ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ & ΥΠΟΛΟΓΙΣΤΩΝ

Implementation of a MapReduce Framework on a Network-on-Chip Platform

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Κωνσταντίνος Π. Γυφτάκης

Επιβλέπων : Δημήτριος Σούντρης
Επίκουρος καθηγητής

Αθήνα, Δεκέμβριος 2013

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ & ΥΠΟΛΟΓΙΣΤΩΝ

Implementation of a MapReduce Framework on a Network-on-Chip Platform

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Κωνσταντίνος Π. Γυφτάκης

Επιβλέπων : Δημήτριος Σούντρης
Επίκουρος καθηγητής

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 17η Δεκεμβρίου 2013.

.....	.....	.....
Δημήτριος Σούντρης	Κιαμάλ Πεκμεστζή	Γιώργος Οικονομάκος
Επίκουρος καθηγητής	Καθηγητής	Επίκουρος καθηγητής

Αθήνα, Δεκέμβριος 2013

.....
Κωνσταντίνος Π. Γυφτάκης

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Abstract

The purpose of the present diploma thesis is the implementation of a MapReduce framework on a Network-on-chip that has DSM characteristics. MapReduce is a programming model capable of processing large data sets with a parallel distributed algorithm using a large number of processing nodes. Our objective goal was to determine the feasibility of implementing MapReduce on a many-core embedded system as the NoC described, and evaluate its performance in terms of scalability. Furthermore, we wanted to exploit platform's characteristics in order to provide synchronization and communication among the cores. The proposed framework performed exceptionally, achieving speedups up to x85.2 for 36 cores, when compared to sequential code. Finally, we analyzed the frameworks behaviour while scaling different parameters.

This thesis includes five chapters. Chapter 1 contains an introduction to many-core systems, Big Data and MapReduce, as a parallel programming method. In chapter 2, we present the most popular MapReduce frameworks, alternative methods for parallel processing, the limitations of MapReduce and, finally, discuss our objectives. Chapter 3 consists of a presentation of the platform on which the proposed framework was implemented and a detailed overview of the framework. In chapter 4, we exhibit our test configuration, as well as the results of our simulations accompanied with an analysis. In the end, chapter 5 concludes our work and presents some topics and ideas that need future study.

Keywords: *MapReduce, Network-on-Chip (NoC), Distributed Shared Memory (DSM), Many-core Embedded Systems, Big Data.*

Acknowledgments

First and foremost, I would like to thank my family, for without their constant love and support, I wouldn't be able to accomplish anything. I would, also, like to thank Microprocessors and Digital Systems Lab and especially professor Dimitrios Soudris for the chance to work on this wonderful project. Last but not least, I would like to thank Ph.D. Researcher Iraklis Anagnostopoulos for his sturdy guidance and patience throughout this project.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

1.1. Many-Core Systems

Ever since the microprocessor made its debut at the dawn of the eighth decade of the last century, the computer software industry has enjoyed something of a free ride—free lunch, if you wish [1] —in terms of computing power available for software programs to use. Moore's law, driven to self-fulfilment by the hardware industry's desire to deliver better and better chips, secured doubling number of transistors— and consequently, at least till recently, hardware performance—every two years or so; if you ran into performance problems, those were solved within a few months through the release of newer, faster, better hardware. Writing parallel programs was something that only the high performance computing experts or domain experts working in similar, highly specialized fields of computing had to worry about.

This free ride however came to a sudden halt at around 2004, when the speed of a single processor core topped out at about 4 GHz. Since then, while Moore's law still held (as exemplified in Fig. 1.1) and there were more and more transistors crammed on smaller and smaller areas, the performance per single processor core, measured as the amount of instructions per second it could execute, remained constant or actually started to decline, along with the frequency at which the core was run.

