíveis remotamente através dos serviços Amazon AWS. Comparamos os resultados com a execução utilizando OpenMP em uma arquitetura SMP e mostramos que a técnica de paralelização proposta é factível e demonstra boa escalabilidade.

Contents

Abstract	ix
Resumo	xi
1 Introduction	1
1.1 Motivation	1
1.2 Research Goal	2
1.3 Contributions	2
2 Background	4
2.1 Parallel Computing	4
2.2 MPI	6
2.3 OpenMP	7
2.3.1 <code>parallel</code> Region Construct	7
2.3.2 Work-sharing Constructs	8
2.3.3 Data Sharing	9
2.3.4 Synchronization Constructs	10
2.3.5 Runtime Library Functions	11
2.4 MapReduce	11
2.4.1 Workflow Overview	12
2.4.2 The Word Count Example	13
2.4.3 Distributed File Systems	13
2.4.4 Advanced Features	15
2.5 Related Work	16
2.5.1 MapReduce Applications	16
2.5.2 MapReduce Framework Optimizations	16
2.5.3 Data Movement/Distribution in the Cloud	17
2.5.4 Parallel Programming Models	17

3	The OpenMR Execution Model	22
3.1	The Concept	22
3.2	The Syntax	25
3.2.1	The <code>mapreduce</code> Construct	25
3.2.2	The <code>input/output</code> Clauses	25
3.2.3	The <code>data</code> construct	26
3.3	Preparing the MapReduce Job	27
3.4	Applications	31
4	Methodology/Experimental Results	33
4.1	Compiler Infrastructure	33
4.2	Benchmarks	36
4.3	Experimental Setup	39
4.4	Results	42
4.4.1	SPEC Benchmarks	42
4.4.2	Toy Benchmarks	52
5	Conclusions	56
5.1	Future Work	56
	Bibliography	58

List of Tables

2.1	Comparison of the discussed parallel programming models	21
4.1	Description of the benchmarks in the SPEC OMP2012 suite (adapted from [42])	36

List of Figures

2.1	OpenMP's <code>parallel</code> construct sample	8
2.2	Output after executing the code from figure 2.1, on a four-core SMP	8
2.3	OpenMP's <code>for</code> construct sample	9
2.4	The execution of the Word Count example	14
3.1	OpenMP's trip count parallelization	23
3.2	OpenMR's execution model	23
3.3	Mapping between iterations and executing nodes, for a four-node setup . .	24
3.4	The <code>mapreduce</code> construct	25
3.5	The <code>input/output</code> clauses	26
3.6	Examples of the <code>data</code> construct	27
3.7	Example of OpenMR code before compilation	29
3.8	Example of OpenMR code after compilation (pseudocode)	29
3.9	Example of Map function (pseudocode)	30
3.10	Example of Reduce function (pseudocode)	31
4.1	The compilation flow in GCC (adapted from [44])	34
4.2	Execution times of SPEC benchmarks with OpenMP	42
4.3	Execution of 358.botsalgn with OpenMR and Amazon EMR	44
4.4	Execution of 358.botsalgn with OpenMR and Amazon EMR + additional overheads	45
4.5	Execution of 372.smithwa with OpenMR and Amazon EMR	47
4.6	Execution of 372.smithwa with OpenMR and Amazon EMR + additional overheads	48
4.7	Execution times of 358.botsalgn and 372.smithwa with OpenMR in a local cluster, with 56 processing cores	49
4.8	Speedups of 358.botsalgn and 372.smithwa with OpenMR in a local cluster, relative to the varying amount of baseline's threads	51
4.9	Execution times of toy benchmarks in a local, 4-core machine	52
4.10	Absolute execution times of toy benchmarks in Amazon EMR	53

4.11 Speedups of toy benchmarks in Amazon EMR relative to local processing .	54
--	----

Chapter 1

Introduction

The exploitation of parallelism has always been a critical factor in the pursuit of increasingly better execution performance. That is because we always wanted to take advantage of the available hardware in its fullness and sometimes barriers forced the exploration of new forms of parallelism.

1.1 Motivation

Throughout the years, different levels of parallelism have been implemented based on different levels of granularity of the computation. Some of the solutions were developed to be totally transparent to the user/programmer, specially the ones which only use hardware mechanisms like *Instruction-Level Parallelism* (ILP): ILP is achieved exclusively by hardware mechanisms implemented in the microarchitecture and enhanced by code optimizations implemented in the compiler.

Parallelism has also been explored in the form of multicore CPUs that feature effectively replicated pipelines and allow for distribution of processes and threads of a system, relieving the workload from what otherwise would be just a uniprocessor. The existence of co-processors and accelerators, like *Graphics Processing Units* (GPUs), also represents a widely explored form of parallelism, often through the so-called *data-level parallelism*.

A recent form of parallelism currently in evidence is the execution of parallel programs in distributed-memory systems. This comprises systems in which: 1) a processing machine can be viewed as a CPU-memory pair, where the memory associated to the CPU is directly accessible by that CPU only and no other, and 2) multiple machines communicate through an interconnect network to cooperatively solve parallel programs. Such systems are often called computer clusters and are normally composed of multiple commodity machines, each one commonly denominated a *node*, that communicate with each other by means of a common *Local Area Network* (LAN).

The growing availability of such clusters is specially important because of the advent of *cloud computing*. The cloud, as it is referred to, is a general concept that describes various kinds of services offered through real-time communication networks such as the Internet, often thought as clusters with a large amount of cooperating computers and/or storage. The denomination “cloud” clearly refers to the sense of abstraction that comes along with it, as it is common for the user not to be aware of the technical details that permeate such services.

However, still today it might be difficult to harness the computational potential of the cloud, since it is clear that a lot of knowledge and effort is required to implement the solution of even simple problems in this paradigm. Not only the developer needs to master the programming models, knowing how to rewrite code to deal with communication among nodes and treating I/O, implementing fault tolerance and workload balancing, but also spend time setting up the cloud environment, optimizing the cluster and its communication with the storage, and so on.

1.2 Research Goal

The discussion above leads us to the following question:

Is it possible to simplify the utilization of the resources available in the cloud in such a way that programmers can parallelize software with little effort, reusing most of the initial solutions developed in a local machine?

In this context, the goal of this dissertation is to propose a new execution model, an extension to the familiar OpenMP API, to enable the usage of highly-parallel, distributed processing in remote clusters of machines. Therefore we propose OpenMR, a framework for cloud programming based on OpenMP and MapReduce, which additionally provides desirable features such as fault tolerance and workload balancing.

OpenMR is evaluated through the implementation of equivalent versions of applications from the SPEC OMP2012 benchmark suite [42] and through toy benchmarks designed to represent some applications with simple mathematical computation, simulating the target code for a compiler implementing OpenMR. We ran our experiments on different data communication scenarios and evaluated the resulting performance and data communication overheads.

1.3 Contributions

The main contributions of this work are the following:

- A novel execution model for parallel programming with distributed processing in the cloud which extends the functionality of OpenMP to the cloud;
- A methodology for the mapping between loop iterations in a sequential programming language to a MapReduce execution model in the cloud;
- The proposal of constructs to abstract the storage of data in the cloud, bringing the cloud one step closer to be a part of a fully-integrated, cloud-aware stack of compute capabilities. This aspect is presented as one of the possible future works related to this project.

This document is organized as follows: chapter 2 covers background concepts on which this work was based and compares this proposal to other similar and related works, chapter 3 elaborates on the main aspect of this work, the proposal of an execution model designed to provide easy usage of the cloud, chapter 4 shows the experimental setup and results obtained from the assessment of such a model, and chapter 5 draws conclusions about the contributions of this work and points out possibilities for future continuation of this specific line of research.

Chapter 2

Background

This chapter defines and details some of the most important background concepts related to this work.

2.1 Parallel Computing

In Computer Science, parallelism is the execution of multiple computations simultaneously, in parallel, taking advantage of redundant hardware. It is one of the most important fields of study in this area, given that it comprises various hardware/software techniques which aim at improving performance. The basic principle of parallel computing is that large problems often exhibit the possibility of being split into different smaller chunks, which might be executed concurrently, thus saving much of the original execution time.

Research and development of solutions for parallel computing are not a recent trend. Regarding the different kinds of parallelism found on computer architectures, a simple way to categorize such systems was proposed by Flynn in 1966 [23, 24], in what later became known as Flynn’s Taxonomy. Back then, the so-called *parallel systems* have been divided based on the instruction and data streams. The categories were:

- Single Instruction Single Data (SISD): basically the uniprocessor as we know it, executing a single instruction at a time, over a a single data stream;
- Single Instruction Multiple Data (SIMD): different sets of data being processed by multiple processors in parallel, exploring *data-level parallelism*. May be extended to Single Program Multiple Data (SPMD), in cases where parallel processors execute a single program but at potentially independent points;
- Multiple Instruction Single Data (MISD): very uncommon architecture, almost no commercial processor exhibiting this characteristic has ever been developed [31],

because very few applications fit such a model;

- Multiple Instruction Multiple Data (MIMD): each processor fetches its own instruction and data streams, exploiting *thread-level parallelism*.

Today, most computer systems are hybrids of these categories, in the sense that most can execute, and/or possess co-processors which are capable of executing, code from more than one of them. Still, because of its simplicity, this taxonomy remains as a basic paradigm for parallel computing.

Another seminal work on parallel computing is what we call *Amdahl's Law* [2], which is an analysis on the limits of computation in general. In his work, Amdahl shows that the theoretical peak of speedup when parallelizing an application is determined by the portion of the application which is parallelized. For instance, if only 50% of the application execution is to be parallelized, even if the parallel portion could theoretically be executed in no time (i.e. time tending to zero), the remaining 50% would still be executed sequentially and therefore a maximum speedup of 2x would be achieved. Latecomers would translate the literal description of Amdahl's argument as follows:

$$Speedup = \frac{1}{r_s + \frac{r_p}{n}}$$

where $r_s + r_p = 1$, r_p represents the portion of the program to be parallelized, r_s represents the portion of the program to remain sequential and n represents the number of processors in which the parallel portion will be executed.

Since early machines, different ways of exploiting the potential of parallelism have been reported. One of the first forms of parallelism was at bit level, since increasing the number of bits of the data words effectively reduced the number of instructions necessary to perform certain arithmetic operations on data which did not fit inside words of a narrow datapath [13]. Soon, microarchitectures began to explore ILP, by implementing mechanisms to issue and execute multiple instructions at a time, out of program order, taking advantage of redundant execution units. That was the case for the so-called *Tomasulo algorithm* [59].

Since then, parallelism has evolved to many forms but for many years it was mainly employed in highly-expensive, high-end machines aiming at supercomputing. However, years ago, due to physical restrictions, transistor frequency scaling methodologies (e.g. *Dennardian Scaling* [19]) started to face problems (such as leakage and physical under-voltage limits) which turned it impossible for manufacturers to continue the development of processors with increasingly higher clock rates [58, 60]. Later, it also led to further barriers like the *power wall* [35, 50] and the *utilization wall*, or *dark silicon* [22, 58, 60].

This, in turn, led to the creation of chips with replicated processing cores. One of the main consequences of this shift in the architectural trends was that parallel programming turned into one of the most important computing paradigms.

Today, different types of parallelism have been developed based on hardware and/or software techniques. Data-level parallelism is one of them. Several programming models regard data-level parallelism, such as OpenCL and NVIDIA's CUDA, which take advantage of massively-parallel accelerators (namely, GPUs) to perform SIMD operations. Distributed processing in computer clusters (the cloud included) is another case in which not only SIMD, but also MIMD processing may be performed.

Unfortunately, many of the parallelization techniques in software demand specific knowledge and technical understanding from the programmer/user, not to mention the effort and time consumption associated to the tasks of writing parallel code in those cases. That is why there are solutions like OpenMP, which intends to abstract implementation details of a parallelization technique in order to turn it possible to write parallel code with little programming effort.

2.2 MPI

Message-Passing Interface (MPI) is an Application Programming Interface (API), coupled with protocol and semantics specifications [29]. It lists function calls which enable programmers to write parallel programs in a distributed-memory processing paradigm [48], using a message-passing paradigm. With MPI, programs run in different hosts (CPUs), each one associated to its own memory, and communicate with each other through messages sent over an interconnection network, such as a LAN. Usually, developers write MPI programs in order to have multiple systems to cooperate in a computer cluster to solve a problem.

MPI is the de-facto standard for distributed processing. Actual distributed-memory supercomputers often run such programs [53, 56], because of its scalability and performance rates.

For the average developer, however, programming with MPI can become a problem. The API was thought to give the developer complete control over implementation details of the communication mechanisms, thus bringing much complexity to the process of developing a distributed application. The execution of an application with MPI usually involves steps [48] like initializing/destructuring MPI resources and using functions to send and receive the messages among processing nodes (`MPI_Send` and `MPI_Receive`), which need to be specifically thought to occur coordinately and with matching parameters (like message size and source/destination). Additionally, the programmer must consider using features like collective communication constructs (broadcast, reduction) and data distri-

bution schemes (scatter, gather). Finally, the developer must deal with all different MPI data types, parallel logic and still some desirable features as fault-tolerance and workload balancing are not provided at all, requiring lots of effort to be implemented.

2.3 OpenMP

OpenMP (short for *Open MultiProcessing*) is also an API composed of a set of compiler directives and runtime library routines to perform parallel calculations on shared-memory multiprocessors (SMP) [11]. OpenMP superseded the attempts for establishing a standard called ANSI X3H5 [51] for distributed-memory message-passing systems, which had scalability problems [15] and soon proved itself much inferior to OpenMP.

OpenMP was designed and published as a set of specifications which serve as guidelines for the development of compliant compilers. These specifications are discussed and reviewed by a board composed of members from many of the most important companies on *High Performance Computing* (HPC) in the world.

The main motivation behind the creation of OpenMP was to bring parallelism to commodity-level applications. It was targeted to developers who wanted to quickly parallelize existing code [11], supporting a broad range of applications. Today, all the most important C/C++ and Fortran compilers (the languages OpenMP supports) have their OpenMP implementations, like GCC, Intel's and, more recently, LLVM/Clang [4].

For a better understanding on how OpenMP works, we show below the basic syntax of OpenMP's API [3, 7, 36, 38, 48] and code samples.

In C/C++, all OpenMP constructs are `#pragma` directives, except for the library functions which provide information about the number and IDs of threads. These directives implement a very important feature: if a specific compiler does not support OpenMP, they are automatically ignored by the C preprocessor and the code compiles and executes sequentially, exactly like if OpenMP clauses were not even present. This, in turn, considerably simplifies the task of parallelizing an existing piece of code, requiring from the programmer only the placement of certain annotations in the right places in the code.

2.3.1 parallel Region Construct

The `parallel` region is the fundamental OpenMP construct, as exemplified in figure 2.1. It defines a region of code which will be executed by multiple threads in potentially multiple processing cores. The compiler will translate this construct into calls to functions of the runtime library which will take care of thread creation and manipulation, splitting data accordingly and effectively extracting parallelism from the original piece of code.

```
#pragma omp parallel
{
    printf("Hello\n");
}
```

Figure 2.1: OpenMP's `parallel` construct sample

The OpenMP runtime offers the chance to manually determine the number of threads created by the `parallel` region. This can be done through specific clauses that may be included in the `#pragma` directive or by setting that number through a function call or environment variable. When unspecified, the runtime creates a number of threads equivalent to the number of CPUs (that includes individual cores) in the system. Thus, the code in figure 2.1 (which does not state the number of threads) would yield the following output when run on a four-core SMP:

```
Hello
Hello
Hello
Hello
```

Figure 2.2: Output after executing the code from figure 2.1, on a four-core SMP

The output in figure 2.2 is just a simple example; in reality, a program in which multiple threads compete for the same resource may cause race conditions that affect the output (this problem may be solved by using synchronization mechanisms). In C/C++, `printf` is *thread-safe*, which means that its data structures and resources are guaranteed to be safely executed by multiple threads at a time.

2.3.2 Work-sharing Constructs

Work-sharing constructs are the OpenMP way to distribute the execution of the associated `parallel` region among the threads that encounter these constructs [3], assigning individual work to each one of them. OpenMP defines the following work-sharing constructs:

- **for** (C/C++) / **do** (Fortran): shares loop iterations across the threads of the region. Every thread that encounters this construct becomes responsible for processing a fraction of the original loop trip count, in a type of data-level parallelism. This may be seen as a form of SPMD parallelism;

- **sections**: breaks the work into separate, discrete sections which are each executed by a thread;
- **single**: determines that a piece of code should be executed by only one thread in the team; useful when dealing with code which is not thread safe (like I/O operations);
- **workshare** (Fortran only): divides the work into separate units, each of which is executed only once.

The **task** construct, defined since version 3.0 of OpenMP's specification document [3], is not explicitly treated as a work-sharing construct but may be considered one. It defines a task (based on the the piece of code it comprises and its data) to be executed immediately by any of the available threads or deferred for later execution.

The **for/do**, **sections** and **workshare** constructs are eligible for condensed use with the **parallel** construct, as a convenience.

Figure 2.3 is an example of a **for** work-sharing construct. In this case, the contents of the array **b[]** are doubled and assigned to the array **a[]**. The OpenMP construct splits the loop between the threads, each of which will be responsible for executing a portion of the **N** iterations (clearly exploiting a form of data-level parallelism, since every thread will execute the same operations over a different set of data).

```
#pragma omp parallel for private(i)
for(i = 0; i < N; i++)
{
    a[i] = 2*b[i];
}
```

Figure 2.3: OpenMP's **for** construct sample

2.3.3 Data Sharing

Since OpenMP executes on an SMP machine, all threads running the same parallel region will have access to the data that they are processing. Nonetheless, in most cases, a portion of the data will need to be copied for each thread to use its own private copy. The code in figure 2.3 is an example. In that case, the loop will be split among threads and each thread will need its own version of the variable **i**. If that would not occur, the execution of the loop would face data race conditions which eventually could affect the correctness of the program and its outcome. Therefore, OpenMP provides a set of clauses to complement its directives with information about data sharing. Some of them are:

- **shared**: all threads share the specified variables. Great for input variables, as reading them concurrently causes no harm. Writing on them may cause race conditions;
- **private**: each thread has its private copy of the specified variable. Useful for data that are updated by each individual thread, like the induction variables of a loop;
- **default**: specifies the default behavior of data sharing. If a variable was not declared in other data-sharing clauses, it will be implicitly declared as determined by the **default** clause. If no **default** clause is present, the default behavior is to have all undeclared variables as **shared**.
- **firstprivate**: the variable is **private** and the current value of the shared version is copied to the private version as the threads enter the constructs;
- **lastprivate**: likewise, but causes the original variable to be updated when the threads finish executing the construct;
- **reduction**: like **lastprivate**, but applying a certain operation to all the private versions, causing a reduction to a single copy.

2.3.4 Synchronization Constructs

In many cases, the threads compete for resources which must be coordinately used in order for the correctness of the program to be assured. That is the case, for instance, of threads which print output to the screen or need to update the value of a shared variable. Such situations create race conditions. To address these situations, OpenMP offers ways to synchronize the execution of threads through synchronization constructs. Some of OpenMP's most important synchronization constructs are:

- **critical**: treats the associated structured block as a critical section, allowing only a single thread to execute that part of the code a time;
- **barrier**: when reached by a thread, this construct suspends its execution until all other threads complete execution until that same point;
- **taskwait**: the same as a barrier, but regarding **task** constructs;
- **flush**: makes the temporary view of memory of a thread consistent with memory, enforcing an order on the memory operations. May explicitly list variables or implicitly do the memory operations to all variables in the scope.

2.3.5 Runtime Library Functions

The OpenMP Specification reference describes yet another feature of OpenMP, runtime library functions which offer information about the execution environment and useful utilities. These functions are divided in three categories:

- Execution environment routines: functions that provide and manipulate information about threads. Examples: `omp_set_num_threads` and `omp_get_num_threads`, which retrieve and set the number of threads executing, `omp_get_thread_num`, which provides the ID associated to a specific thread and `omp_set_dynamic`, which enables or disables the dynamic adjustment of the number of threads;
- Lock routines: functions that can be used for synchronization. The locks are represented by OpenMP lock variables. Examples: `omp_init_lock`, `omp_destroy_lock` and `omp_set_lock`;
- Timing routines: functions to implement a wall clock timer. There are two of them: `omp_get_wtime` and `omp_get_wtick`.

In this work, we associate the programming model of OpenMP to SPMD processing in the cloud, taking the model of the `for / do` work-sharing construct to its logical extension.

2.4 MapReduce

MapReduce [17] is a framework for distributed processing based on concepts from functional programming. It aims at providing programming support to address problems in the area commonly called *Big Data*, in which sets of data in the order of petabytes, which could not fit the memory of a single processing node, are to be processed in a distributed-memory context. In the last few years, MapReduce has been widely used to achieve interesting results in HPC. For instance, genome indexing, protein alignment, genotype calling and many other biological applications are well suited for MapReduce [37, 40]. Another example is the work of Deligiannis et al., in which diagnoses of heart diseases through the analysis of medical records are performed with MapReduce [18]. Machine learning algorithms also suit MapReduce's model well and may be used for computer vision, compiler optimizations and other applications [27, 57]. Also, many large companies have utilized MapReduce to solve their business problems at affordable costs [12].

The computation in MapReduce is based on the concepts of *Map* and *Reduce functions*, therefore its name. These functions are written by the developer and perform the actual computation in the job.

Conceptually, in functional languages, the map function applies an operation over a set of data, individually [12]. This might be seen as follows:

$$\text{map}(\{1, 2, 3, 4\}, (\times 2)) \rightarrow \{2, 4, 6, 8\}$$

In the context of MapReduce, analogously, Map is a function that takes as input a key/value pair, perform some sort of calculation over it, and produces an intermediate output, which is composed of one or more pertinent key/pair values [17].

The concept of reduction in functional languages, in turn, regards to applying an operation to all the input data iteratively. Then:

$$\text{reduce}(\{1, 2, 3, 4\}, (\times)) \rightarrow \{24\}$$

In MapReduce, the Reduce function takes as input a list of key/pair values with the same key and performs some more operations on it, typically outputting a single value or a single key/value pair as result.

In general, the logical definition for the Map and Reduce functions in the context of MapReduce is as follows:

$$\begin{aligned} \text{map}(k1, v1) &\rightarrow \text{list}(k2, v2) \\ \text{reduce}(k2, \text{list}(v2)) &\rightarrow \text{list}(k3, v3) \end{aligned}$$

The most used implementation for MapReduce is currently Apache Hadoop [25, 62].

2.4.1 Workflow Overview

Using the above defined concepts, MapReduce executes distributed workloads as follows [12, 17, 62]:

- The MapReduce framework starts running the job by designating the Map role to processors and assigning a set of key/value pairs to each one of them;
- The Map function starts execution. It performs its calculation exactly once for each $\langle k1, v1 \rangle$ key/value pair and generates its corresponding output;
- The MapReduce framework designates the Reduce role to available slots and performs the *shuffle* phase. In this step, the framework sorts all Map outputs per key and redirects all key/pair values with the same key to the same processor running the Reduce function;

- The Reduce function starts executing. It performs its calculation exactly once for each $\langle k2, list(v2) \rangle$ set of key/pair values;
- The MapReduce framework gathers the output of all Reduce functions and sorts the contents of each one per key. The sorted output of the Reduce functions is the output of the job.

2.4.2 The Word Count Example

The most traditional example of execution of a MapReduce job is the Word Count program. Its purpose is to count the occurrences of each particular word in an input text. This section describes how it works and figure 2.4 illustrates the execution flow.

For the execution of the example, the following text will be considered as the input of the job:

```
This is an example of MapReduce job
This program is called Word Count
Word Count is a MapReduce job example
```

Following the steps previously discussed, consider a 3-mapper, 1-reducer MapReduce cluster setup. In this case, the input is split between the three nodes executing the Map function and each of them gets approximately a balanced one third of the total input.

In Word Count, the purpose of the Map function is to generate key/value pairs counting the occurrences of each word of text. Therefore, for each word W , a key/value pair $\langle W, 1 \rangle$ is generated.

The output of all mappers is then subject to the shuffle phase, in which the MapReduce framework sorts and groups the generated key/value pairs (all intermediate pairs with the same key are grouped). Then, the pairs with the same key (in this case, each different word is a key) are all redirected to the same Reduce node, which in this program will sum up all values for the same word and compute the final count of each word from the input. In this example, as we assumed only one Reduce node would be available, the Reduce step is simplified to a single box in the figure.

2.4.3 Distributed File Systems

MapReduce is built upon the availability of *Distributed File Systems* (DFSs). Such file systems enable distributed storage and bring many advantages to the execution of MapReduce jobs. Different MapReduce framework implementations normally present their own DFSs, like Google and its *Google File System* (GFS) [28] or Apache Hadoop and its *Hadoop Distributed File System* (HDFS) [62].

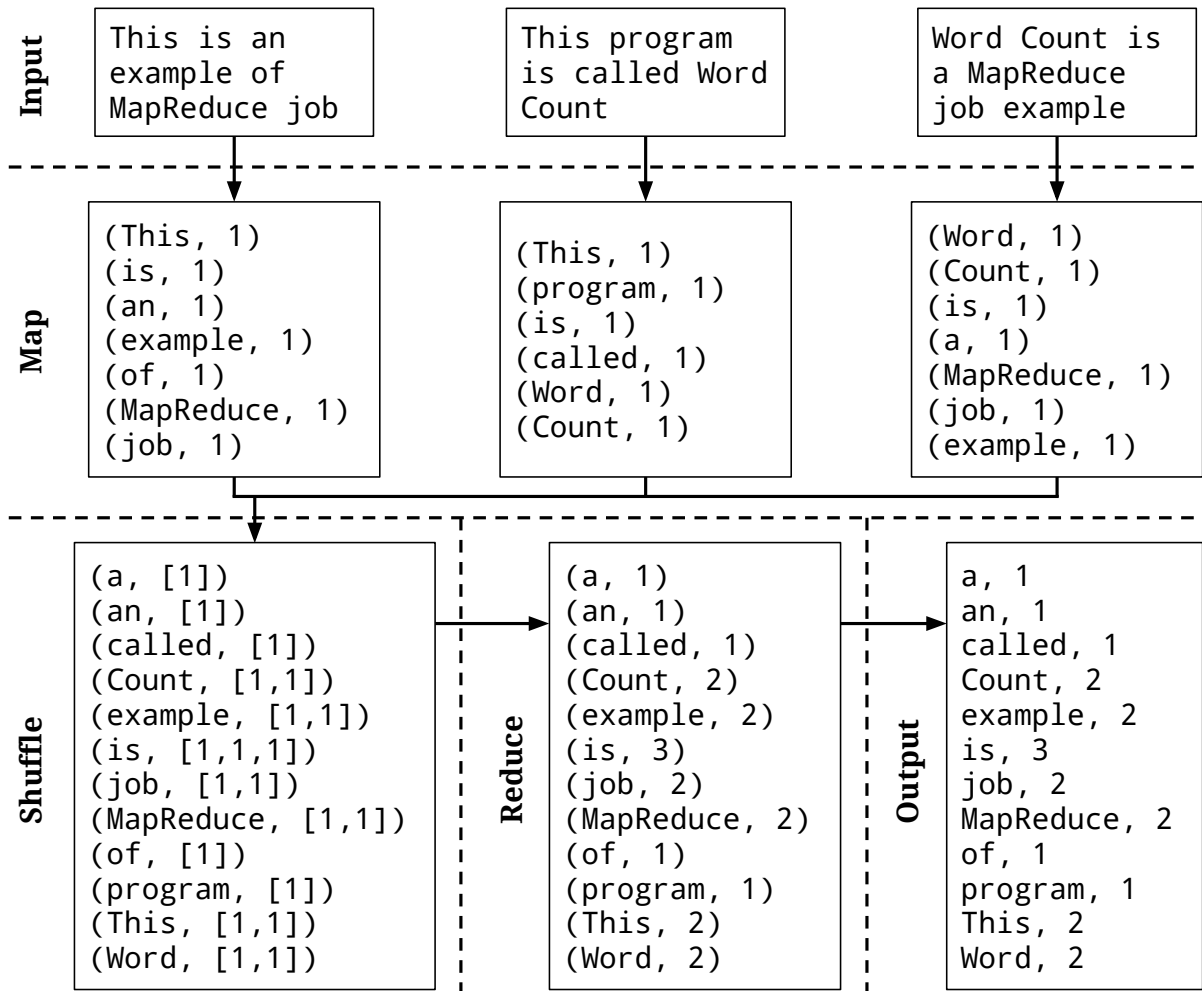


Figure 2.4: The execution of the Word Count example

DFSs are file systems designed to be able to store very large files in commodity storage hardware with the possibility of streaming access through block-designed access patterns. Files are stored in blocks of typically 64MB, distributed among nodes and/or replicated for high availability and low access latency, coupled with strong reliability.

With the utilization of DFSs in the job workflow, MapReduce offers the benefits of locality. Since network may be considered a scarce, low performance resource, MapReduce tries to minimize the communication overhead by allocating as much computation as possible near to the locations where data is already available.

2.4.4 Advanced Features

MapReduce offers the possibility for the developer to determine some specific additional steps in the job workflow by means of the *Partition function*. By using it, the programmer is able to implement a custom logic on how the Map outputs should be split and redirected to the Reduce nodes. The Partition function determines the ID of the desired Reduce node based on the key, and that is specially useful for database sharding purposes. Other additional steps of interest are provided, such as the *Combiner function*, which associates different values of different pairs with the same key in a Map node, in order to optimize communication latency of exceedingly repetitive intermediate keys.

MapReduce also enables fault tolerance. Distributed systems unfortunately inherently suffer from high chances of hardware failure, which translates into the adage “failures will occur”. For instance, say a cluster has ten thousand machines and assume that the *Mean Time Between Failures* (MTBF) indicator of a hypothetical super reliable machine is thirty years [16]. Therefore, on average, one of such machines will run uninterruptedly for thirty years before a fatal failure occurs, but on the other hand, if ten thousand machines execute concurrently, one must be prepared to face failures on a daily basis. Failures may include power supply shortages, network unavailability and defective components. Because of that, MapReduce was thought to dynamically relocate tasks and resources in case of failure. The master node basically pings the worker nodes periodically and queries about the status of the current tasks, and if a worker node fails to reply to that ping timely, the master node considers its task as failed. Therefore, that particular task becomes available for relocation to any other available node. At any time the faulty machine recovers from failure, it becomes idle again for new tasks to be allocated. On the other hand, failures may also occur at the master node. That kind of failure can be recovered by periodically checkpointing the progress of the master node, and in case it fails, a copy of its state can be recovered from the last checkpointed state. Even if MapReduce recovers from such a failure state, though, situations like this are not likely to occur, since there is always a single master node in the cluster and the MTBF for a single machine is usually sufficiently high to be ignored.

Frameworks still may offer features that may be useful for progress tracking and performance analysis, such as real-time HTTP services that provide information on the progress of jobs and status of the cluster and counters that enable deep analysis of performance bottlenecks.

This work integrates MapReduce to a proposed programming model for distributed-memory SIMD processing, in order to harness its advantages like fault tolerance and workload balancing, while avoiding any need to implement those features.

2.5 Related Work

Plenty of research effort has been concentrated into the field of cloud computing, demonstrating that it has become a very interesting paradigm for parallel computing. There are various lines of research regarding parallel, distributed processing, MapReduce and cloud computing applications and optimizations. The following is a brief description of works which somehow relate to OpenMR, our proposed model.

2.5.1 MapReduce Applications

MapReduce is an attractive framework for distributed processing, therefore its application areas are widely investigated. Recent works have demonstrated, for instance, that MapReduce's model is well suited for machine learning algorithms. Ganjisaffar et al. [27] showed that the tuning of machine learning parameters can be done in parallel with MapReduce jobs, while Tartara and Reghizzi [57] showed that parallel training runs for speedup of machine learning in compilers naturally fit MapReduce's model.

MapReduce is applied not only inside Computer Science but also in other areas of knowledge such as Chemistry and Biology. Examples of such usage of MapReduce are the works of Leo et al. [37] and Menon et al. [40], which both deal with genome analyses performed in a distributed manner with MapReduce. The former involves the investigation of genetic causes for phenotypical expressions through the highly parallel processing of genetic variants in large batches, while the latter involves the alignment of sequences of various types of biological data, such as proteins, DNA or variations between individuals. There are interesting MapReduce applications even in fields such as health care, which is the case of the work of Deligiannis et al. [18]. That work uses the MapReduce framework to analyze extensive medical records in search for parameters which may correlate with heart disease conditions for faster and better diagnosis.

2.5.2 MapReduce Framework Optimizations

As previously mentioned, MapReduce has become a very interesting framework for parallel, high-performance computing. Therefore, it has also naturally attracted researchers who seek to analyze its characteristics, such as performance and reliability, and propose optimizations and enhancements to the model.

An example of such a work is the work of Cardoso et al. [6], that analyzes the behavior of globally, highly distributed MapReduce clusters with respect to different workloads and output aggregations and provides recommendations for cluster setup configurations in order to better harness widely distributed resources. The works of Bressoud/Kozuch, Jin et al. and Zheng [5, 32, 67] analyze, investigate and model MapReduce's fault tolerance

system, which is based on *checkpointing*. Those works propose, for instance, techniques such as task and data redundancy in situations when failure rate is high. The impacts of data locality and scheduling fairness are investigated by Guo et al. [30], which propose algorithms for yielding optimal data locality and to balance the tradeoffs of locality and scheduling fairness. Chen et al. [9] use a tiling strategy to split the chunks of data in order to achieve better performance rates of MapReduce applications on multicore platforms. Cho et al. [10] propose task preemption for MapReduce to support jobs with priorities and hard real-time constraints.

2.5.3 Data Movement/Distribution in the Cloud

Since MapReduce is primarily aimed at Big Data, data movement and distribution in the cloud are critical issues which may hinder the effectiveness of cloud applications. Our results demonstrate that OpenMR's performance depends heavily on the remote data availability and/or on data movement/distribution. Therefore, works in this field try to address these issues. Abu-Libdeh et al. [1], for instance, propose the striping and replication of data, common RAID techniques, in the context of cloud storage. These techniques intend to address service outages and improve overall performance through the higher availability of data. Wang et al. [61] investigated the effects of replicated data distribution in distributed database join queries and propose algorithms to improve the performance in such cases. Yu et al. [66], in turn, address the security and data confidentiality of cloud storage, overcoming the computation overhead inherent to an ordinary key-based cryptography and providing scalable secure storage for fine-grained data.

2.5.4 Parallel Programming Models

In recent years, researchers have proposed the extension of parallel programming models to turn it easier and simpler for the developer to write parallel code and take advantage of available parallel hardware, in various granularities. Such works are the ones which relate to OpenMR the most, and are listed below.

OpenACC

One example is OpenACC [47], an API with specifications very similar to those of OpenMP that aims at parallelizing code on available accelerators, or co-processors, in particular GPUs. Just like OpenMP, it was created and is maintained by a committee composed by members from some of the most important companies and universities/research centers on HPC in the world.

OpenACC was first published in 2012 and is composed by C/C++ and Fortran directives, very similar to those of OpenMP but which enable code parallelization not only in the CPU but also in GPUs, by exploiting their large potential of data-level parallelism in a massively parallel type of hardware.

Until the establishment of such a standard, there were only two significant programming interfaces that enabled the usage of GPUs for general-purpose programming (GPGPU), namely OpenCL and CUDA. CUDA [46] is a proprietary programming model for NVIDIA’s GPUs. By means of an extension of the C language (other languages are also supported through wrappers), CUDA enables the development of code specific to the GPU architecture. Such code is written as C functions called kernels that are asynchronously triggered by the CPU for execution on the GPU. OpenCL [33], in turn, is an open standard maintained by a consortium called The Khronos Group. It is a more general framework for general-purpose computation in various types of co-processors/accelerators, such as GPUs, Digital Signal Processors (DSPs) and others. OpenCL is also based on a specific programming model which encompasses plenty of implementation details and conventions, such as its execution and memory models, which must be very well understood by the developer.