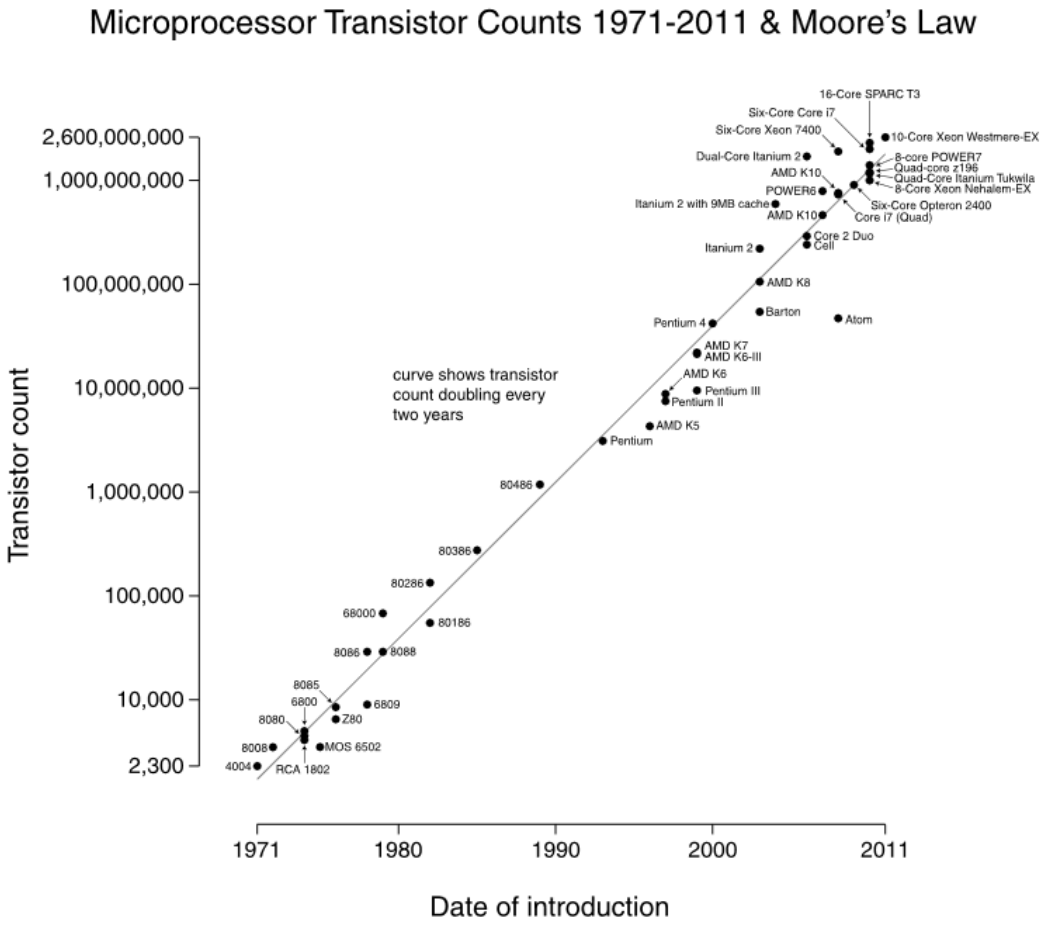


Fig.1.1 Moore's law (Source: Wikipedia)

All major hardware vendors started shipping processors with multiple processor cores, finding thus a new use for the transistors churned out in accordance with Moore's law.

Still, the currently available Multi-core processors with two, four, or eight execution units—"cores"—indicate just the beginning of a new era in which parallel computing stops being a niche for scientist and starts becoming mainstream. Multi-core processors are just the beginning. With the amount of cores increasing further, *Multi-cores* become *many*-cores. The distinction between these two classes of parallel processors is not precisely defined. Multi-core processors typically feature up to 32 powerful cores. Their memory architecture allows the usage of traditional shared memory programming model without suffering from significant performance penalties.

Many-core processors on the other hand comprise more than 64 rather simple and less powerful cores. With an increasing number of cores, a *scalable* on-chip interconnect between cores on the chip becomes a necessity. Memory access, especially to off-chip memory, constitutes a bottleneck and becomes very expensive. Therefore, traditional shared memory architectures and corresponding programming models suffer from significant performance penalties—unless they are specifically optimized. When looking at raw performance alone (total number of instructions executed within a unit of time), more but less powerful cores clearly outperform chips with few but powerful cores, within a set power budget. This is a tempting prospect, especially in domains with abundant parallelism such as networking, graphics, web servers etc. Accordingly, this category of chips—with many, but simpler cores—is usually represented by processors targeting a specific domain.

Some of the best known representatives of this category are Tilera's Tile GX family, NVIDIA's Graphics Processing Units (GPUs), picoChip's 200-core DSP as well as Intel's Many Integrated Core (MIC) architecture [2]. Many-core processors with up to 72 cores on a single chip are currently commercially available [3].

1.1.1. The Tile Architecture

The Tile architecture has its origins in the RAW research processor developed at MIT and later commercialized by Tilera, a start-up founded by the original research group. Chips from the second generation are expected to scale up to 100 cores based on the MIPS ISA and running at 1.5 GHz, within a power budget of under 60 W. The key differentiating technology of the Tile architecture is the on-chip interconnect and the cache architecture. As the name implies, the chip is structured as a 2D array of *tiles,* where each tile contains a processor core, associated L1 (64 kb) and L2 cache (256 kb per core) and an interconnect switch that can connect the tile to its 3 (on the edges) or 4 (inside the mesh) neighboring tiles. This way, the interconnect is essentially a switched network with very short wires connecting neighbouring tiles linked through the tile-local switch.

The interconnect network—called *iMesh* by Tilera—actually consists of five different networks, used for various purposes:

- application process communication (UDN)
- I/O communication (IDN)
- memory communication (MDN)
- cache coherency (TDN)
- static, channelized communication (STN)

The latency of data transfer on the network is 1–2 cycle/tile, depending on whether there's a direction change or not at the tile. The overall architecture of the Tile processor concept is shown in Fig. 1.2

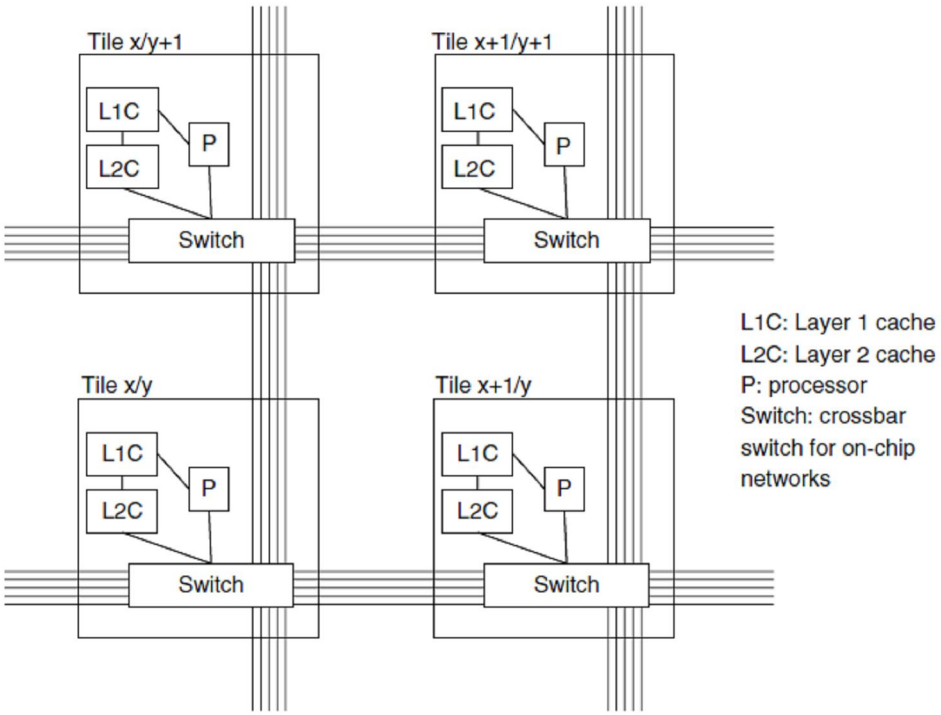


Fig. 1.2 The Tile architecture [2]