Based on that, the purpose of OpenACC is to facilitate the task of implementing code to be parallelized in those kinds of accelerators. Through OpenMP-like annotations to the code, OpenACC generates code that manages the acceleration of programs in the available devices. OpenACC compilers usually translate regions of code annotated with OpenACC directives to CUDA or OpenCL code [55], with no need for the developer to write specific code to any of the two platforms.

OpenACC may be thought as an extension to OpenMP, expanding the processing to co-processors in something that may be called a finer-grained level of SIMD parallelism. The three standards (CUDA, OpenCL and OpenACC), though, have been developed to suit specific types of parallel applications which in general exhibit great potential for data-level parallelism of simple arithmetic operations.

SnuCL

SnuCL [34] was developed at the Seoul National University and is a framework for heterogeneous CPU/GPU clusters that extends the semantics of OpenCL to distributed-memory, heterogeneous environments. It is similar to OpenMR to some extent, since it represents an abstraction with which a developer writes code to execute remotely. It implements a runtime that allows for the development of accelerator-enabled code similar OpenCL code. It executes in distributed, heterogeneous clusters as if the accelerators were present in the local host. The researchers responsible for SnuCL argue that this extension to the OpenCL model naturally fits into the accelerator-programming semantic.

The SnuCL-enabled compiler is responsible to perform code transformations which characterize the distributed execution of SnuCL programs. Still, SnuCL code is very similar to OpenCL code (representing little advancements toward ease of programming) and SnuCL depends on the availability of accelerator-enabled computer clusters.

SnuCL does not require the explicit utilization of external communication APIs such as the MPI library, but the complexity of developing applications with an interface so similar to OpenCL's remains. In fact, the SnuCL proposal removes the complexity of integrating OpenCL applications which were thought to execute in a local context to a remote, distributed environment, but the developer still needs to master specific programming conventions and execution models. That is, in fact, one of the main motivations behind OpenACC, which actually maps directive-annotated code to OpenCL-equivalent code, in a manner much closer to OpenMR than SnuCL's. Additionally, the runtime proposed alongside SnuCL does not provide highly desirable features such as fault tolerance and workload balancing (essential in such an environment which is highly prone to hardware failures), features which are pre-existent in MapReduce and automatically harnessed by OpenMR.

PGAS Languages

Partitioned Global Address Space (PGAS) is a parallel programming model that abstracts a global memory address space among threads/processes while providing data locality features, providing means of defining shared and local variables and supporting the execution of applications in both shared-memory and distributed-memory architectures. This model is implemented by several programming languages, such as *Unified Parallel C* (UPC) [21, 41], *Coarray Fortran* [45], *X10* [8] and *Titanium* [65].

Such languages are mostly derived from versions of popular languages such as C, Java and Fortran. Their premise is that the languages they are based on are not well suited for non-uniform data access across nodes of a distributed system [8]. Although the original languages may support shared memory parallel execution, attempts have been made to provide support for a distributed shared memory architecture. Such an approach is clearly very difficult to scale, since the latency involved in the access of remote data is prohibitive, thus such attempts have been unsuccessful [52].

PGAS languages usually provide constructs such as data locality keywords and runtime functions which enable the distributed parallelization of sections of the code. That is the case, for instance, of UPC, which is based on the C language. It provides keywords for the declaration of shared and local variables, alongside parallelization functions and constructs such as the `THREADS` and `MYTHREAD` native variables. Because of that, although being adequate for various levels of code parallelization, such languages have not prospered, since basically all of them represent different programming paradigms

and require specific programming skills to some extent. Also, it is usually easy for the programmer to write inefficient code, by accessing data in another machine which was not supposed to be accessed and causing large overheads. It is interesting to notice that even though the concepts behind PGAS have been proposed back in the early/mid-90's, the same period corresponding to the beginning of MPI, the message-passing paradigm was the one which was widely adopted as the de-facto standard for distributed processing.

Still, PGAS languages do support parallelization of various levels of distributed parallelism. Specifically, the works Paudel/Amaral [49] and Mehta [39] use the suite of Cowichan problems [63] (which are a set of problems created to analyze the usability of parallel programming solutions) to determine the types of parallelism that UPC and X10 support, and find that most Cowichan problems can be subject to parallelization with such languages.

However, even though OpenMR's model currently supports a narrower range of parallel problems (DOALL loops), the amount of problems which can be parallelized using OpenMR is still very large (specially among scientific applications). Besides that, OpenMR thrives because of its ease of programming and similarity to the well-known OpenMP API. Additionally, no PGAS language has native, automatic support to features such as fault-tolerance, which are present in OpenMR/MapReduce (although very recent works have started to address issues related to that [14, 64]).

Elastic OpenMP

Elastic OpenMP [26] is a mechanism proposed to provide *elasticity* to OpenMP applications. Elasticity refers to the system's ability to adapt to workload changes, during runtime, by provisioning or deprovisioning resources. Such mechanism implements an elasticity middleware which communicates with the cloud infrastructure and allows for the utilization of resources for the execution of OpenMP applications in the cloud. The elasticity provided is vertical, which means that for a pre-determined number of running machines, it is possible to increase or decrease the amount of processing cores during runtime (thus saving energy, for instance).

OpenMR, on the other hand, is built upon MapReduce, which means that elasticity solutions for MapReduce are also available for our model. For instance, the experiments conducted during this work, described in chapter 4, used the Amazon EMR services, which provide horizontal elasticity. Horizontal elasticity means that not only the resources within the available machines are manageable, but also new machines can be instantiated (or deactivated) during the execution of an application, providing extra computing power on-the-fly. This is particularly interesting because the use of *Infrastructure as a Service* (IaaS) clouds is usually charged based on the amount of machines and time per machine used, so if the developer/user needs more computing power at a particular moment, more

machines can be instantiated for a short period of time only.

Elastic OpenMP, like the previously discussed programming models, does not support fault tolerance, while OpenMR/MapReduce does. Also, its programmability is not as simple as the original OpenMP, since the elasticity feature is not provided automatically and the developer needs to implement the elasticity logic (the programming model is extended).

Table 2.1 summarizes the characteristics of each of the discussed works and allows their comparison to OpenMR.

	Based on	Target	Elasticity	Fault tolerance	Program-mability
OpenACC	OpenCL, CUDA	Local Accelerators	—	—	+++
SnuCL	OpenCL	Cloud accelerators	No	No	-/+
PGAS	—	Cloud CPUs	No	No	-/+
Elastic OpenMP	OpenMP	Cloud CPUs	Yes (vertical)	No	+
OpenMR	OpenMP, MapReduce	Cloud CPUs	Yes (horizontal)	Yes	+++

Table 2.1: Comparison of the discussed parallel programming models

Chapter 3

The OpenMR Execution Model

This chapter proposes an execution model that, based on the syntax of OpenMP and using the MapReduce framework provided by Hadoop, allows the programmer to harness the compute capabilities of a full-sized cluster, in a manner that involves the simple addition of annotations to the code. We call it *OpenMR* (short for Open MapReduce).

3.1 The Concept

As previously discussed, OpenMP helps the programmer in the task of converting sequential code to parallel code that executes across multiple cores in an SMP context by transparently implementing data sharing among cores and thread creation, destruction and synchronization. In this case, the compiler's job is to reduce the complexity of handling these issues, which include the use of libraries such as Pthreads and their synchronization constructs and the need to take care of race conditions and other forms of problems that parallelism can bring.

OpenMR is an extension of OpenMP which reproduces that train of thought, but within another context. Instead of only parallelizing pieces of code across multiple cores in a single machine, it parallelizes loop iterations across multiple machines while transparently providing desirable features like fault-tolerance, efficient data distribution and workload balancing through heterogeneous commodity machines.

One of the most important features of OpenMP is undoubtedly the parallelization of `for/do` loops. It does so by splitting the trip count of the loop across the threads, so that each core executes the same piece of code over a fraction of the possible values for the basic induction variable of the loop. In the example shown in figure 3.1, a `for` loop with an original trip count of ten thousand iterations is parallelized among twenty threads using OpenMP, assigning to each of them a particular subset of the trip count, i.e. five hundred iterations.

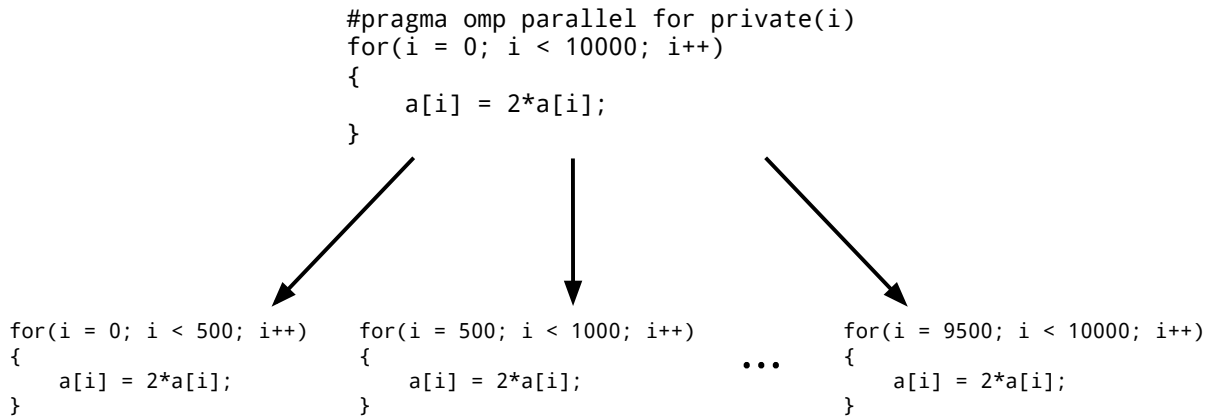


Figure 3.1: OpenMP's trip count parallelization

In OpenMR, the parallelization of `for` loops also plays an important role. Figure 3.2 shows how this happens in the proposed execution model. By using constructs which are very similar to those from OpenMP, the programmer tells the compiler to prepare a MapReduce job which, when executed, will yield the same results expected from the original sequential piece of code. The goal is to have the sequential code to execute normally until an OpenMR region is reached, then trigger the job in a remote cluster, get the results back when they are ready and resume execution.

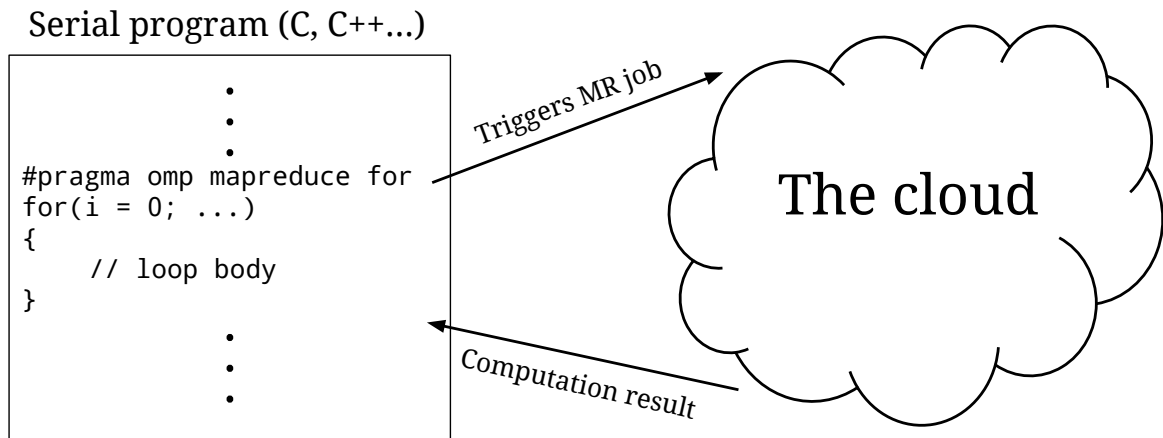


Figure 3.2: OpenMR's execution model

The flow of a MapReduce job generated by OpenMR is based on the mapping between iterations and processing nodes. Figure 3.3 is an example of how it works in a small instance. The job takes as input a single file, which at first is created by OpenMR itself during runtime before the execution of the parallel region. The file consists of one line of text per iteration, each of which containing the number of the iteration and the current

values of all the variables to be used during that specific iteration. Each node of the MapReduce cluster running the Map function (henceforth denominated “mapper nodes”) will receive one or more of those lines. Then, for each line, a mapper node will parse the variable values, store them and execute one of the iterations of the original loop. Combined, the mapper nodes will have executed the whole loop by the end of the job. Each mapper node will output a key-value pair with the iteration number and the final values of the pertinent variables, as computed during the job, and those values will be retrieved by the local machine which triggered the job.

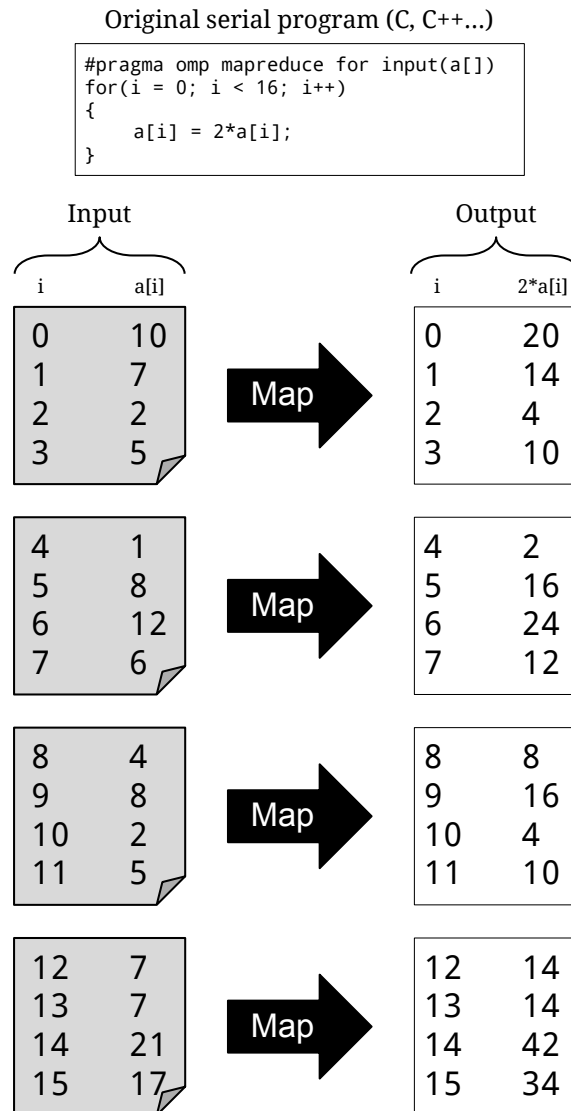


Figure 3.3: Mapping between iterations and executing nodes, for a four-node setup

If a **reduction** clause has been declared in the **#pragma** directive, a reduction phase

will be included in the MapReduce job. The node(s) running the Reduce function (henceforth the “reducer(s)”) will take care of reducing the Map results to the variable(s) in the clause.

Due to its distributed nature (that means no communication among threads via shared memory), OpenMR requires the loop to be DOALL. As we can virtually assume that each machine deals with its own data set, data produced by a certain machine will not be available to any of the others during processing.

3.2 The Syntax

We propose a couple of directives based on OpenMP, designed to extend the functionalities of the existing API.

3.2.1 The mapreduce Construct

The `mapreduce` construct is the distributed equivalent to OpenMP’s `parallel` construct. Figure 3.4 is an example of usage of that keyword. It defines a region of code which should be parallelized in the cloud by the compiler. During compile time, that region of code will be replaced by constructs that prepare and trigger the MapReduce job and retrieve its output, as will be discussed in section 3.3.

```
#pragma omp mapreduce for
for(i = 0; i < N; i++)
{
    // loop body
}
```

Figure 3.4: The `mapreduce` construct

The `mapreduce` construct was initially thought to work only with `for/do` loops, therefore all of our examples will feature this type of work-sharing construct. We will discuss more about the other OpenMP’s work-sharing situations (`sections`, `task`) later. Additionally, we propose a special case of use of the `mapreduce` construct, alongside the `data` keyword (subsection 3.2.3).

3.2.2 The input/output Clauses

When defining an OpenMR region, the programmer will be able to use the computational potential of the cloud on his/her behalf without having to write specific code in another

language or paradigm. However, the person will still have to keep in mind that not all data manipulations during the execution of that region will remain as a side effect. In fact, the programmer will need to use the `input/output` clauses to specify which variables will be taken as input and as output of the processed region. Figure 3.5 shows an example, in which the contents of the array `b[]` are passed to the mapper nodes as input data and the contents of the array `a[]` are retrieved by the local machine when the remote processing is done.

```
#pragma omp mapreduce for input(b[]) output(a[])
for(i = 0; i < N; i++)
{
    a[i] = b[i] + 1;
}
```

Figure 3.5: The `input/output` clauses

With the use of the `input` clause, the programmer informs to the compiler that those specific variables are live-ins to the region. Thus, they will need to be provided to every executing node, namely through each line of the input file.

The `output` clause, in turn, specifies which variables are subject to be retrieved by the end of the processing. Without it, no variables are locally updated after the execution of the `mapreduce` region and we can assume the job worked strictly as data generator for a possible subsequent job.

The following subsection discusses the data manipulation constructs of OpenMR.

3.2.3 The data construct

The programmer might not want to send the input data to the job at runtime, but rather use data already available remotely. For this reason, OpenMR proposes a construct called `data`. Figure 3.6 shows examples of the usage of `data`. With this directive, we want to let the programmer tell the compiler that the data which will be used by the MapReduce job are previously available in the remote cluster, avoiding the performance degradation of sending big data chunks through a potentially highly inefficient communication network.