1.1.2. Graphics Processing Units

The term Graphics Processing Unit (GPU) was first used back in 1999 by NVIDIA. The introduction of the OpenGL language and Microsoft's DirectX specification resulted in more programmability of the graphics rendering process; eventually this evolution led to the introduction of a GPU architecture (by the same company, NVIDIA) that dramatically improved programmability of these chips using high level (C like) languages and turned this type of processors into an interesting choice for supercomputers targeting massively data parallel applications. NVIDIA's CUDA (Computer Unified Device Architecture) programming model was the first that enabled high level programming of GPUs; more recently the OpenCL language (which we will also cover in this book) is set to become the de facto standard for data-parallel computing in general and for programming GPUs in particular (NVIDIA itself adopted it as a layer above CUDA). On hardware side, the most radical new chip design to date by NVIDIA is the Fermi family of chips.

The Fermi architecture crams an impressive 3 billion transistors into a system with 512 CUDA cores running at 700 MHz each. A CUDA core has a pipelined 32 bit integer arithmetic logic unit (ALU) and floating point unit (FPU) with fused multiply-add support, being capable to execute an integer or floating point operation in every cycle. The CUDA cores are grouped into 16 *streaming multi-processors,* each featuring 32 CUDA cores and 16 load/store units allowing memory access operations for 16 threads per each clock cycle. There are also four special function units (SFU) that can calculate values of functions such as sin or cosine. For a schematic view of the SM architecture, see Fig. 1.3

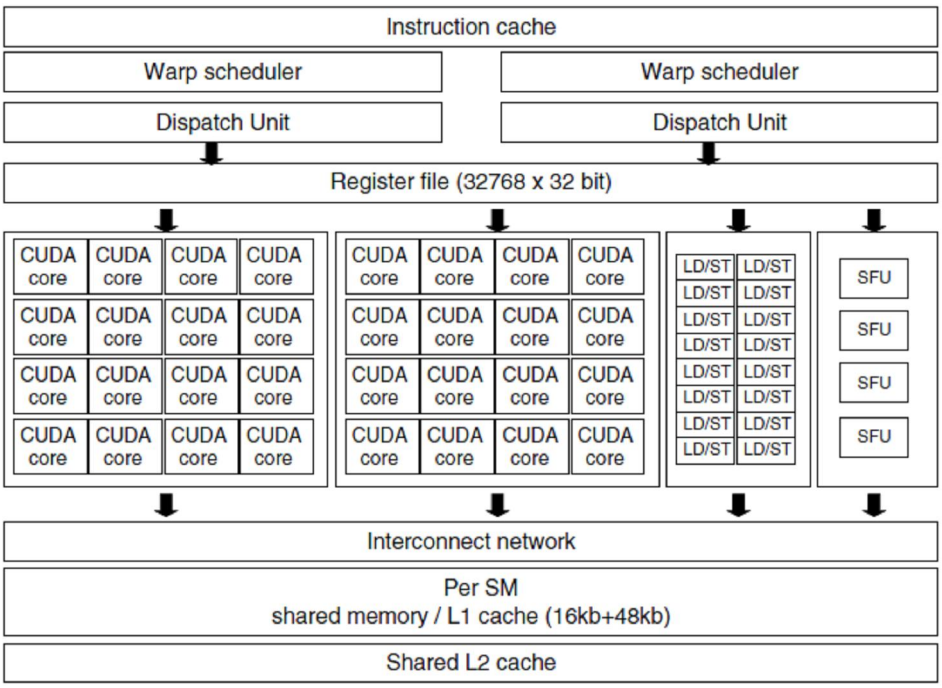


Fig. 1.3 NVIDIA streaming multiprocessor architecture [2]

GPUs have clearly outgrown their roots in graphics processing and are now targeting the broader field of massively data-parallel applications with significant amount of scientific and floating point calculations. While a particular programming model must be followed, the programmability of these chips has increased to a level that is on par with regular CPUs, turning these chips into a compelling choice for general purpose utilization.

1.1.3. PicoChip Architecture

The DSP architecture developed by picoChip is a prime example of successfully addressing a particular application domain with a massively multi-core processor.

The basic architecture of a picoChip DSP consists of a large number of various processing units (more than 250 in the largest configuration), connected in a mesh network (called the *picoArray*) using an interconnect resembling Tilera's iMesh technology. However, the communication is based on time division multiplexing mechanisms with the schedule decided deterministically at compile time, thus eliminating the need for execution time scheduling and arbitration. While this may be a big advantage in some cases, it will also limit the usability of the technology for more dynamic use-cases.

The processing elements in the *picoArray* can be

- *proprietary DSP cores:* 16 bit Harvard architecture processors with three-way very long instruction words, running at 160 MHz
- *hardware accelerators* for functions such as Fast Fourier Transformation (FFT), cryptography or various wireless networking algorithms that are unlikely to change and hence can be hard-coded into hardware
- *ARM core* for more complex control functions (such as operation and maintenance interface)

The DSP cores come in three variants, differentiated by target role and, as a consequence, available memory:

- *standard:* used for data-path processing with very low amount (< 1 kb) own memory
- *memory:* used for local control and buffering with approx. 8 kb of memory
- *control:* global control functions, with 64 kb of memory

Chips based on the *picoArray* architecture—due to the low amount of memory and very simple cores—have very low power consumption. For example, a model with 250 DSP cores running at 160 MHz, one ARM core at 280 MHz and several accelerators consumes just around 1 W, while capable of executing 120 billion basic arithmetic operations per second.

The picoChip architecture is a prime example of massively multi-core processors found in the embedded domain, especially telecommunication products (wireless infrastructure nodes and routers). These nodes usually perform the same, relatively simple sequence of operations on a very large amount of entities (packets, phone calls or data connections), hence such architectures are the best match in terms of balance between programmability and efficiency.