In figure 3.6 below, three array declarations are used to assign code to the cloud. In the first declaration, every iteration potentially uses the whole array `a[]`. The second one is intended to be used as an optimization to the problem of data distribution, something like MPI's `scatter` function. In this case, each mapper node will expect to receive $1/\text{nodes}$ elements from the matrix `b[] []`, effectively scattering the rows of the matrix. The latter


```
#pragma omp mapreduce data
int a[10000];

#pragma omp mapreduce data
int b[10000:nodes][10000];

#pragma omp mapreduce data
int c[10000:nodes][10000:nodes];
```

Figure 3.6: Examples of the `data` construct

example is like the previous one, but it scatters the contents of the matrix `c[][]` on a block basis.

These novel variable declarations may also be seen as hints to the compiler, so that the declared variables will not be a part of the input file generated by the local machine which is sent to the remote cluster. They can also be seen as an abstraction to opening a file in a remote/distributed file system, thus changing our perspective on remote processing from a totally different paradigm to just a matter of granularity.

In many cases, the use of the `data` construct is the key to achieve high speedups in OpenMR, since the data communication overhead is dismissed.

3.3 Preparing the MapReduce Job

The compiler is responsible for generating the Map and Reduce functions, which will allow for the execution of an OpenMR job. As previously stated, the machines executing the Map function will each execute a subset of the iterations of the original loop and the machines executing the Reduce function will take care of an optional reduction phase.

The MapReduce framework used in this dissertation is Apache Hadoop [25, 62]. Hadoop is written in Java, and the ordinary method for creating a Hadoop job is by writing both Map and Reduce functions as Java classes. As we are dealing with procedural languages like C, it is logical that using the same language to write the functions would be much more practical, not to mention how easier it would be to implement OpenMR in a real compiler in the future. Thus, the adopted solution was Hadoop Streaming. Streaming is a Java class included in the standard Hadoop distribution which allows for using scripts and/or binaries written in virtually any programming language as the Map and Reduce functions. It is simply a matter of writing a program that reads input data from the `stdin` stream and writes pertinent output (key-value pairs) to the `stdout` stream.

Hadoop Streaming is said to add overhead to the flow of a MapReduce job, but it has

been shown that it may compensate the added overhead, depending on the nature of the processing [20]. For problems that require intensive data manipulation, the performance degradation is clear, mostly due to the data transportation through Linux pipes. However, for computational intensive jobs which deal with little input/output, Hadoop Streaming may even yield results comparable to those achieved by MPI implementations of the same problem. That is because the programmer is able to write the functions in a much more optimized language, and input/output are proportionally insignificant. Fortunately, that is the case of problems that suit OpenMR best.

That said, we will explain OpenMR's parallelization strategy by looking at figure 3.7, which shows the several pieces of code that replace an OpenMR region:

1. The creation of the input file to the MapReduce job, which contains multiple lines of text with all the data required for the OpenMR region to be executed remotely, and a call to the library function `omr_push_input()`, which sends the input data via SFTP to the remote cluster (which may or may not be present, given the `data` construct discussed in section 3.2.3). In figure 3.8, this piece of code is indicated as number 1;
2. Library function call (`omr_trigger_and_wait()`) to trigger the job in the cluster via SSH and perform a (synchronous) busy waiting by constantly querying the job status until it has succeeded. In figure 3.8, this piece of code is indicated as number 2;
3. Library function call (`omr_retrieve_output()`) to retrieve the job output via SFTP and associated piece of code to parse the output and store the new variable values, before the program continues normal local execution (also may or may not be present, given the `data` construct discussed in section 3.2.3). In figure 3.8, this piece of code is indicated as number 3.

When compiled, the binary file to be run locally will be equivalent to the code in Figure 3.8.

To create the Map function, the compiler has to extract the loop body exactly as it is (except for the reduction commands, which need to be changed to a simple assignment) and inline it into another loop, which takes one line of input, parses the variables in it and executes one single iteration from the original loop, while there is input. Figure 3.9 is the corresponding Map function to the example from figure 3.7. The mapper function also needs to output its processing, through simple print instructions. In fact, it needs to output every last assignment to the variables listed in the `output` clause. The output is formatted to key-value pairs, in which the separation between key and value is done by a tab character and the separation between pairs is done by a newline character. If a reduction phase is present, every Map output will have the same key, since all of them

```

int main()
{
    int sum, a[N], i;
    // Initialization of a[] and sum

    #pragma omp mapreduce for input(a[]) \
        output(sum) reduction(+:sum)
    for(i = 0; i < N; i++)
    {
        sum += 2*a[i];
    }
}

```

Figure 3.7: Example of OpenMR code before compilation

```

int main()
{
    int sum, a[N], i;
    // Initialization of a[] and sum

    for(i = 0; i < N; i++)
    {
        file_write(omr_input, i);
        file_write(omr_input, a[i]);
        file_write(omr_input, "\n");
    }
    omr_push_input();

    omr_trigger_and_wait();

    omr_retrieve_output();
    sum += file_read(omr_output);
}

```

Figure 3.8: Example of OpenMR code after compilation (pseudocode)

will have to be redirected to the same reducer node. Otherwise, each mapper outputs its own key, which will be its iteration ID.

The Reduce function is quite similar. Figure 3.10 is an example, also relative to the code of figure 3.7. While there is input, a reducer node will read the individual results of mapper nodes from `stdin` and do the specified reduction operation. After reducing all the input records to a variable, it outputs the result and the MapReduce job is finished.

```
int main()
{
    int sum, a[N], i;

    sum = 0;

    while(buffer = getLine())
    {
        i = getToken(buffer);
        a[i] = getToken(buffer);

        // The actual loop body:
        sum = 2*a[i];

        // Print output
        printf("0\t%d\n", sum);
    }
}
```

Figure 3.9: Example of Map function (pseudocode)

Generating the Map and Reduce functions is one of the tasks of OpenMR during compile time. Besides that, it is still a task of OpenMR to send these binaries to the remote cluster before execution, so that all nodes have local access to the functions.

It is important to emphasize that this example works strictly as an explanation of how the OpenMR region is translated into a MapReduce job. One may notice that it may not be worth using OpenMR to parallelize such a simple loop, since all the overhead involving the making of the input file, performing a job and getting back the results is clearly much higher than the actual local processing. However, in the case of much larger, longer-lasting loops and/or loops which are executed over huge amounts of data, OpenMR attains very interesting results, as demonstrated in chapter 4.

It is also important to notice that OpenMR's Map functions maintain total compatibility with nested OpenMP `parallel` regions. If, for instance, an OpenMR piece of code presents a nested OpenMP `parallel` construct, it will be kept as-is and the mapper nodes will each execute their nested version of the parallel code. This possibility may be interesting when a pre-existing piece of OpenMP code is transposed into an OpenMR application, avoiding the need to modify the nested parallelism. However, in most cases when an OpenMR application executes inside a totally busy cluster (preferably using 100% of Map/Reduce slots), such nested parallelism will not likely attain higher performance gains. Still, in some cases covered in chapter 4, the data distribution might result

```
int main()
{
    int sum, a[N], i;

    sum = 0;

    while(buffer = getLine())
    {
        // Waste key and get value
        getToken(buffer);
        tmp = getToken(buffer);

        sum += tmp;
    }

    // Print output
    printf("%d\n", sum);
}
```

Figure 3.10: Example of Reduce function (pseudocode)

in cluster underutilization. In those cases, nested parallelism with OpenMP may be useful to fill idle processing slots.

3.4 Applications

The example from figures 3.7 through 3.10 is simple enough to didactically explain the mechanism of obtainment of all components necessary to a MapReduce job. However, that specific loop is clearly too simple for OpenMR to be worth it. It would be necessary a huge amount of data (previously available in the cloud) and a proportionally compatible amount of processing nodes to have the parallelism to outperform the local processing of such a simple loop.

Since not all applications are structured in a way that could harness OpenMR, the following is a brief discussion on which kinds of code are more likely to achieve better performance rates in the cloud.

First of all, the parallelized loops must be DOALL, since there is no communication between processing nodes.

Second, the OpenMP-based loop should not contain explicit synchronization constructs. These constructs, such as `barrier`, are not yet modeled into OpenMR. Still, in some cases, the translation to loops without these constructs is possible.

With that in mind, potential OpenMR applications may be split in two categories: the ones which demonstrate a profile of high computation over a small set of data, i.e. compute-bound applications, and the ones which compute over a huge set of data, i.e. I/O-bound applications. Although this is not a strict rule, because there may be other types of applications which may take advantage of cloud execution when there are insufficient local resources, those are the most relevant classes of applications for OpenMR under normal conditions.

For compute-bound applications, the longer a mapper node keeps processing during a single iteration and the smaller the input data set for that iteration, the better. This is because we want to avoid the overheads imposed by communication and data manipulation and achieve the highest computation/communication ratio as possible. The execution time of the application, when running locally, should also compensate the time it takes for OpenMR to start and finish running. Even if the application has a high ratio between computation and I/O, it may execute fast enough locally to make the computation in the cloud undesirable. Specific examples (SPEC OMP2012 benchmarks) are discussed in chapter 4, in which the benchmarks used to do performance evaluation are explained. Also, for that type of applications, the loop which is subject to parallelization must have a high trip count, which will account for how much parallelism the execution can achieve, since each iteration will be mapped to a mapper node. Eventually this will become a trade-off, because as the number of iterations grows, so might the data replication and the communication overhead.

I/O-bound applications, on the other hand, exhibit a much more profitable potential for OpenMR. Since a huge set of data is to be processed, it is reasonable to assume that the data will not fit the memory of a local machine, thus it will have to be fetched from the disk in a timely manner. This, in turn, will become a bottleneck. Therefore, despite being called I/O-bound, these are not usual memory-bound applications (I/O, in this case, indicates secondary storage). Even if the loop is OpenMP-parallelized, the performance of the application will be bound to the storage access rate, thus serializing the application. However, the distributed nature of MapReduce storage grants OpenMR applications the possibility to fetch data and move computation distributedly. This, in practice, translates into nearly linear speedups, given data previously available in the cloud. To enable distribution of OpenMR data in the cloud prior to the actual execution, we discuss the creation of specific constructs in chapter 5. Examples of I/O-bound OpenMR applications are discussed in chapter 4. Another huge advantage of parallelizing I/O-bound applications into the cloud with OpenMR is that the annotation-based code parallelization allows for code verification on a small data subset and then, with the addition of simple directives, unleash the processing of a much larger data set in the cloud.

Chapter 4

Methodology/Experimental Results

This chapter describes the methods used to assess the feasibility of OpenMR and shows the experimental results which support the ideas behind it.

4.1 Compiler Infrastructure

OpenMR’s goal is to allow for the easy usage of the cloud’s computational potential through OpenMP-like directives. Thus, a fully-implemented compiler support to OpenMR is required to enable the automatic and transparent implementation of its features.

Since OpenMR extends the functionalities of OpenMP, a solid compiler implementation of OpenMP is needed for the development of OpenMR. *GNU Compiler Collection* (GCC), one of the most important open-source compilers, has its own implementation of the OpenMP specifications, through its *libgomp* library. Moreover, in 2013, support to the OpenMP 3.1 specifications in the *Clang/LLVM* compiler has been announced and is currently under development.

The support for OpenMR in a compiler such as GCC or Clang/LLVM was not implemented in this project; we rather suggest that as a future work. In this work, however, the implementation of OpenMR in a real compiler has been investigated. To the date of this work, GCC was the only open-source compiler supporting OpenMP and therefore it was the compiler of choice for the analysis.

The compilation process in GCC is illustrated in figure 4.1. The front end parses input code from various languages, generating the corresponding *Abstract Syntax Trees* (ASTs). Then, the AST is converted to *GENERIC*, a language-independent form of syntax trees, which is then subject to the process of *gimplification*, the conversion from *GENERIC* to *GIMPLE*. *GIMPLE* is the in-memory *Intermediate Representation* (IR) used by GCC’s middle end optimization techniques on the code through dozens of tree optimizer passes. After the optimization passes, GCC obtains code in *Register Transfer*

Language (RTL), a LISP-inspired representation very close to assembly code but still architecture-independent. This RTL code is the input for the back end of the compiler, which will execute the actual code generation for the specified architecture.

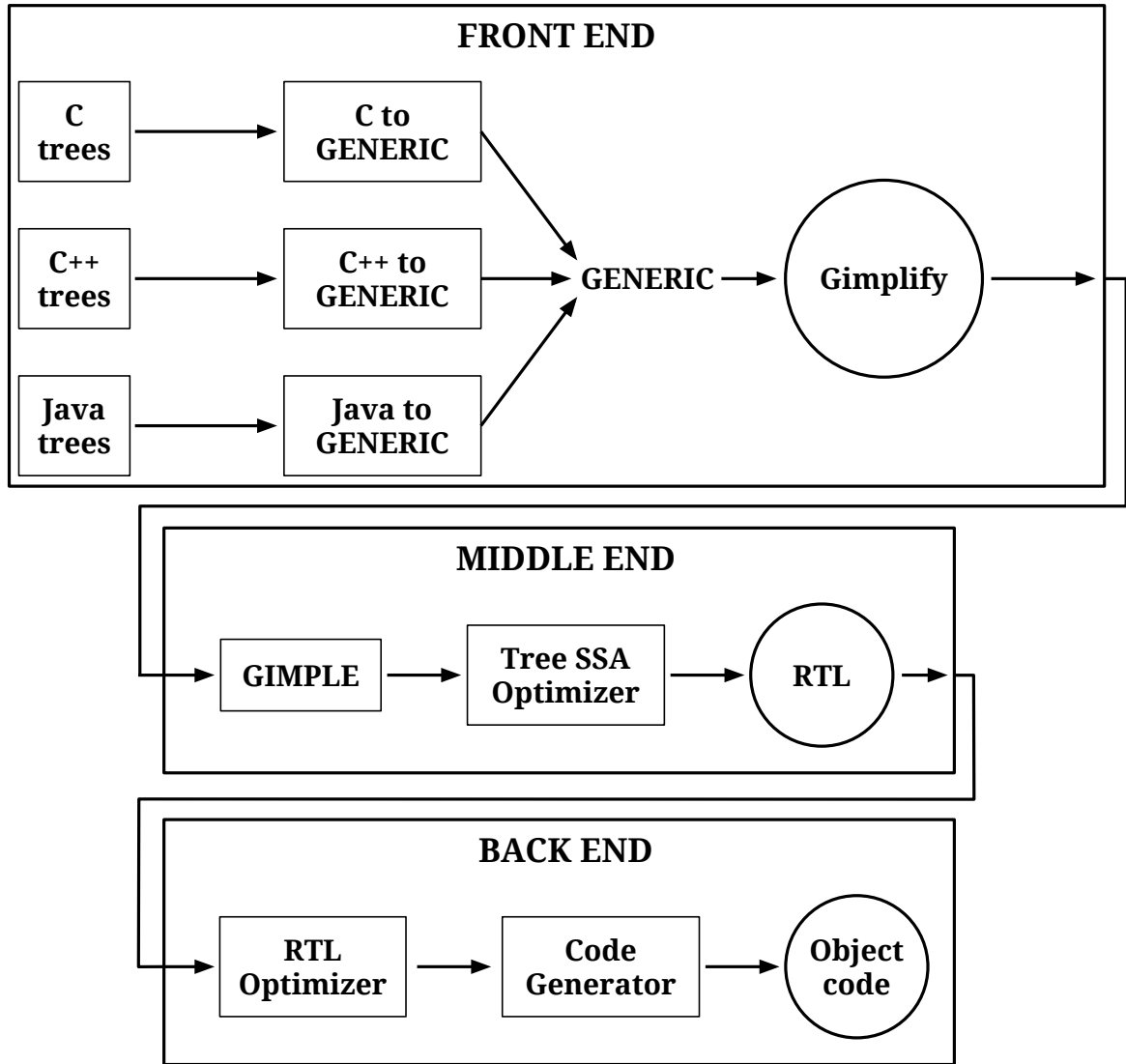


Figure 4.1: The compilation flow in GCC (adapted from [44])

In GCC, OpenMP handling is done during various stages of the compilation process. The C/C++ and Fortran front ends take care of parsing the `#pragma omp` directives, annotating the GIMPLE IR with information about the parallelization constructs. The middle end, in turn, is responsible for actually handling them during the optimization passes. These passes reorganize the content of parallel regions in subfunctions, handle the

data according to the data sharing clauses and insert calls to libgomp, the library that manages the creation and manipulation of threads.

The rearrangement of the code of a `parallel` region is as follows:

```
#pragma omp parallel
{
    body;
}
```

becomes, after the appropriate transformations,

```
void subfunction (void *data)
{
    use data;
    body;
}
```

```
setup data;
GOMP_parallel_start (subfunction, &data, num_threads);
subfunction (&data);
GOMP_parallel_end ();

void GOMP_parallel_start (void (*fn)(void *), void *data,
                        unsigned num_threads)
```

In other words, OpenMP extracts the body of a `parallel` region and creates a subfunction which is executed as a callback to the function of libgomp, which in turn is responsible for the creation of the team of threads that will perform the computation. One interesting fact about the way GCC does this transformation is that the function outlining is performed without generating debug information for the data structures used in the process, thus turning it difficult to debug the execution of parallel code.

This outlining of OpenMP code is particularly interesting for this project, given that it is very close to the way OpenMR proposes to manipulate code, as described in figures 3.8 through 3.10. In OpenMR's case, outlining is the main technique used in the creation of the Map and Reduce functions, since the actual loop body is inlined in the code for the Map function and the reduction operations are also extracted from the original loop into the Reduce code, in case a `reduction` clause has been defined. Also, the original loop is removed from the sequential code and library function calls are inserted with the purpose of triggering the remote job and retrieving the result, just like what OpenMP does with respect to the creation of threads and what we may abstract as the firing of tasks in a parallel context.

4.2 Benchmarks

This investigation on the compilation process of OpenMP confirmed the structure of the code that should be generated by the compiler in a possible implementation of OpenMR. Then, with that in mind, experiments were made to assess the performance of OpenMR.

The experiments performed during this work were based on manually modifying parallel computing benchmarks, in order to transpose the OpenMP code into our new execution model, writing code that is equivalent to the one that the compiler would generate in a transparent fashion.

The compute-bound benchmarks used were those from the SPEC OMP2012 suite. This was the benchmark suite of choice because it presents various types of scientific applications that have already been parallelized with OpenMP, therefore those applications might exhibit potential to be parallelized with the directives of OpenMR. Table 4.1 presents a description of some characteristics of all applications composing the benchmark suite, including the languages in which the programs are written, the memory size used by the applications' data and the area of knowledge the applications are associated with.

Code	Memory (MB)	Language	Area
350.md	5	Fortran	Molecular Dynamics
351.bwaves	22,800	F77	Computational Fluid Dynamics
352.nab	618	C	Molecular Modeling
357.bt331	11,188	Fortran	Computational Fluid Dynamics
358.botsalgn	156	C	Sequence Alignment
359.botsspar	7,179	C	LU Factorization
360.ilbdc	16,482	Fortran	Lattice Boltzmann
362.fma3d	5,205	F90	Finite Element Method
363.swim	6,490	Fortran	Finite Difference
367.imagick	1,733	C	Image Processing
370.mgrid331	13,972	Fortran	Multi-Grid Solver
371.applu331	14,884	Fortran	PDE/SSOR
372.smithwa	177	C	Optimal Pattern Matching
376.kdtree	119	C++	Sorting and Searching

Table 4.1: Description of the benchmarks in the SPEC OMP2012 suite (adapted from [42])

In the experiments, the focus was on the transcription of the C/C++ applications to the OpenMR code structure. This approach was used both because the author had little familiarity with the Fortran language and because even though the Fortran applications were briefly analysed, most of them demonstrated little potential for performance gains

with the constructs designed so far for OpenMR. The reason for that, though, is obviously not the language in which the applications were written. That is actually a particularity of the applications, since Fortran is still today one of the most important languages for scientific HPC.

From the set of analysed benchmarks [42], two of them were selected and implemented with OpenMR, namely:

- 358.botsalgn: protein sequence alignment application. A set of protein sequences is read from input file and each sequence is compared against each other using the Myers and Miller algorithm [43]. The SPEC implementation of this program presents two nested `for` loops which basically perform the whole computation. The outer one is parallelized with a `parallel for` work-sharing construct, while the inner one runs sequentially in each of the threads of the team, triggering individual `tasks` that execute one comparison each. The utilization of `tasks` is aimed at the solution of possible imbalance inside an SMP context, but it makes less sense in a coarser-grained context like the cloud. That is because the purpose of `tasks` is that they can be relocated among threads during execution, something that is difficult to do in such an environment. With certain manipulation, the application can be modified to remove the `tasks` and turn the structure of the code adequate to OpenMR. By doing so, the workload balancing strategy may be modified, but 1) the semantics of the code remains unaltered and 2) workload balancing is performed, even if in a different granularity, by the MapReduce framework. In a first investigation, this was one of the applications which seemed to fit better the OpenMR model, since it demonstrated a profile of high computation over a small set of data.
- 372.smithwa: pattern matching, somehow similar to the problem of 358.botsalgn. This application is a parallel implementation of a sequence alignment routine for Matlab. The sequences processed in this application, contrary to the ones in 358.botsalgn, are generated randomly. There are two of them which are compared, with six embedded, pre-determined pieces of sub-sequences for correctness verification. The code is structured in three function denominated Kernels 1, 2A and 2B. In Kernel 1, each OpenMP thread processes a subset of the two sequences with the Smith-Waterman algorithm [54] and builds its individual list of best alignments along with their ending points. In Kernel 2A, threads sweep each of the previous outputs from end to start listing best alignments, starting and ending points and codon sequences. In Kernel 2B, the outputs are merged and displayed. The largest portion of execution time, by far, is associated with Kernel 1, in which the heavy computation is actually performed. That function was parallelized with OpenMR. To do that, though, some manipulation had to be performed with the

code. Originally, the parallelization was achieved by launching a `parallel` region with a dynamic number of threads (typically defined by the runtime as the number of cores in the host), each of which would be responsible for processing a subset of the input data. This dynamic division of the data space had to be modified to a static one, since OpenMR maps iterations to processing nodes and an explicit iteration count must be defined. Thus, with this modification, OpenMR is able to replicate and/or distribute data accordingly through nodes, in a potentially dynamic number of processing cores. This, in turn, has led to what may represent a problem: there are cases in which there is a trade-off between the amount of parallelization and the replication of data/the performance penalty for data communication. That happens because of exceedingly replicated data, which may be necessary among many processing nodes.

There were still four other SPEC benchmarks which did not exhibit potential for parallelization with the current constructs of OpenMR, namely:

- 352.nab: molecular dynamics application. Performs heavy, floating-point calculations. Not a DOALL loop. In an SMP context, inside each individual step, work may be individually parallelized with OpenMP, but the overhead involved in launching a MapReduce job for each operation (data communication included) would be prohibitive.
- 359.botsspar: LU matrix factorization over sparse matrices. There is a lot of imbalance in this problem, due to the sparseness of the matrix, which has been solved in the OpenMP implementation by the use of `tasks`. As discussed about the previous application, the manipulation of the constructs to remove the `tasks` from the code would be possible, except for the fact that two `taskwait` directives are present, imposing explicit synchronization between loop iterations. This, coupled with the serial execution of the outer loop, demonstrates that the loop is not DOALL, fact that was proved by experimentation.
- 367.imagick: Image Magick, an open-source image processing suite. Processes a large variety of image formats, applying dozens of different filters and effects to images, but the operations depend on a strict serialization. Additionally, this application presents a lot of complex data structures which would require an extended strategy on how to serialize complex data.
- 376.kdtree: k-dimensional tree generation and searching. A k-d tree is a space partitioning data structure, which may be seen as an expansion of binary trees. The search is performed through recursive functions, that are currently not well suited

for the constructs of OpenMR, since each call for them would trigger a MapReduce job. The dependencies between iterations are another limiting factor for this application, since a recursive search procedure depends on the data space division of the previous function call, thus precluding the parallelization of programs in a distributed-memory context, with no communication among processing nodes.

Additionally, a set of toy benchmarks has been implemented to assess the performance gains of relatively simple, increasingly complex loop bodies. These benchmarks exhibit regular mathematical operations and are I/O-bound applications, which take full advantage of MapReduce’s distributed processing. The applications were:

- Vector add: application which sums two arrays of integer numbers. The value of each position of array A is summed to the value of the corresponding position in array B. The local execution of the benchmark reads the input data from the disk and processes the two arrays through a simple, DOALL **for** loop. The distributed execution assumes the use of OpenMR’s **data** construct and processes, in each mapper node, fixed-size arrays which are previously stored remotely and which collectively sum up a much larger amount of data.
- Inner product: given two arrays of integer numbers, calculates the inner product. This is done through the pairwise multiplication of elements of both arrays (similar to the previous application) and a final reduction step, which sums all partial results to a single accumulator. The local and distributed execution perform just like in the previous application, with an extra step of reduction which is performed in a single reducer node.
- Matrix-vector multiplication: multiplies a matrix by a vector, or in other words, a bi-dimensional matrix by a single-column matrix. This application exhibits another particular case: the local version executes the operation over a single set of matrix and vector, while the previously-adjusted input data for the distributed job holds a set of data for each mapper node to process. That means that the OpenMR code performs the exact same operations over a single set of data inside each thread, and the whole job processes multiple, distributed sets.
- Matrix-matrix multiplication: very similar to the previous application, now processing sets of full-sized, bi-dimensional matrices.

4.3 Experimental Setup

Three contexts were represented in the performance measurements. The first one was the local, parallel execution with OpenMP. This context has been used in order to compare

the performance gains of the distributed processing to the local processing in an SMP. It will be henceforth referred to as our *baseline* or execution in a *local machine*. The next one was the benchmark execution in a computer cluster locally available (i.e. in the same LAN), which will be referred to as *local cluster*. The third one was the actual execution in the cloud, in a remotely available cluster. It will be referred to as *Amazon EMR*, as *remote cluster* or simply as *cloud execution*.

The baseline benchmarks were executed in a machine with three Intel Xeon X7560 CPUs, each one with eight cores, and 64 GB of RAM. In this 24-core machine, the OpenMP benchmarks were executed with growing amounts of threads, from 1 to 48.

For all OpenMR benchmarks, the required SSH/SFTP communication (remote execution of shell commands and file transfers) was implemented with the libssh API, therefore this library is a prerequisite to OpenMR. Likewise, all benchmarks were executed using Apache Hadoop as the MapReduce framework.

For the experiments with a local cluster, the benchmarks were executed from a machine with an Intel i5/Sandy Bridge processor with two cores and 4 GB of RAM and the OpenMR's distributed processing was performed in a heterogeneous cluster composed of five machines with CPUs ranging from Intel Core 2 Quads to Xeon X7560s with eight cores each and memory ranging from 4 to 64 GB. The five machines, together, had 56 processing cores. All machines executed Linux/Ubuntu 10.04.3 64-bit or 12.04 64-bit. Also, all machines were connected to the same subnet with regular Gigabit Ethernet. The Hadoop framework executed in this cluster was version 1.1.1.

For the cloud experiments, the benchmarks were triggered from the same machine used for the baseline and the distributed processing was performed using the Amazon Web Services (AWS). These are services provided by Amazon which enable the usage of their cloud computation and/or storage. The user creates an account and then is able to instantiate remote clusters in various locations on the planet with custom amounts of machines, paying for the time the machines were used. There are various different types of AWS services, among which the Simple Storage Service (S3), Elastic Compute Cloud (EC2) and Elastic MapReduce (EMR) were used in this work.

S3 is Amazon's cloud storage service. Users may upload and download data to the cloud, being charged for specific network transaction scenarios. The download and upload of files can be performed both through AWS's web console and through certain provided command-line tools, denominated Command-Line Interface (CLI).

EC2 is the service with which users can allocate machines in Amazon's clusters. The user specifies amounts of machines with different characteristics/compute capabilities and EC2 allocates idle machines and instantiates virtual machines in them, giving the user control to execute applications in the cloud. The access to EC2 instances is performed both through the web console and through SSH connections. There are various tiers of

EC2 instances, each one with different costs per hour. Basically, Amazon offers machines of the current generation, at relatively higher costs, and machines of the previous generation, at more affordable prices. Besides that, in the same generation of machines, there are types which are aimed at more intensive computation jobs, with better CPUs and components like SSD storage, others which aim at storage, having larger hard drive capacities, others which have GPUs, and so on. In this work, all EC2 instances were of the type `c1.medium`, except for the Hadoop master node, which was always a `m1.small` (this is because the task tracker node, which manages the slave nodes and assigns the tasks, does not need to be powerful, as it does not perform the actual computation of the job). The `c1.medium` instances are the medium-tier, compute optimized type of machines, at the price of US\$ 0.20/hour each. The `m1.small` ones are smaller instances with more balanced components (e.g. without SSD storage), at the price of US\$ 0.08/hour each. Cost data are as of January, 2014.

EMR is Amazon's MapReduce service. With it, the user is able to instantiate EC2 machines which come with a pre-configured, running Hadoop framework. EMR clusters use Amazon's own modified Hadoop distribution. EMR clusters also exhibit the possibility to be resized in real time, during job execution (i.e. in the case there are exceeding or not sufficient processing slots).

The EC2 machines accessible to the remote user are instantiated virtual machines, thus Amazon describes the processing capacities of each instance type in terms of virtual CPUs (vCPUs), not in terms of actual CPUs. A vCPU is a CPU core allocated for the virtual machine. The `c1.medium` instance type, used in this work, has 1.7 GiB of memory, 2 vCPUs and 350 GB of local instance storage. The two vCPUs basically translate into two Map/Reduce slots per instance. The `m1.small` instance type, also used in this work, has 1.7 GiB of memory, 1 vCPU and 160 GB of local instance storage. No further details on the machines' specifications is given, except that both `c1.medium` and `m1.small` instance types are based on the Intel Xeon family of processors.

There is a standard limit on the utilization of Amazon EC2 instances, which is 20 machines. In this work, the limit has been raised to 235 (234 `c1.medium` processing nodes with 2 vCPUs each, totalling 468 simultaneous threads + 1 `m1.small` master node to track the tasks among slaves), which was a sufficiently high number of machines that we were granted by the Amazon support service and which still was reasonably affordable to simultaneously instantiate.

4.4 Results

4.4.1 SPEC Benchmarks

Baseline

Figure 4.2 shows the execution times for the local processing with OpenMP of the two SPEC benchmarks which were implemented with OpenMR. The benchmarks have been executed multiple times, with an increasing number of threads. The number of threads doubles with each increment, clearly showing a tendency to decrease the execution times by approximately half. The benchmarks were executed on 24-core machine, thus justifying the estagnation of execution times perceived between 24 and 48 threads (the CPUs support execution of two simultaneous threads per core, but since SPEC OMP2012's benchmarks are compute-bound applications, demonstrating little I/O, they take no advantage from thread-level parallelism).

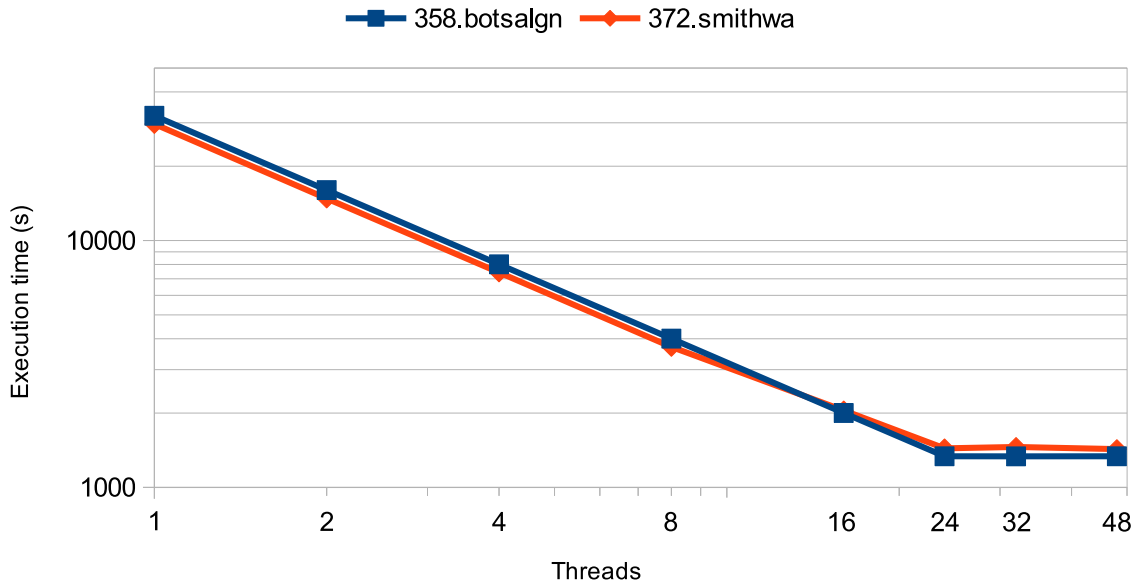


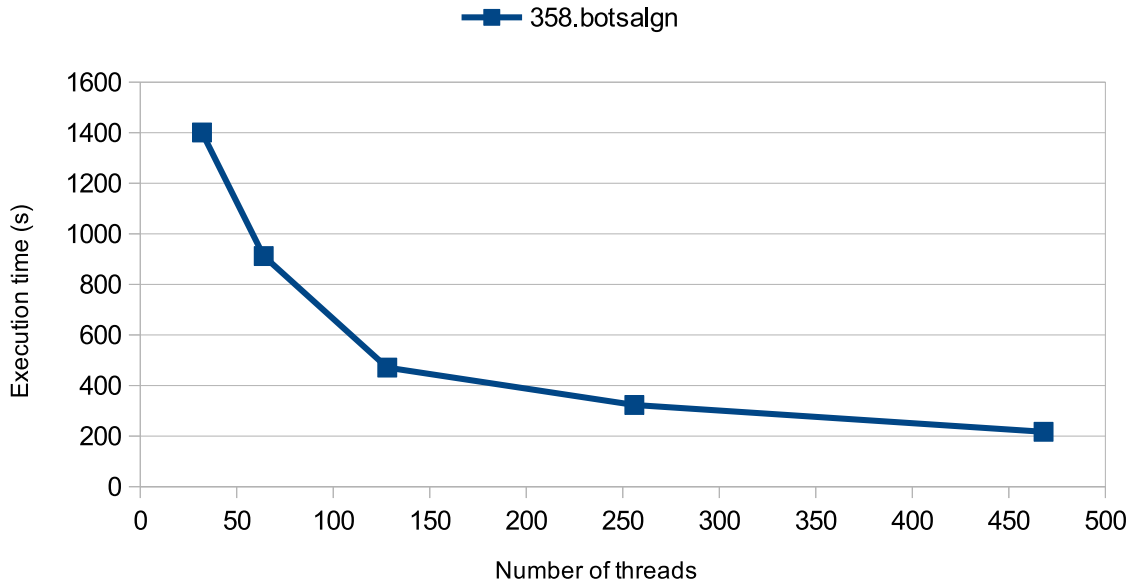
Figure 4.2: Execution times of SPEC benchmarks with OpenMP

358.botsaln + Amazon EMR

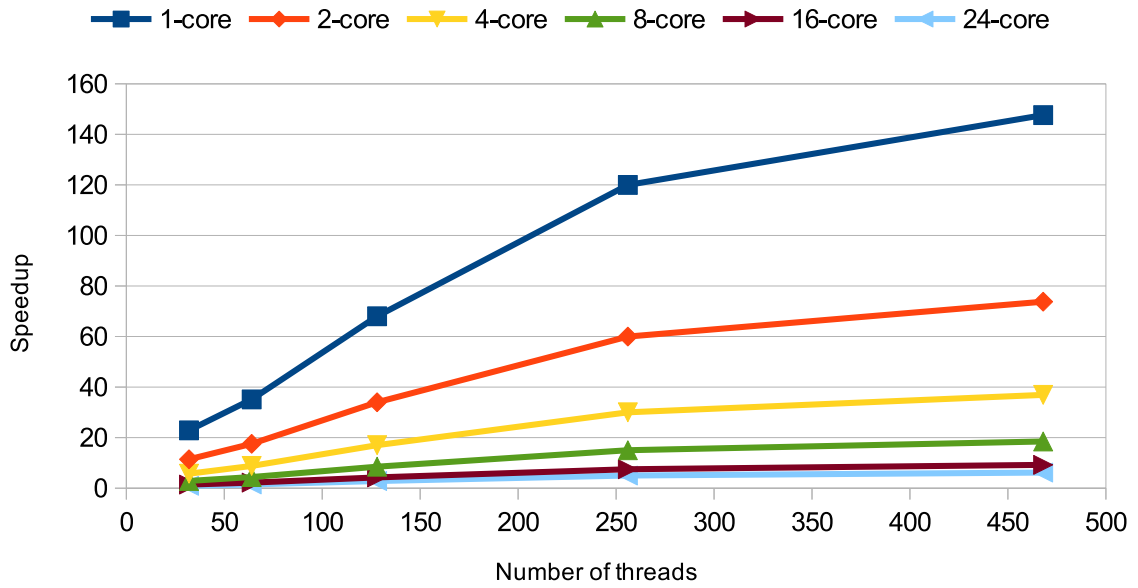
The pattern of figure 4.2 was also noticed when the benchmarks were executed in the cloud scenario using OpenMR. The execution of 358.botsaln in Amazon EMR was profiled as displayed by figure 4.3. In this case, a remote cluster was deployed on demand with a few

processing nodes and then gradually resized, doubling the number of available threads each time. It has also been assumed that the input data was previously available remotely. It is interesting to notice that the pattern from the previous case was successfully reproduced, as expected. Here, as we double the number of cores to process the benchmark, the execution time reduces accordingly. It only does not reproduce the exact same behavior of the previous context because there is plenty of associated overhead which represents a fixed portion of the time. The execution with 468 threads achieved the lowest execution time, 217 seconds, while the best execution time achieved by the baseline scenario using OpenMP was 1336 seconds, with 24 threads. That represents a speedup of $1336/217 = 6.15x$. Chart 4.3b shows the speedups achieved with respect to the baseline scenario, with various amounts of processing cores.