1.1.4. Many Integrated Core Architecture

In 2010 Intel announced their first commercial many-core chip code-named Knights Corner that will be manufactured in 22 nm and will integrate more than 50 x 86 cores. While not publicly stated, it's most likely the continuation of the Larrabee program that aimed at developing a GPU-like device.

There were few details available publicly about this device that was planned to be released in 2011. Based on an Intel presentation, it is based on "vector Intel Architecture cores", which indicate an emphasis on data parallel processing. The cores support hardware threading (at least 4 threads per core) and share a tiled, coherent on-chip cache. The chips likely include hardware accelerators, called *fixed function logic* in Intel's presentation. A schematic view of the architecture is shown in Fig. 1.4.

On June 18, 2012, Intel announced that Xeon Phi will be the brand name used for all products based on their Many Integrated Core architecture.

On November 12, 2012, Intel announced two Xeon Phi coprocessor families which are the Xeon Phi 3100 and the Xeon Phi 5110P.

The Intel Xeon Phi coprocessor 3100 family will provide great value for those seeking to run compute-bound workloads such as life science applications and financial simulations. The Intel Xeon Phi 3100 family will offer more than 1000 Gigaflops (1 TFlops) double-precision performance, support for up to 6GB memory at 240GB/sec bandwidth, and a series of reliability features including memory error correction codes (ECC). The family will operate within a 300W thermal design point (TDP) envelope.

The Intel Xeon Phi coprocessor 5110P provides additional performance at a lower power envelope. It reaches 1,011 Gigaflops (1.01 TFlops) double-precision performance, and supports 8GB of GDDR5 memory at a higher 320 GB/sec memory bandwidth. With 225 watts TDP, the passively cooled Intel Xeon Phi coprocessor 5110P delivers power efficiency that is ideal for dense computing environments, and is aimed at capacity-bound workloads such as digital content creation and energy research. [6]

The Xeon Phi uses the 22 nm process size.

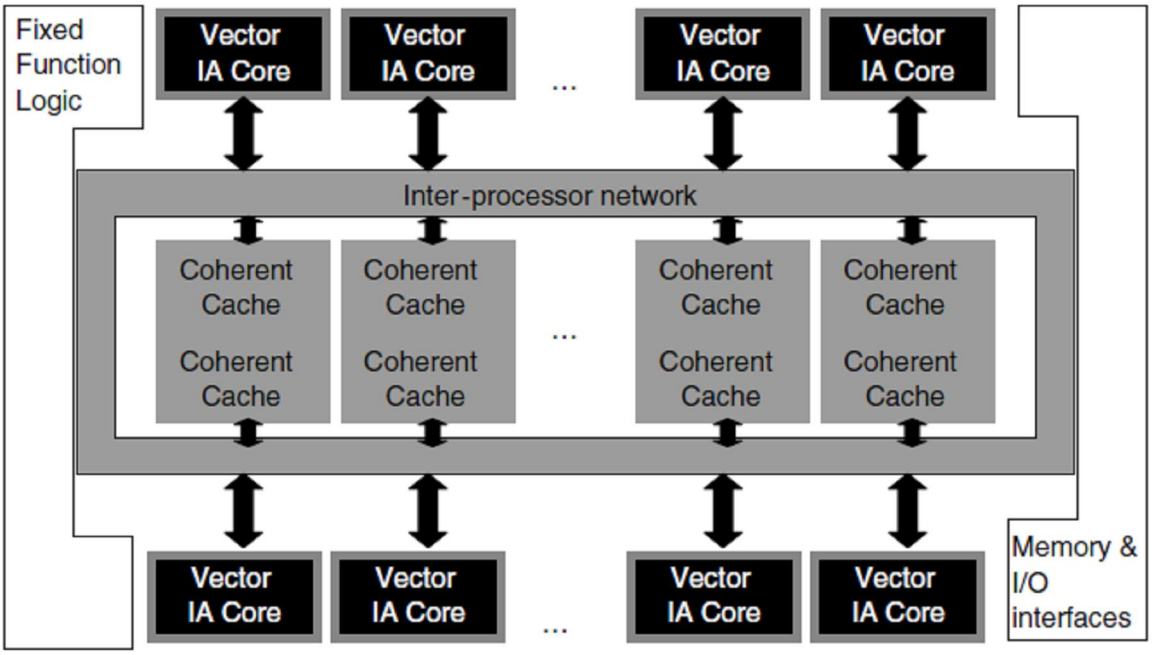


Fig. 1.4 Schematic architecture of Intel's Knights Corner chip design [2]

1.2. Why Many-Core

The observation that triggered the push for chips with large numbers of simpler cores was most concisely formulated in Ref. [4] as the KILL rule (stands for Kill if Less than Linear). The Kill Rule is a simple scientific way of making the tradeoff between increasing the number of cores or increasing the core size. The Kill Rule states that a resource in a core must be increased in area only if the core's performance improvement is at least proportional to the core's area increase. Put in another way, increase resource size only if for every 1% increase in core area there is at least a 1% increase in core performance. The rationale for the Kill Rule is that a multicore's performance can always be increased proportionally by adding more cores (assuming application parallelism exists). Thus, increase in a core's size can only be justified if it results in a proportional increase in performance. It is one of the fundamental principles underpinning the design of chips such as Tilera's family of Tile-architecture based, massively multi-core chips.

On manufacturing technology level, major chip manufacturers estimate that we will be able to continue with current CMOS technologies down to approximately the 6 nm manufacturing process (as of 2012, the most advanced technology node is the 22 nm node). However at that level at least two major challenges will emerge:

- As the size of transistors will be measured in just a few tens of atoms at most, quantum effects will have to be taken into account and consequently we will seean increased *unreliability of the hardware*, with components failing more often and—more importantly—intermittently
- There will be so many transistors on the chip that it will be, power wise, impossible to switch all of these at the same time; this phenomenon—called the *dark silicon* problem—will have a significant impact on how we will build future processors.