Figure 4.4, in turn, shows additional analyses performed over the same benchmark. In this case, we consider not only the processing times, but also the overhead implied in generating the input data and sending them to the remote cluster. As the network overhead is variable and depends much on the available infrastructure, we performed sample executions considering those steps separately, isolating the tasks of input data generation and communication. Each step has been executed three times and we considered the average execution time for each of these steps. Chart 4.4a shows that the best execution time including data generation and communication (2139 seconds) was better than the execution of the local OpenMP version with 8 threads (4008 seconds), even though the actual processing time was equivalent to only less than 10% of the total. Actually, it almost outperformed the local execution with 16 threads (2002 seconds). That demonstrates that OpenMR's distributed processing may achieve better results than local execution even in cases where the data are not previously available in the cloud. Just like in the previous experiment, chart 4.4b shows the speedups achieved by total execution times with respect to the baseline scenario.

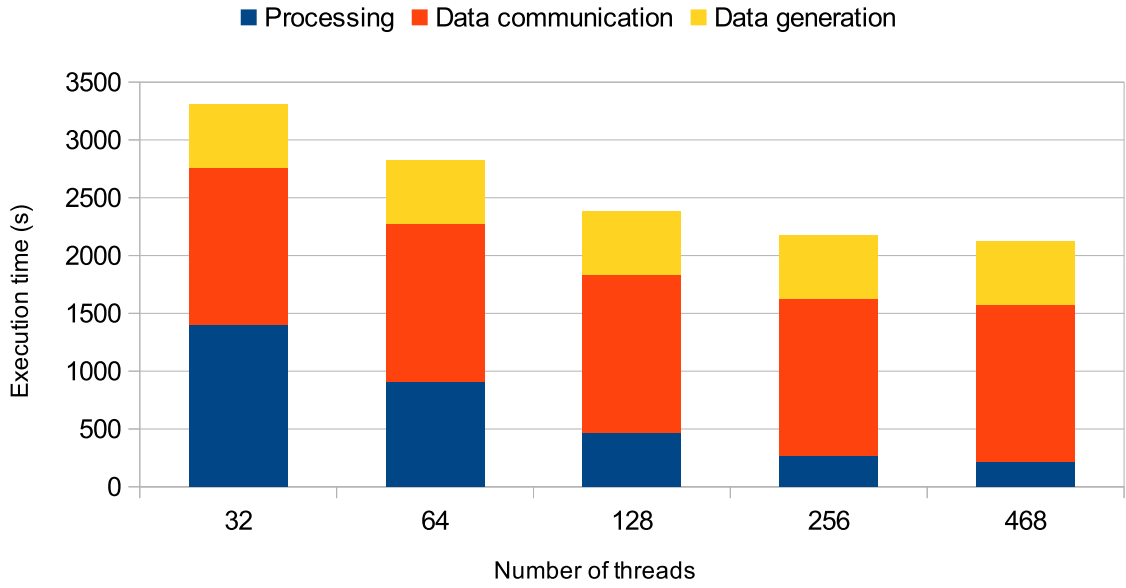


(a) Absolute execution times

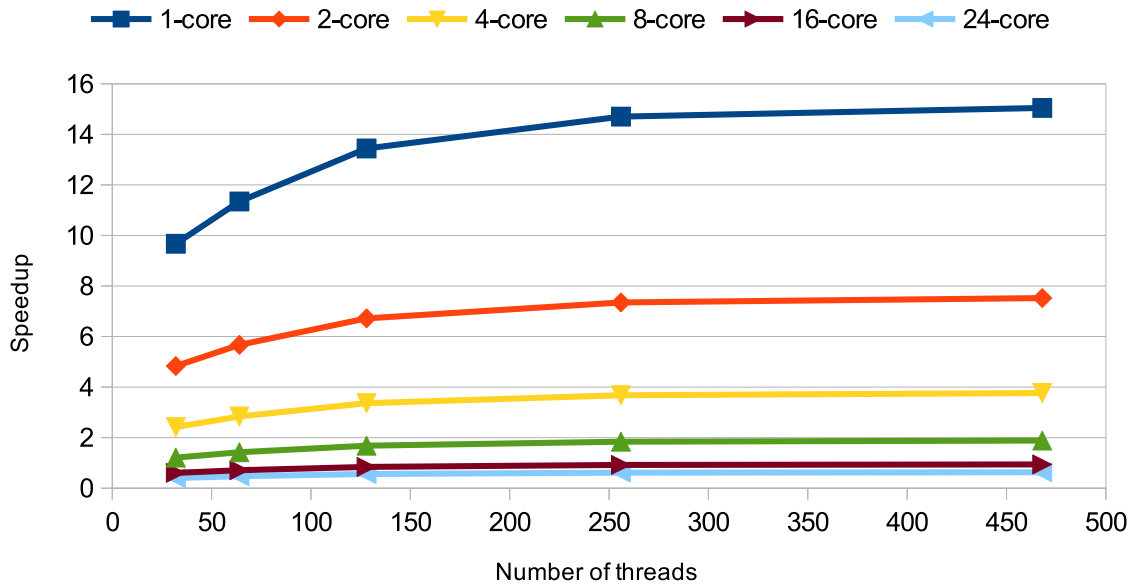


(b) Speedups relative to the OpenMP local execution

Figure 4.3: Execution of 358.botsaln with OpenMR and Amazon EMR



(a) Absolute execution times



(b) Speedups relative to the OpenMP local execution

Figure 4.4: Execution of 358.botsalgn with OpenMR and Amazon EMR + additional overheads

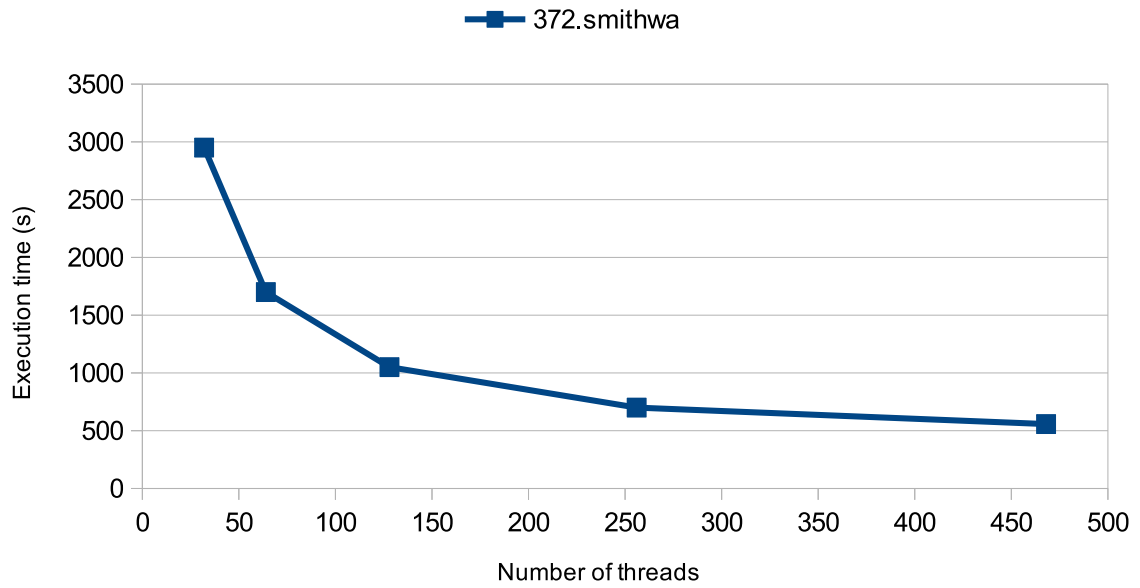
372.smithwa + Amazon EMR

The execution of 372.smithwa was also performed in the cloud. Figure 4.5 shows the results. This experiment also considered the data as previously available remotely and its execution followed the same procedures as the previous one. The lowest execution time for this benchmark was 557 seconds, while the best OpenMP time, with 24 threads, was 1430 seconds. That represents a speedup of $1430/557 = 2.56x$. Chart 4.5b shows the speedups achieved with respect to the OpenMP local execution, with various amounts of processing cores.

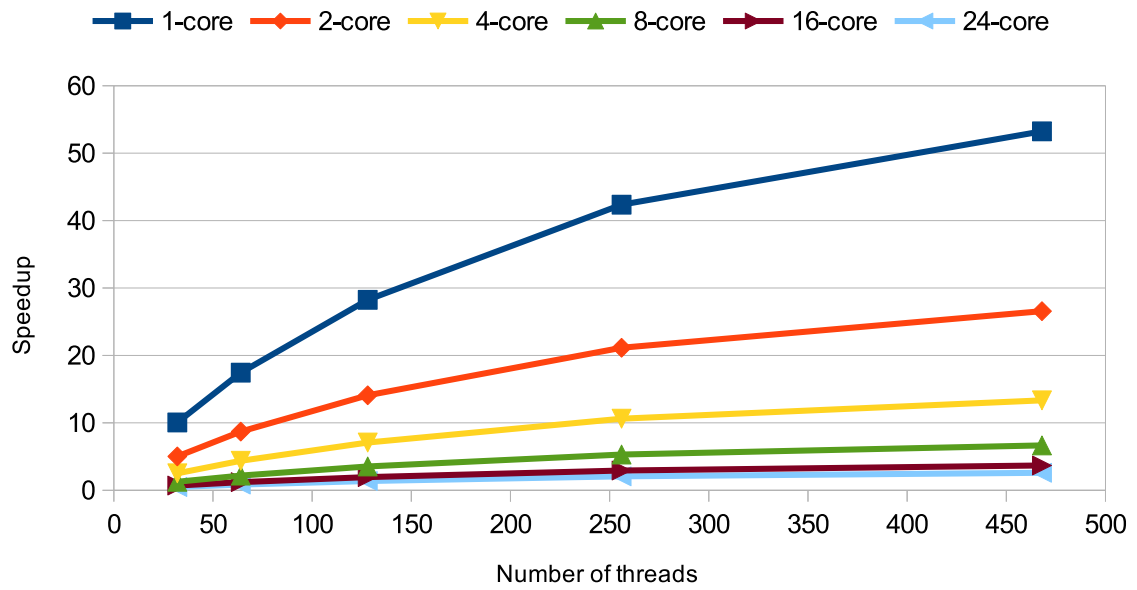
The improvement pattern from the other experiments could also be noticed in this one, even though in a less prominently fashion. This is mainly because all overhead associated to the deployment of a MapReduce job in Amazon EMR is doubled, since the execution of the benchmark consists of two program runs, thus executing the OpenMR region twice.

It is interesting to notice that 372.smithwa does not exhibit a fixed number of threads, but instead partitions the input data among the available threads. Therefore, additional experiments were performed not considering the data previously available remotely, but instead measuring the tradeoff which involves the action of splitting the execution among more threads but having to replicate more data and send them through the network. The results are displayed by figure 4.6.

In this case, chart 4.6a shows that there is an important tradeoff to be considered with respect to distributed processing, since it displays the proportion of each step related to the total execution time. The execution of the benchmark with only 32 threads takes much less time to generate the input data and send them through the network than the execution with more threads, and that was expected. The performance gain of the execution with 468 threads, when compared to the execution with 128 or 256 threads, does not compensate the bigger time spent on data generation and communication, thus totalling a performance loss. Notice that the 256-thread case (1210 seconds) outperforms even the best OpenMP case (1430 seconds), reiterating that even when data are not available remotely, OpenMR might be a good option. Chart 4.6b shows the speedups achieved by this benchmark and its additional overheads with respect to the OpenMP local execution.

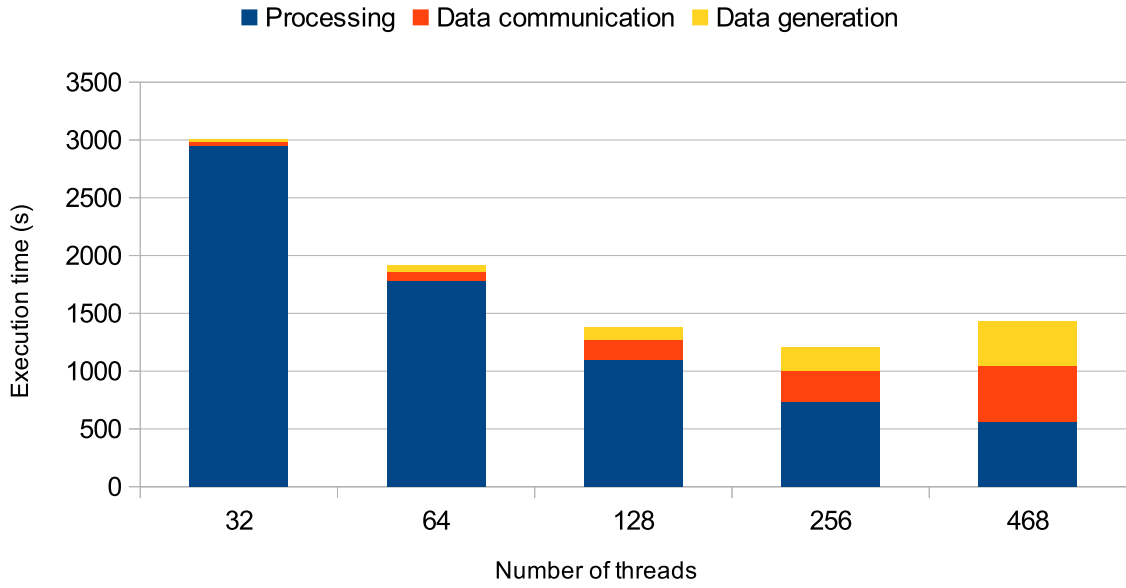


(a) Absolute execution times

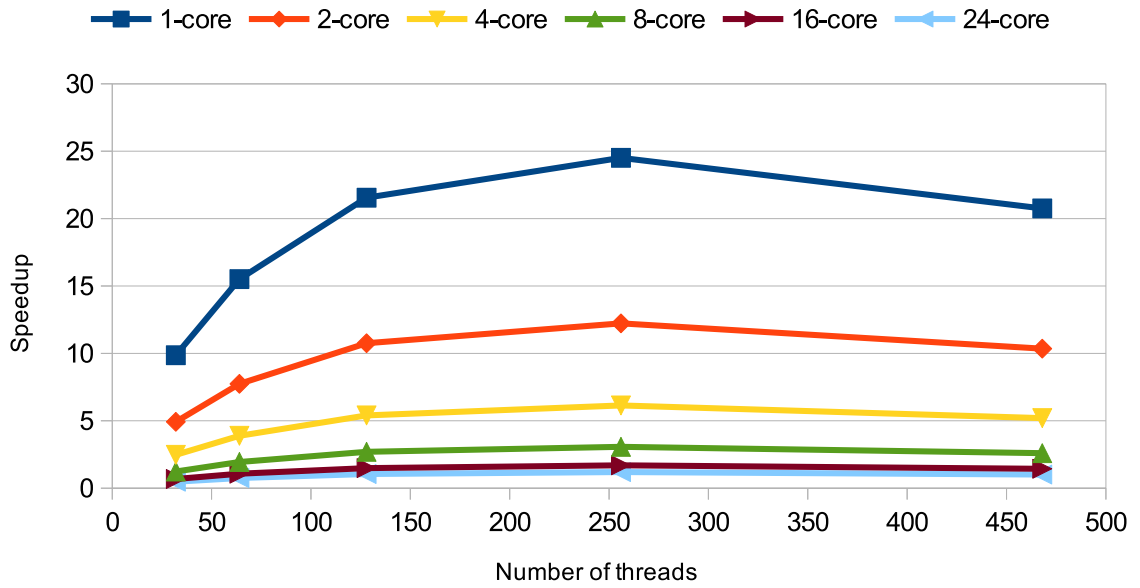


(b) Speedups related to the OpenMP local execution

Figure 4.5: Execution of 372.smithwa with OpenMR and Amazon EMR



(a) Absolute execution times



(b) Speedups related to the OpenMP local execution

Figure 4.6: Execution of 372.smithwa with OpenMR and Amazon EMR + additional overheads

Local cluster scenario

The SPEC 358.botsalgn and 372.smithwa benchmarks have also been assessed in a locally available cluster. Experiments with the various logical thread counts and data distribution have been executed. The results are displayed in figure 4.7. The figure displays the different measurements of execution time for the different steps on the different combinations. As explained previously, 372.smithwa relies on a data set partitioning, given the number of threads. Since the local cluster experiments have all been performed with a fixed number of processing cores (namely, 56), we kept the manual data replication and workload partitioning amounts the same as in Amazon EMR. That means that 372.smithwa has been executed multiple times with the various logical thread counts from the previous experiments over the 56-core cluster. In this chart there has also been considered the step of inserting the input data into the distributed file system (HDFS), step that did not occur with the use of Amazon’s S3 service in the previous experiments. That is because HDFS in this local cluster is a logical file system, not a real fully-distributed storage solution, such as S3. The benchmarks axis is labeled with the number of the benchmarks plus the number of logical threads in the cases that this applies.