The unreliability of future hardware will likely lead to the implementation of redundant execution mechanisms. Multiple cores will perform the same computation, in order to increase the probability that at least one will succeed; in some cases a voting

scheme on the result (verifying if all the computations yielded the same result) may be used to guarantee correctness of the calculation. Such mechanisms will likely be invisible to the software, but will impact the complexity of the design of logical processor cores.

The dark silicon problem is trickier to address. By lowering the frequency at which chips operate, we can push the limit further, but eventually it will become an issue, no matter how low we go with the frequency.

For the reasons mentioned above, there is a shift towards *simple, low frequency, low complexity* cores, coupled with an increase of the core count to the level of several hundreds within the next years.

1.3. Scalability Issues for Many-core Processors

A scalability bottleneck relates to the design of cache coherency protocols. Synchronizing access to the same memory area from large number of cores will increase the complexity of coherency design and will lead to increasing delay and latency in accessing frequently modified memory areas. In our view, it will be an uphill battle to maintain a coherent view across hundreds, let alone a thousand cores; even if we will be able to do this, it will be hard to justify the cost associated with it.

Memory bandwidth will be another scalability bottleneck. The levelling out of the core frequency will lead to reduced latency, but the increase in the number of cores will multiply the amount of data required by the cores and thus the aggregate memory bandwidth that future chips will require. If we will indeed see the continuation of Moore's law, translated into an ever-increasing number of cores—perhaps following the same trend of doubling core-count every two years—a similar trend would need to be followed by memory bandwidth, something the industry failed to deliver in the past.

From a programming point of view, current engineering practices and software architectures for embedded systems are not ready to adopt a massively parallel hardware platform just yet. Current system designs heavily rely on a single-processor model with a serialized execution of tasks. The execution order of tasks is often expressed as task priorities. This approach helps to ensure proper timing by giving exclusive access for a running task to the underlying hardware. If interrupts are disabled, tasks may run to completion without interference. Dynamic scheduling algorithms at run-time focus on assigning priorities to tasks, so that the most important task is selected for execution by a dispatcher component. Albeit the simplicity of the execution model, programmers still struggle to implement complex applications. This is especially the case when quality requirements, such as real-time behaviour or functional safety, have to be satisfied. Many-core processors raise the level of system complexity significantly by offering a parallel execution at run-time. [4]

Last, but not least, the speedup of parallel execution is governed by Amdahl's law. [11] It distinguishes between code to be executed in parallel and code to be executed sequentially. In Amdahl's model, the total execution time is the sum of the execution time of the parallel part and of the sequential part. By adding more processors, the execution time of the parallel part can be reduced, because the work can be split among several processors. This reduces the total execution time and increases the performance. According to Amdahl's law, an application containing a parallel code portion of 95% and a sequential code portion of 5%, the potential speedup on a processor with 8000 cores in comparison to a single-core processor with the same clock speed is at most 20. It becomes apparent that the performance of the sequential part dominates the speedup.

1.4. Convergence of Embedded systems with HPC

High performance computing (HPC) refers to running large, compute-intensive applications to solve complex numerical algorithms typically used in areas such as image processing and simulation. Some HPC applications include computational fluid dynamics, weather modelling, and circuit simulation. Trying to get a computer to emulate something in the real world requires a tremendous amount of computational power. Historically, if you were to actually see a supercomputer, you would find it housed in a large facility with rows and racks of blinking lights, heavy air conditioning, and maybe even a water-based cooling system. Today, with the introduction of more compact and more powerful embedded processors, embedded systems are becoming HPC capable.

Traditionally, embedded systems are associated with microcontrollers. These systems conduct basic functions such as blinking lights, turning on relays, checking to make sure the thermostat is on, or turning on the air conditioner. But with the recent advances in computing platforms, it's now possible to run far more sophisticated software, enabling exciting new applications. For the U.S. military, HPC enables airborne vehicles to navigate without pilots. In the automotive sector, HPC serves as a key building block in the development of anti-collision systems. These systems can actually look out over the road, collect data on what's going on around the vehicle, disengage the cruise control, or sound an alert to the driver, depending on the severity of the event.

A sparkling example of the evolution of embedded systems in the last decade comes from mobile communications. The first commercial mobile phones were just means to ease everyday communication. Nowadays, mobile phones accommodate up to four processing cores, high definition cameras, GPS receivers, GPUs and a plethora of other hardware modules, all available to consumers at an affordable cost. Today’s phones exhibit greater computing capacity than yesterday’s high-performance systems.

Even FPGA companies like Xilinx and Altera are getting into the HPC act by introducing new devices that combine FPGA fabrics and multicore CPUs. Devices such as the new Zynq-7000 from Xilinx offer a dual-core ARM A9 processor with the Neon vector accelerator, plus an extensive FPGA fabric providing developers the ability to roll their own custom hardware accelerator for specialized HPC applications. Traditional HPC engines, like DSP processors, are not going away. A quick a look at TI's latest offerings reveal the Integra line, which features a C6x DSP core and an ARM Cortex A8 CPU on the same chip.

This sounds like a wonderful new world; a plethora of powerful computing devices you can just hook together to meet your HPC needs. The problem is that programming some of these specialized devices like DSPs or GP-GPUs for maximum performance is not at all trivial. Things are further complicated by the fact that modern HPC platforms are likely to contain a heterogeneous mixture of processing cores, such as an FPGA, a GP-GPU, or a DSP combined with a single or multicore CPU. Programming a heterogeneous compute platform becomes even more difficult. As a result of this complexity, typical embedded software development costs are exceeding well over 50% of the entire system cost.