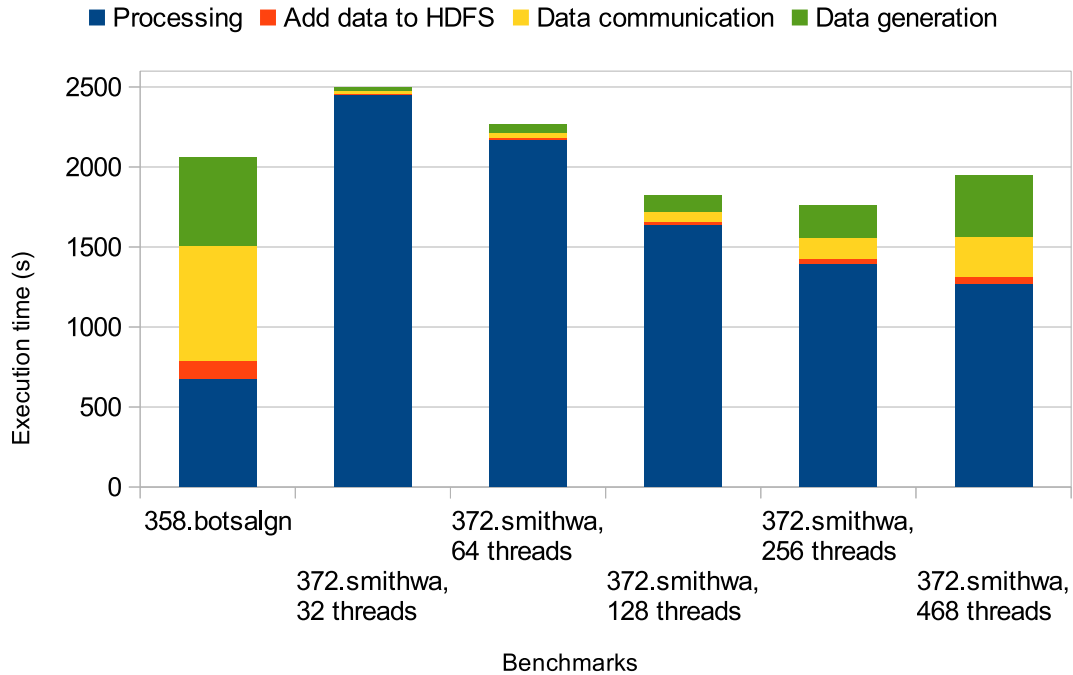
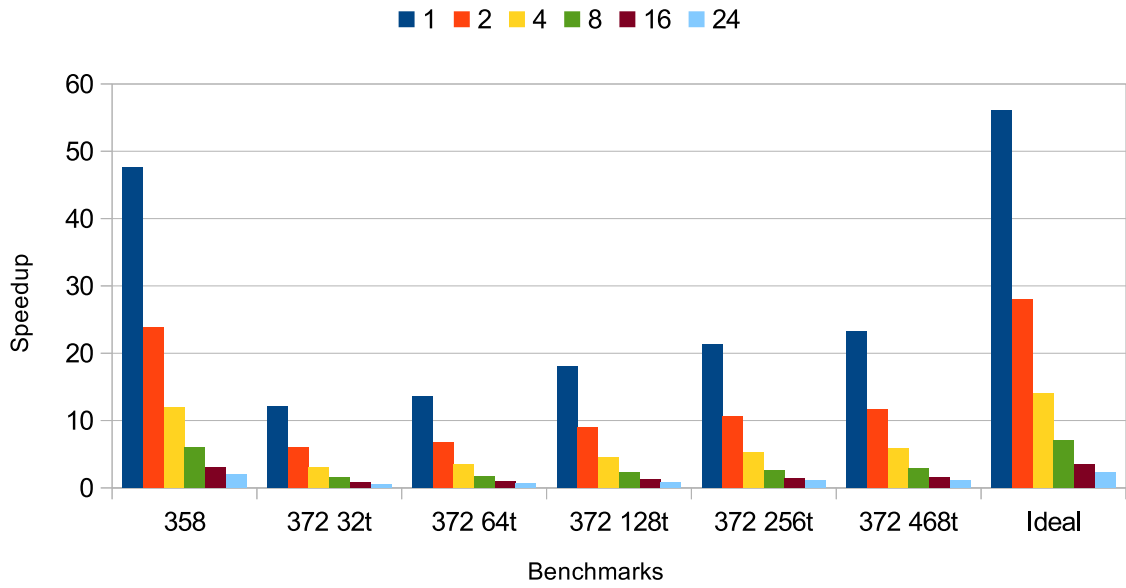


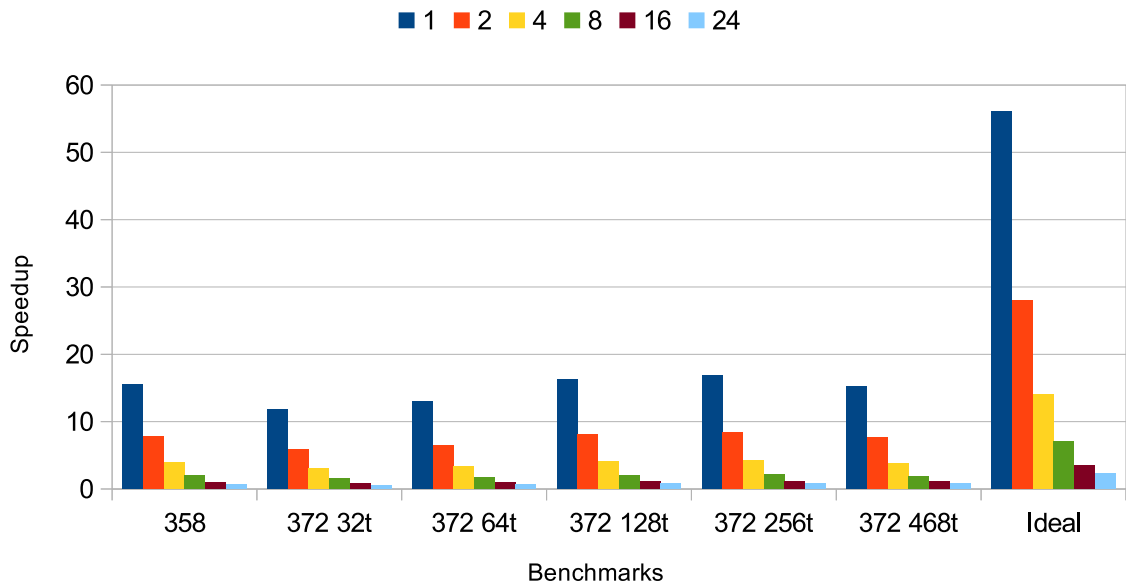
Figure 4.7: Execution times of 358.botsalgn and 372.smithwa with OpenMR in a local cluster, with 56 processing cores

Figure 4.8 shows the speedups achieved by the SPEC benchmarks when executed in a local cluster, relative to the execution times obtained by the local execution. Chart 4.8a shows the speedups considering the processing times only. Chart 4.8b, in turn, also considers the additional data distribution overheads involved. The latter shows that all executions exhibit similar speedup trends. That is because, disregarding communication overheads, the processing speedups are almost linear, thus favoring the parallel execution of the compute-bound applications. The former, meanwhile, shows that 358.botsalgn’s speedups are interestingly closer to the ideal than 372.smithwa’s. Again, 372.smithwa stays farther from the ideal speedups than 358.botsalgn because one execution of the benchmark equals the execution of its OpenMR region twice, thus doubling the MapReduce overhead. In this experiment, all executions have been performed in a local cluster with a total of 56 processing cores.

In the local cluster scenario, it is interesting to notice that 358.botsalgn took complete advantage of the 56 cores, while 372.smithwa did not. It happens that Hadoop performs an analysis at the beginning of each job, through a non-overridable heuristic, which aims to balance the workload among the slave nodes. This heuristic assumes that the Map and Reduce functions have a finer granularity and execute in something like a minute or less over a big set of various data, which is the normal behaviour of common MapReduce jobs. However, the SPEC benchmarks are compute-bound and subvert that logic by implementing Map functions that may last tens of minutes. In our experiments, as we dealt with no data replication inside HDFS, we concluded that 372.smithwa’s cluster underutilization happened because Hadoop calculated that it would be better if less machines executed the job, so that there would not be as much communication to take the data to the nodes that did not have them. It basically tries to reduce the impact of data communication during runtime (one of MapReduce’s assumption is that it is easier to move computation to where the data are present, not the opposite). In the case of 372.smithwa, only 48 cores participated in the processing, because the data was not evenly distributed among the nodes. This, unfortunately, diminished the performance. As we may notice, 358.botsalgn has a fixed (and larger) number of finer-grained iterations, 3000, so it did not suffer from this problem. The issue did not affect the experiments in Amazon AWS either, since the data is stored in a service which is evenly distant from all nodes, not privileging any of them and fully exploring the cluster’s capacity.



(a) Considering processing times only



(b) Considering additional overheads

Figure 4.8: Speedups of 358.botsaln and 372.smithwa with OpenMR in a local cluster, relative to the varying amount of baseline's threads

4.4.2 Toy Benchmarks

Baseline

The toy benchmarks represent a different type of applications, which are I/O-bound. The subversion of MapReduce’s logic as in the previous SPEC benchmarks does not apply to them. Like the other benchmarks, they have also been executed in the cloud and in a local machine. Figure 4.9 presents their execution times in a local machine with 2 cores. These benchmarks process data sets ranging from ~ 100 GB to ~ 200 GB each. They consist mostly of OpenMP loops which process data that are stored in the disk.

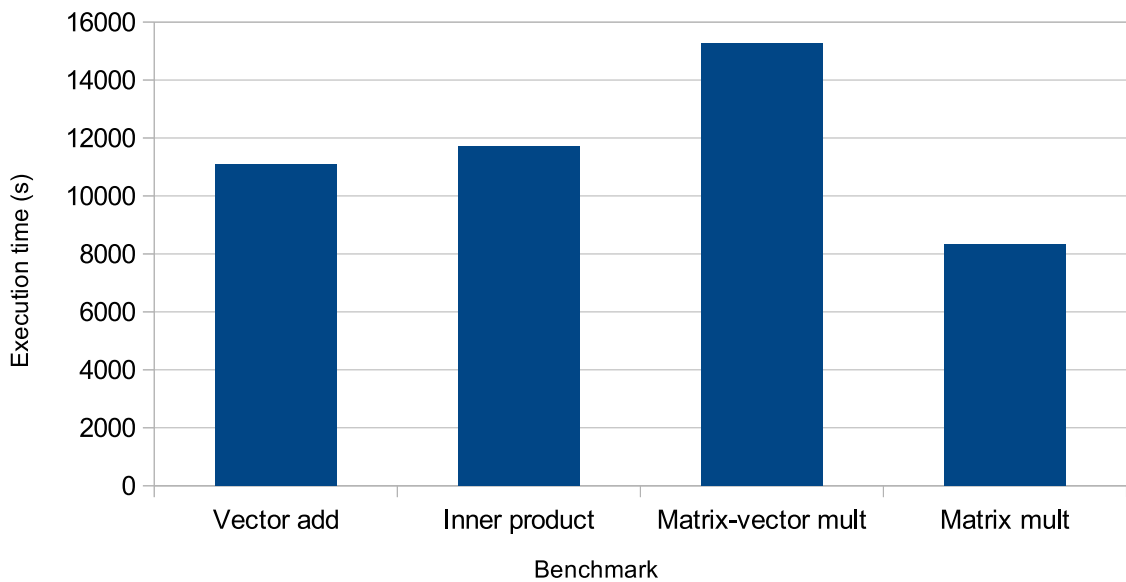


Figure 4.9: Execution times of toy benchmarks in a local, 4-core machine

Cloud execution

Since the data do not fit the main memory of the machine, the processing is bound by the disk read rate, which becomes a bottleneck. This can be noticed when executing the benchmarks locally without the OpenMP parallelization, because the execution times remain roughly the same. That bottleneck diminishes the performance gain obtained by OpenMP. This, in turn, represents the case of best potential for OpenMR, because the distributed processing with MapReduce relies on an also distributed file system, which drastically reduces the data access bottleneck. This way, every processing node can obtain

its portion of data to process in parallel, effectively removing the serialization imposed by the disk in a non-distributed execution.

Figures 4.10 and 4.11 display the analyses performed over the set of toy benchmarks. Figure 4.10 shows that the performance gains follow the expected trend, being roughly reduced by half everytime the number of available threads is doubled. Only processing times are considered, since the nature of the benchmarks is to process previously available data (thus not considering data generation or communication). The speedups achieved with respect to the local execution are demonstrated by figure 4.11. The execution of inner product was the one which attained the best results: with 468 threads, it was performed in 124 seconds, representing a speedup of $11723/124 = 94.5x$. That is a very high and interesting rate, given that it was achieved by the execution with $468/4 = 117x$ more processing cores.

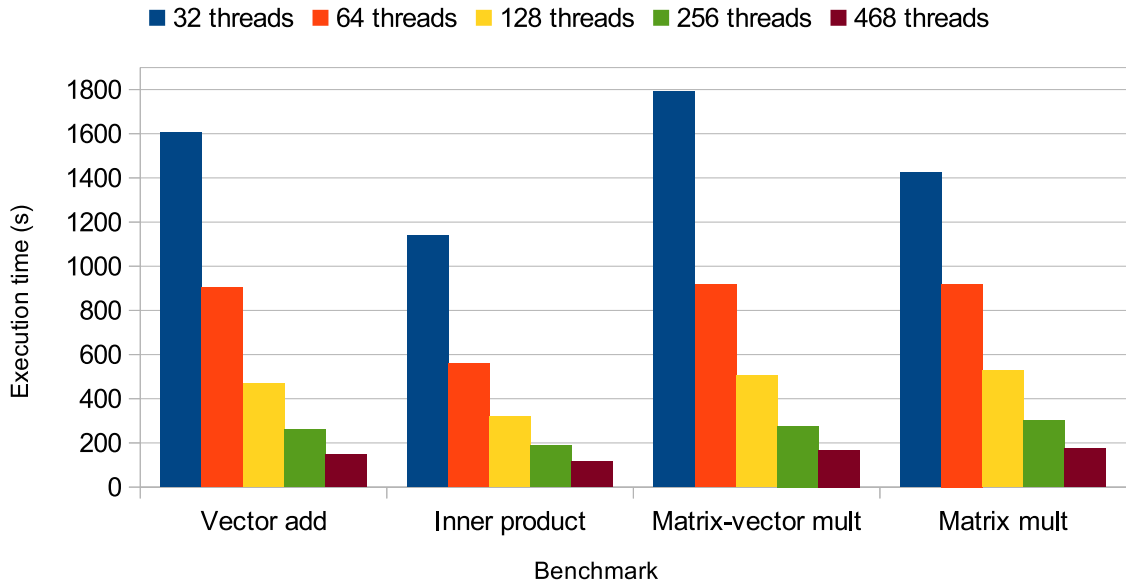


Figure 4.10: Absolute execution times of toy benchmarks in Amazon EMR

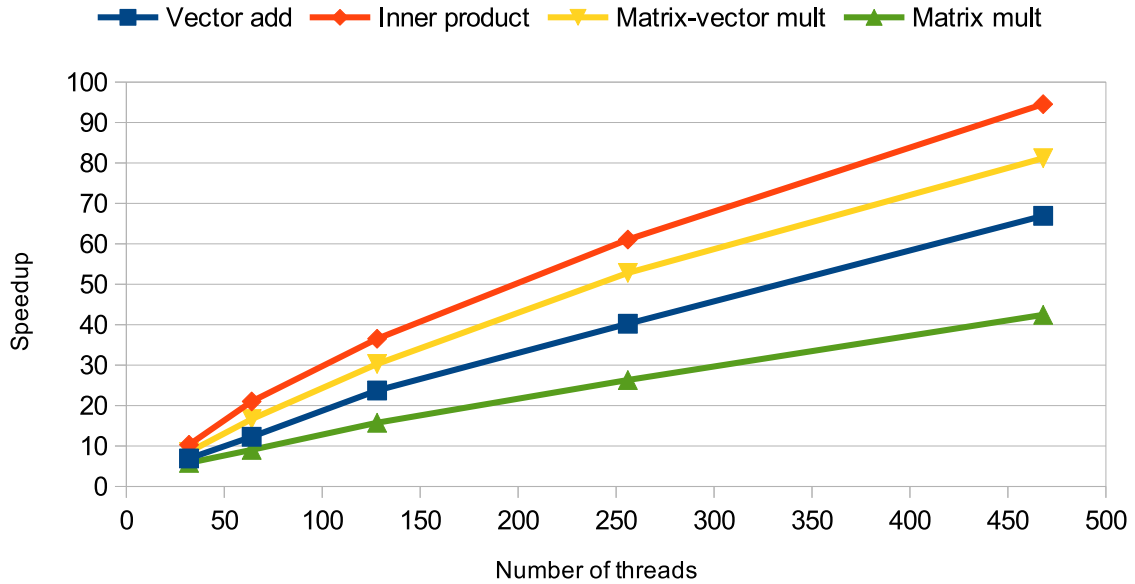


Figure 4.11: Speedups of toy benchmarks in Amazon EMR relative to local processing

The execution of benchmarks with OpenMR and Amazon EMR, when compared to their local, OpenMP versions, has been demonstrated to perform well. The speedups, however, kept a sub-optimal pattern. That relates to the overheads inherent to distributed/remote processing, including networking delays, data communication and the MapReduce framework's overhead. One clear example of this is 358.botsalgn, which executed locally (OpenMP) in 1336 seconds with 24 threads and remotely (OpenMR + Amazon EMR) in 217 seconds with 468 threads. The speedup achieved in this case is $1336/217 = 6.15x$. However, as there were $468/24 = 19.5x$ more threads executing, that represents a $6.15/19.5 = 0.31x$ efficiency (speedup per thread added, the ideal is 1). This rate is tightly coupled to the workload and the cluster infrastructure.