In addition to achieving maximum compute performance, the other key problem area that developers face is the idea of portability. If project leaders are going to have their software teams develop applications that are capable of image recognition, they don't want to reinvent that activity every time a new platform is announced. Yet traditionally, for high-performance computing, to get things to run very, very fast--especially on small embedded systems--the order of the day was hand coding in assembly language. This was laboriously done by coding down at the bit level to make sure the code ran as fast as possible. Needless to say, that kind of labour, especially on more sophisticated applications, is prohibitive. It is both very expensive and certainly not very portable. If somebody is going to invest in this

space by building wonderful new applications, they will want their application to be portable and easily map to any new hardware. This is especially true nowadays when it seems that every six months a new type of device is announced. So portability means at least future-proofing software development against the roadmap of the hardware vendor. Because software costs are rising, anything that can be done to help software developers be more productive and get them out of the business of hand tuning low-level routines will reduce development costs and allow more time to concentrate on the high-level features.

Two key components are required to make this happen. One is a good interface between the hardware and software development; this often takes the form of hardware-specific, low-level libraries. These low-level libraries have to be tuned specifically for maximum performance on a given processor. They're basically the functions that enable a particular piece of silicon to execute a simple instruction like "add these two numbers" and do it as quickly as possible, leveraging powerful accelerators that might be available for that given data type.

But that's still not enough. At some point, embedded systems developers have to compose an image recognition system out of a series of add, multiply, subtract, and divide operations while simultaneously handling all the data details such as various data types, storage location, storage arrangement in memory (called stride), and memory copy operations to ensure the right data is available where and when it's needed. Ideally, you would have a high-level portable library that provides high-level function abstraction and data encapsulation that sits atop those low-level hardware-specific libraries. Such an approach basically allows you to write algorithms in software very much as you would in a specification document or on a chalkboard.

To sum up, convergence, both in hardware and software platforms, is rampant throughout the industry with desktop processors and embedded processors merging and embedded system developers having to deal with portability and high performance computing.

1.5. Big data - MapReduce

1.5.1. Big Data

We live in the data age. Welcome to the "digital universe" — a measure of all the digital data created, replicated, and consumed in a single year. It's also a projection of the size of that universe to the end of the decade. The digital universe is made up of images and videos on mobile phones uploaded to YouTube, digital movies populating the pixels of our high-definition TVs, banking data swiped in an ATM, security footage at airports and major events such as the Olympic Games, subatomic collisions recorded by the Large Hadron Collider at CERN, transponders recording highway tolls, voice calls zipping through digital phone lines, and texting as a widespread means of communications.

Consider the following:

- The New York Stock Exchange generates about one terabyte of new trade data per day.
- Facebook hosts approximately 10 billion photos, taking up one petabyte of storage.
- Ancestry.com, the genealogy site, stores around 2.5 petabytes of data.
- The Internet Archive stores around 2 petabytes of data, and is growing at a rate of 20 terabytes per month.

- The Large Hadron Collider near Geneva, Switzerland, will produce about 15 petabytes of data per year.

It's not easy to measure the total volume of data stored electronically, but an IDC [7] estimate put the size of the "digital universe" at 0.18 zettabytes in 2006, A zettabyte is 10^{21} bytes, or equivalently one thousand exabytes, one million petabytes, or one billion terabytes. That's roughly the same order of magnitude as one disk drive for every person in the world. IDC also estimates that from 2005 to 2020, the digital universe will grow by a factor of 300, from 130 exabytes to 40,000 exabytes, or 40 trillion gigabytes (more than 5,200 gigabytes for every man, woman, and child in 2020) (Fig. 1.5). From now until 2020, the digital universe will about double every two years.

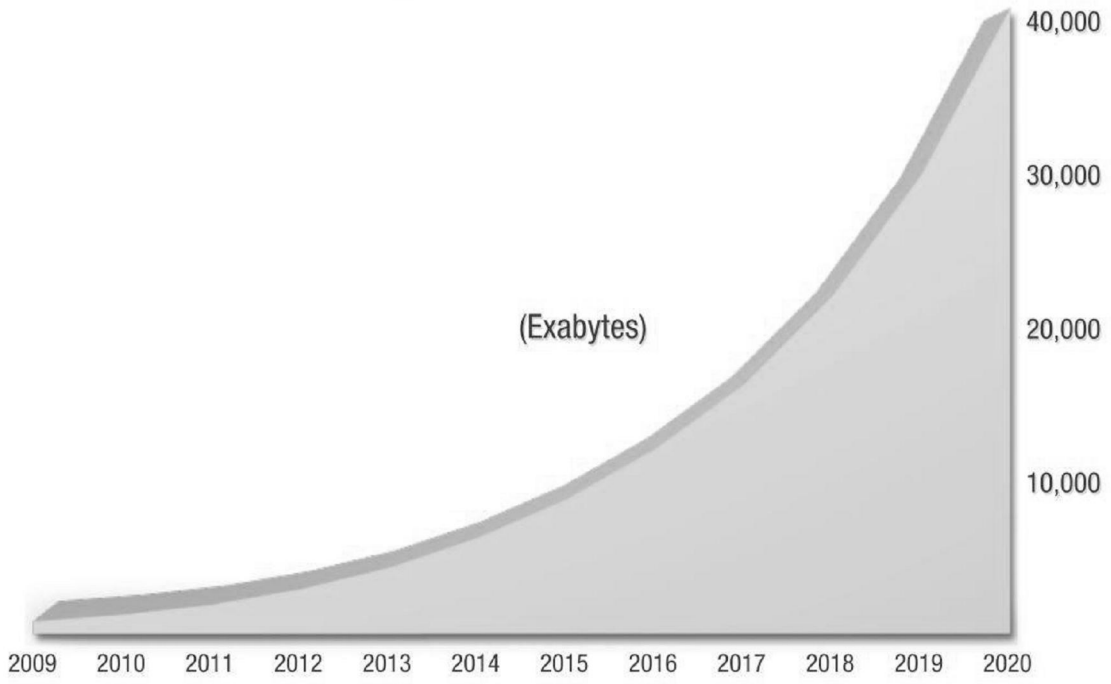


Fig. 1.5 The Digital Universe [7]

The problem is simple: while the storage capacities of hard drives have increased massively over the years, access speeds—the rate at which data can be read from drives—have not kept up. One typical drive from 1990 could store 1370 MB of data and had a transfer speed of 4.4 MB/s, so you could read all the data from a full drive in around five minutes. Almost 20 years later one terabyte drives are the norm, but the transfer speed is around 100 MB/s, so it takes more than two and a half hours to read all the data off the disk.

This is a long time to read all data on a single drive—and writing is even slower. The obvious way to reduce the time is to read from multiple disks at once. Imagine if we had 100 drives, each holding one hundredth of the data. Working in parallel, we could read the data in less than two minutes.

Only using one hundredth of a disk may seem wasteful. But we can store one hundred datasets, each of which is one terabyte, and provide shared access to them. We can imagine that the users of such a system would be happy to share access in return for shorter analysis times, and, statistically, that their analysis jobs would be likely to be spread over time, so they wouldn't interfere with each other too much.

Consequently, this immense growth in data size combined with the emergence of distributed computing power, in form of either multi-core processors or computer clusters, created the need for programming models capable of processing large data sets with a

parallel, distributed algorithm. One such model for processing parallelizable problems across huge datasets using a large number of processing nodes is *MapReduce*.

1.5.2 MapReduce Origins

MapReduce was introduced in 2004 by Google [8]. MapReduce has its roots in functional programming, which is exemplified in languages such as Lisp and ML. A key feature of functional languages is the concept of higher order functions, or functions that can accept other functions as arguments. Two common built-in higher order functions are map and fold, illustrated in Figure 1.6. Given a list, map takes as an argument a function f (that takes a single argument) and applies it to all elements in a list (the top part of the diagram). Given a list, fold takes as arguments a function g (that takes two arguments) and an initial value: g is first applied to the initial value and the first item in the list, the result of which is stored in an intermediate variable. This intermediate variable and the next item in the list serve as the arguments to a second application of g, the results of which are stored in the intermediate variable. This process repeats until all items in the list have been consumed; fold then returns the final value of the intermediate variable. Typically, map and fold are used in combination.

For example, to compute the sum of squares of a list of integers, one could map a function that squares its argument over the input list, and then fold the resulting list with the addition function using an initial value of zero.

We can view map as a concise way to represent the transformation of a dataset (as defined by the function f). In the same vein, we can view fold as an aggregation operation, as defined by the function g. One immediate observation is that the application of f to each item in a list (or more generally, to elements in a large dataset) can be parallelized in a straightforward manner, since each functional application happens in isolation. In a cluster, these operations can be distributed across many different machines. The fold operation, on the other hand, has more restrictions on data locality - elements in the list must be "brought together" before the function g can be applied. However, many real-world applications do not require g to be applied to all elements of the list. To the extent that elements in the list can be divided into groups, the fold aggregations can also proceed in parallel. Furthermore, for operations that are commutative and associative, significant efficiencies can be gained in the fold operation through local aggregation and appropriate reordering.

In a nutshell, we have described MapReduce. The map phase in MapReduce roughly corresponds to the map operation in functional programming, whereas the reduce phase in MapReduce roughly corresponds to the fold operation in functional programming.

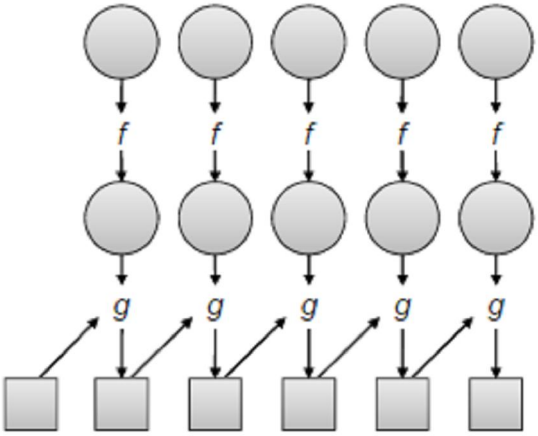


Fig. 1.6 Illustration of map and fold [10]

1.5.2 MapReduce Programming Model

The computation takes a set of *input* key/value pairs, and produces a set of *output* key/value pairs. The user of the MapReduce library expresses the computation as two functions: *Map* and *Reduce*.

Map, written by the user, takes an input pair and produces a set of *intermediate* key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key I and passes them to the *Reduce* function.

The *Reduce* function, also written by the user, accepts an intermediate key I and a set of values for that key. It merges together these values to form a possibly smaller set of values. Typically just zero or one output value is produced per *Reduce* invocation. The intermediate values are supplied to the user's reduce function via an iterator. This allows us to handle lists of values that are too large to fit in memory.

As an example, consider the problem of counting the number of occurrences of each word in a large collection of documents. The user would write code similar to the following pseudo-code:

```
map(String key, String value):
        // key: document name
        // value: document contents
        for each word w in value:
                EmitIntermediate(w, "1");

reduce(String key, Iterator values):
        // key: a word
        // values: a list of counts
        int result = 0;
        for each v in values:
                result += ParseInt(v);
        Emit(AsString(result));
```

The map function emits each word plus an associated count of occurrences (just `1' in this simple example). The reduce function sums together all counts emitted for a particular word.

Key-value pairs form the basic data structure in MapReduce. Keys and values may be primitives such as integers, floating point values, strings, and raw bytes, or they may be arbitrarily complex structures (lists, tuples, associative arrays, etc.). Programmers typically need to define their own custom data types.

Part of the design of MapReduce algorithms involves imposing the key-value structure on arbitrary datasets. For a collection of web pages, keys may be URLs and values may be the actual HTML content. For a graph, keys may represent node ids and values may contain the adjacency lists of those nodes.