Experimentally, one may perceive a delay of 2+ minutes between the triggering of the OpenMR job and the actual start of the execution in Amazon EMR. If that overhead was to be disregarded, the speedup achieved would be much higher. This means that such delay can be blamed for most of the loss of potential speedup. Say that, for instance, a 120 second overhead was imposed by that starting delay in Amazon EMR. That would mean that the actual execution time was about 97 seconds, and the $1336/97 = 13.77x$ speedup achieved would be much more significant than the original 6.15x. This, in turn, would represent an efficiency of $13.77/19.5 = 0.7x$.

This is particularly interesting because the experiments with a local cluster demon-

strated to incur much less of that kind of overhead. The same benchmark, 358.botsalgn, has been executed in 673 seconds. Given that only 56 threads were available, that represents a $1336/673 = 1.98x$ speedup, with $56/24 = 2.33x$ more threads than the baseline. The efficiency is $1.98/2.33 = 0.85x$, far better than the previous cases of $0.31x$ and $0.7x$. That shows that the locality of the cluster is very relevant with respect to performance with OpenMR.

Also, these numbers are influenced by the workload. If the benchmark took significantly longer to execute, the overhead would mean a significantly smaller portion of the execution time, thus bringing the speedup curve closer to the optimal.

Chapter 5

Conclusions

This dissertation presents OpenMR, an extension to the OpenMP API which enables the parallel, distributed execution of loops in the cloud with MapReduce through directive-annotated code.

The feasibility of the proposed execution model has been assessed through the execution of SPEC OMP2012 benchmarks, which have been modified to represent OpenMR-equivalent code that would be transparently generated by an OpenMR-enabled compiler, and by toy benchmarks developed specifically to represent I/O-bound applications. The benchmarks have been executed in three different computation contexts that have been compared, namely: local execution with OpenMP, distributed execution in a local cluster and distributed execution in remote clusters using Amazon AWS services. The experimental results allowed an interesting analysis of the tradeoffs involved in workload and data distribution and show that OpenMR achieves good performance and good scalability.

OpenMR includes the proposal of the **data** construct, which allows for the execution of parallel code in the cloud with input data which is previously available remotely. OpenMR may be seen as an abstraction layer which enables the developer to look to the cloud as an additional layer in the computation stack, and the **data** construct may be seen as an initial step towards the inclusion of the cloud also as an additional layer of the memory hierarchy stack.

5.1 Future Work

The following is a highlight of related research topics which, to the best of our understanding, are promising and relevant to the near future.

- *The data construct*: despite having been proposed in this work and its existence having been assumed during the execution of our experiments, the **data** construct

presents many research problems that need to be solved before its implementation. For instance, how could such a mechanism interact with the cloud infrastructure of various vendors and with the OS? Is it even possible to implement a mechanism which abstracts from the developer data input/output to/from the cloud? If so, what are the tradeoffs and behaviors of such a mechanism?

- *Further integration of the cloud to the computation/memory stack:* as previously stated, having the cloud to constitute a higher computation stack layer suggests the use of a cloud-aware caching mechanism. Given that we are to look to the cloud as a natural extension of the computing stack, one might wonder if there is some way to have cache mechanisms to be aware of remote storage clouds and communicate with them. How would such a mechanism behave? What are the tradeoffs? How could a cloud-aware cache coherence protocol be defined?
- *Automatic scattering of workloads among computation stack layers:* considering the stack which comprises the various levels of hardware in which computation may take place (CPUs, GPUs/accelerators, the cloud and so on), is there some way to automatically spread workloads across such layers, considering the availability of resources and workload characteristics? Are there heuristics which can successfully determine the potential gains of using various levels of parallelism granularity and split the computation in chunks, accordingly? Also, how could such a mechanism take advantage of heterogeneous cloud machines?
- *OpenMR model optimizations.* The movement of input data to the cloud currently remains reasonably inefficient, because the replication of data among processing nodes is not performed in Distributed File System level, but the data are sent to the cloud previously replicated instead. This behavior was necessary to guarantee that processing nodes could be provided with all necessary variables used during the loop. Unfortunately, this incurred large network latencies which harmed the performance. Is there some way to avoid this kind of expensive, superfluous overhead?
- *Extension of OpenMR's model:* OpenMP's synchronization constructs are a very important part of the API and are widely used by a number of applications. However, that type of constructs does not naturally fit the MapReduce paradigm, preventing OpenMR to cover a large portion of preexistent OpenMP codes. If such constructs can be modeled with OpenMR, is it possible to have a one-to-one mapping of OpenMP constructs to OpenMR constructs, i.e. to automatically map OpenMP applications to the cloud? As a result, applications such as the remaining benchmarks from the SPEC OMP2012 suite could be eligible for cloud parallelization.

Bibliography

- [1] Hussam Abu-Libdeh, Lonnie Princehouse, and Hakim Weatherspoon. RACS: A Case for Cloud Storage Diversity. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 229–240, New York, NY, USA, 2010. ACM.
- [2] Gene M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.
- [3] OpenMP Architecture Review Board. OpenMP Application Program Interface Version 3.0 Specification. <http://www.openmp.org/mp-documents/spec30.pdf>, May 2008.
- [4] OpenMP Architecture Review Board. OpenMP website. <http://openmp.org/>, December 2013.
- [5] T.C. Bressoud and M.A. Kozuch. Cluster fault-tolerance: An experimental evaluation of checkpointing and MapReduce through simulation. In *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, pages 1–10, Aug 2009.
- [6] Michael Cardosa, Chenyu Wang, Anshuman Nangia, Abhishek Chandra, and Jon Weissman. Exploring MapReduce Efficiency with Highly-distributed Data. In *Proceedings of the Second International Workshop on MapReduce and Its Applications*, MapReduce '11, pages 27–34, New York, NY, USA, 2011. ACM.
- [7] Robit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.
- [8] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An Object-oriented Approach to Non-uniform Cluster Computing. In *Proceedings of the 20th*

- Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM.
- [9] Rong Chen, Haibo Chen, and Binyu Zang. Tiled-MapReduce: Optimizing Resource Usages of Data-parallel Applications on Multicore with Tiling. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 523–534, New York, NY, USA, 2010. ACM.
- [10] Brian Cho, Muntasir Rahman, Tej Chajed, Indranil Gupta, Cristina Abad, Nathan Roberts, and Philbert Lin. Natjam: Design and Evaluation of Eviction Policies for Supporting Priorities and Deadlines in Mapreduce Clusters. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 6:1–6:17, New York, NY, USA, 2013. ACM.
- [11] D. Clark. OpenMP: a parallel standard for the masses. *Concurrency, IEEE*, 6(1):10–12, 1998.
- [12] D. Cordeiro. Apache Hadoop: Conceitos teóricos e práticos, evolução e novas possibilidades. Escola Regional de Alto Desempenho de São Paulo (ERAD/SP), <http://erad.lsc.ic.unicamp.br/slides/erad-hadoop-DanielCordeiro.pdf>, July 2012.
- [13] David E. Culler, Anoop Gupta, and Jaswinder Pal Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1997.
- [14] David Cunningham, David Grove, Benjamin Herta, Arun Iyengar, Kiyokuni Kawachiya, Hiroki Murata, Vijay Saraswat, Mikio Takeuchi, and Olivier Tardieu. Resilient X10: Efficient Failure-aware Programming. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '14, pages 67–80, New York, NY, USA, 2014. ACM.
- [15] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science Engineering, IEEE*, 5(1):46–55, 1998.
- [16] Jeff Dean. Designs, Lessons and Advice from Building Large Distributed Systems. The 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware (LADIS), <http://odbms.org/download/dean-keynote-ladis2009.pdf>, October 2009.
- [17] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Operating*

- Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.
- [18] Pantazis Deligiannis, Hans-Wolfgang Loidl, and Evangelia Kouidi. Improving the Diagnosis of Mild Hypertrophic Cardiomyopathy with MapReduce. In *Proceedings of Third International Workshop on MapReduce and Its Applications Date*, MapReduce '12, pages 41–48, New York, NY, USA, 2012. ACM.
 - [19] R.H. Dennard, F.H. Gaensslen, V.L. Rideout, E. Bassous, and A.R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *Solid-State Circuits, IEEE Journal of*, 9(5):256 – 268, oct 1974.
 - [20] Mengwei Ding, Long Zheng, Yanchao Lu, Li Li, Song Guo, and Minyi Guo. More Convenient More Overhead: The Performance Evaluation of Hadoop Streaming. In *Proceedings of the 2011 ACM Symposium on Research in Applied Computation*, RACS '11, pages 307–313, New York, NY, USA, 2011. ACM.
 - [21] Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick. *UPC: Distributed Shared-Memory Programming*. Wiley-Interscience, 2003.
 - [22] H. Esmailzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark Silicon and the End of Multicore Scaling. *Micro, IEEE*, 32(3):122 –134, may-june 2012.
 - [23] M. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, 1966.
 - [24] M. Flynn. Some Computer Organizations and Their Effectiveness. *Computers, IEEE Transactions on*, C-21(9):948–960, 1972.
 - [25] The Apache Software Foundation. Apache Hadoop. <http://hadoop.apache.org/>, February 2014.
 - [26] Guilherme Galante and Luis C.E. Bona. Supporting Elasticity in OpenMP Applications. In *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on*, pages 188–195, Feb 2014.
 - [27] Yasser Ganjisaffar, Thomas Debeauvais, Sara Javanmardi, Rich Caruana, and Cristina Videira Lopes. Distributed Tuning of Machine Learning Algorithms Using MapReduce Clusters. In *Proceedings of the Third Workshop on Large Scale Data Mining: Theory and Applications*, LDMTA '11, pages 2:1–2:8, New York, NY, USA, 2011. ACM.

- [28] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.
- [29] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A High-performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Comput.*, 22(6):789–828, September 1996.
- [30] Zhenhua Guo, Geoffrey Fox, and Mo Zhou. Investigation of Data Locality and Fairness in MapReduce. In *Proceedings of Third International Workshop on MapReduce and Its Applications Date*, MapReduce '12, pages 25–32, New York, NY, USA, 2012. ACM.
- [31] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.
- [32] Hui Jin, Kan Qiao, Xian-He Sun, and Ying Li. Performance under Failures of MapReduce Applications. In *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on*, pages 608–609, May 2011.
- [33] Khronos Group. The OpenCL Specification Version 2.0. <http://www.khronos.org/registry/cl/specs/openc1-2.0.pdf>, November 2013.
- [34] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee. SnuCL: An OpenCL Framework for Heterogeneous CPU/GPU Clusters. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 341–352, New York, NY, USA, 2012. ACM.
- [35] T. Kuroda. CMOS design challenges to power wall. In *Microprocesses and Nanotechnology Conference, 2001 International*, pages 6 –7, 2001.
- [36] Lawrence Livermore National Labpratory. OpenMP tutorial. <https://computing.llnl.gov/tutorials/openMP/>, December 2013.
- [37] Simone Leo, Luca Pireddu, and Gianluigi Zanetti. SNP Genotype Calling with MapReduce. In *Proceedings of Third International Workshop on MapReduce and Its Applications Date*, MapReduce '12, pages 49–56, New York, NY, USA, 2012. ACM.
- [38] T. Mattson and L. Meadows. A 'Hands-on' Introduction to OpenMP. <http://openmp.org/mp-documents/omp-hands-on-SC08.pdf>, 2008.

- [39] Paras Mehta. Usability of Unified Parallel C. Technical Report TR04-20, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, 2004.
- [40] Rohith K. Menon, Goutham P. Bhat, and Michael C. Schatz. Rapid Parallel Genome Indexing with MapReduce. In *Proceedings of the Second International Workshop on MapReduce and Its Applications*, MapReduce '11, pages 51–58, New York, NY, USA, 2011. ACM.
- [41] M.A. Miller. *Implementing and Understanding Algorithms in Unified Parallel C*. Michigan Technological University, 2008.
- [42] Matthias S. Müller, John Baron, William C. Brantley, Huiyu Feng, Daniel Hackenberg, Robert Henschel, Gabriele Jost, Daniel Molka, Chris Parrott, Joe Robichaux, Pavel Shelepugin, Matthijs van Waveren, Brian Whitney, and Kalyan Kumaran. SPEC OMP2012 – an Application Benchmark Suite for Parallel Systems Using OpenMP. In *Proceedings of the 8th International Conference on OpenMP in a Heterogeneous World*, IWOMP'12, pages 223–236, Berlin, Heidelberg, 2012. Springer-Verlag.
- [43] Eugene W. Myers and Webb Miller. Optimal alignments in linear space. *CABIOS*, 4:11–17, 1988.
- [44] D. Novillo. From Source to Binary: The Inner Workings of GCC. <http://www.redhat.com/magazine/002dec04/features/gcc/>, 2004.
- [45] Robert W. Numrich and John Reid. Co-array Fortran for Parallel Programming. *SIGPLAN Fortran Forum*, 17(2):1–31, August 1998.
- [46] NVIDIA. CUDA C Programming Guide Version 5.5. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, July 2013.
- [47] OpenACC. The OpenACC Application Program Interface Version 2.0. http://www.openacc.org/sites/default/files/OpenACC.2.0a_1.pdf, August 2013.
- [48] Peter Pacheco. *An Introduction to Parallel Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2011.
- [49] Jeeva Paudel and José Nelson Amaral. Using the Cowichan Problems to Investigate the Programmability of X10 Programming System. In *Proceedings of the 2011 ACM SIGPLAN X10 Workshop*, X10 '11, pages 4:1–4:10, New York, NY, USA, 2011. ACM.

- [50] Fred J. Pollack. New microarchitecture challenges in the coming generations of CMOS process technologies (keynote address)(abstract only). In *Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, MICRO 32, pages 2–, Washington, DC, USA, 1999. IEEE Computer Society.
- [51] Walter G. Rudd. X3H5 Parallel Extensions for Programming Language C. Technical report, Corvallis, OR, USA, 1993.
- [52] Vijay Saraswat, George Almasi, Ganesh Bikshandi, Calin Cascaval, David Cunningham, David Grove, Sreedhar Kodali, Igor Peshansky, and Olivier Tardieu. The Asynchronous Partitioned Global Address Space Model. In *Proceedings of The First Workshop on Advances in Message Passing*, Toronto, Canada, June 2010.
- [53] Min Si, Y. Ishikawa, and M. Tatagi. Direct MPI Library for Intel Xeon Phi Co-Processors. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 816–824, 2013.
- [54] T. F. Smith and M. S. Waterman. Identification of Common Molecular Subsequences. *Journal of molecular biology*, 147(1):195–197, March 1981.
- [55] M. Sugawara, S. Hirasawa, K. Komatsu, H. Takizawa, and H. Kobayashi. A Comparison of Performance Tunabilities between OpenCL and OpenACC. In *Embedded Multicore Socs (MCSoc), 2013 IEEE 7th International Symposium on*, pages 147–152, 2013.
- [56] Sayantan Sur, Matthew J. Koop, and Dhabaleswar K. Panda. High-performance and Scalable MPI over InfiniBand with Reduced Memory Usage: An In-depth Performance Analysis. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.
- [57] Michele Tartara and Stefano Crespi Reghizzi. Parallel Iterative Compilation: Using MapReduce to Speedup Machine Learning in Compilers. In *Proceedings of Third International Workshop on MapReduce and Its Applications Date*, MapReduce '12, pages 33–40, New York, NY, USA, 2012. ACM.
- [58] Michael B. Taylor. Is Dark Silicon Useful? Harnessing The Four Horsemen Of The Coming Dark Silicon Apocalypse. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 1131–1136, New York, NY, USA, 2012. ACM.
- [59] R. M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, 11(1):25–33, 1967.

- [60] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation cores: reducing the energy of mature computations. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, pages 205–218, New York, NY, USA, 2010. ACM.
- [61] Haiping Wang, Xiaofeng Meng, and Yunpeng Chai. Efficient Data Distribution Strategy for Join Query Processing in the Cloud. In *Proceedings of the Third International Workshop on Cloud Data Management*, CloudDB '11, pages 15–22, New York, NY, USA, 2011. ACM.
- [62] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.
- [63] Gregory V. Wilson. Assessing the Usability of Parallel Programming Systems: The Cowichan Problems. In *In Proceedings of the IFIP Working Conference on Programming Environments for Massively Parallel Distributed Systems*, pages 183–193, 1993.
- [64] Chenning Xie, Zhijun Hao, and Haibo Chen. X10-FT: Transparent Fault Tolerance for APGAS Language and Runtime. In *Proceedings of the 2013 International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '13, pages 11–20, New York, NY, USA, 2013. ACM.
- [65] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: A High-Performance Java Dialect. In *In ACM*, pages 10–11, 1998.
- [66] Shucheng Yu, Cong Wang, Kui Ren, and Wenjing Lou. Achieving Secure, Scalable, and Fine-grained Data Access Control in Cloud Computing. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–9, March 2010.
- [67] Qin Zheng. Improving MapReduce fault tolerance in the cloud. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–6, April 2010.