One of the most important idea behind MapReduce is separating the what of distributed processing from the how. A MapReduce program, referred to as a job, consists of code for mappers and reducers packaged together with configuration parameters (such as where the input lies and where the output should be stored). The developer submits the job to the submission node of a cluster and execution framework takes care of everything else: it transparently handles all other aspects of distributed code execution, on clusters ranging from a single node to a few thousand nodes. Specific responsibilities include:

Scheduling: Each MapReduce job is divided into smaller units called For example, a map task may be responsible for processing a certain block of input key-value pairs; similarly, a reduce task may handle a portion of the intermediate key space. It is not uncommon for MapReduce jobs to have thousands of individual tasks that need to be assigned to nodes in the cluster. In large jobs, the total number of tasks may exceed the number of tasks that can be run on the cluster concurrently, making it necessary for the scheduler to maintain some sort of a task queue and to track the progress of running tasks so that waiting tasks can be assigned to nodes as they become available. Another aspect of scheduling involves coordination among tasks belonging to different jobs.

Speculative execution is an optimization that is implemented by both Hadoop and Google's MapReduce implementation. Due to the barrier between the map and reduce tasks, the map phase of a job is only as fast as the slowest map task. Similarly, the completion time of a job is bounded by the running time of the slowest reduce task. As a result, the speed of a MapReduce job is sensitive to what are known as stragglers, or tasks that take an usually long time to complete. One cause of stragglers is flaky hardware: for example, a machine that is suffering from recoverable errors may become significantly slower. With speculative execution, an identical copy of the same task is executed on a different machine, and the framework simply uses the result of the first task attempt to finish.

Data/code co-location: The phrase data distribution is misleading, since one of the key ideas behind MapReduce is to move the code, not the data. However, the more general point remains - in order for computation to occur, we need to somehow feed data to the code. In MapReduce, this issue is inexplicably intertwined with scheduling and relies heavily on the design of the underlying distributed file system. To achieve data locality, the scheduler starts tasks on the node that holds a particular block of data (i.e., on its local drive) needed by the task. This has the effect of moving code to the data. If this is not possible (e.g., a node is already running too many tasks), new tasks will be started elsewhere, and the necessary data will be streamed over the network. An important optimization here is to prefer nodes that are on the same rack in the datacenter as the node holding the relevant data block, since inter-rack bandwidth is significantly less than intra-rack bandwidth.

Synchronization: In general, synchronization refers to the mechanisms by which multiple concurrently running processes “join up”, for example, to share intermediate results

or otherwise exchange state information. In MapReduce, synchronization is accomplished by a barrier between the map and reduce phases of processing. Intermediate key-value pairs must be grouped by key, which is accomplished by a large distributed sort involving all the nodes that executed map tasks and all the nodes that will execute reduce tasks. This necessarily involves copying intermediate data over the network, and therefore the process is commonly known as "shuffle and sort". A MapReduce job with m mappers and r reducers involves up to $m \times r$ distinct copy operations, since each mapper may have intermediate output going to every reducer.

Note that the reduce computation cannot start until all the mappers have finished emitting key-value pairs and all intermediate key-value pairs have been shuffled and sorted, since the execution framework cannot otherwise guarantee that all values associated with the same key have been gathered. This is an important departure from functional programming: in a fold operation, the aggregation function *g* is a function of the intermediate value and the next item in the list, which means that values can be lazily generated and aggregation can begin as soon as values are available. In contrast, the reducer in MapReduce receives all values associated with the same key at once.

Error and fault handling: The MapReduce execution framework must accomplish all the tasks above in an environment where errors and faults are the norm, not the exception. Since MapReduce was explicitly designed around low-end commodity servers, the runtime must be especially resilient. In large clusters, disk failures are common and RAM experiences more errors than one might expect. Datacenters suffer from both planned outages (e.g., system maintenance and hardware upgrades) and unexpected outages (e.g., power failure, connectivity loss, etc.).

And that's just hardware. No software is bug free; exceptions must be appropriately trapped, logged, and recovered from. Large-data problems have a penchant for uncovering obscure corner cases in code that is otherwise thought to be bug-free. Furthermore, any sufficiently large dataset will contain corrupted data or records that are mangled beyond a programmer's imagination, resulting in errors that one would never think to check for or trap. The MapReduce execution framework must thrive in this hostile environment.

1.5.3 Why MapReduce

Move processing to the data: In traditional high-performance computing (HPC) applications (e.g., for climate or nuclear simulations), it is commonplace for a supercomputer to have "processing nodes" and "storage nodes" linked together by a high-capacity interconnect. Many data-intensive workloads are not very processor-demanding, which means that the separation of compute and storage creates a bottleneck in the network. As an alternative to moving data around, it is more efficient to move the processing around. That is, MapReduce assumes an architecture where processors and storage (disk) are co-located. In such a setup, we can take advantage of data locality by running code on the processor directly attached to the block of data we need. The distributed file system is responsible for managing the data over which MapReduce operates.

Process data sequentially and avoid random access: Data-intensive processing by definition means that the relevant datasets are too large to fit in memory and must be held on disk. Seek times for random disk access are fundamentally limited by the mechanical nature of the devices: read heads can only move so fast and platters can only spin so rapidly. As a result, it is desirable to avoid random data access, and instead organize computations so that data is processed sequentially. A simple scenario poignantly illustrates the large performance gap between sequential operations and random seeks: assume a 1 terabyte

database containing 10^{10} 100-byte records. Given reasonable assumptions about disk latency and throughput, a back-of-the-envelop calculation will show that updating 1% of the records (by accessing and then mutating each record) will take about a month on a single machine. On the other hand, if one simply reads the entire database and rewrites all the records (mutating those that need updating), the process would finish in under a work day on a single machine. Sequential data access is, literally, orders of magnitude faster than random data access.

The development of solid-state drives is unlikely the change this balance for at least two reasons. First, the cost differential between traditional magnetic disks and solid-state disks remains substantial: large-data will for the most part remain on mechanical drives, at least in the near future. Second, although solid-state disks have substantially faster seek times, order-of-magnitude differences in performance between sequential and random access still remain.

MapReduce is primarily designed for batch processing over large datasets. To the extent possible, all computations are organized into long streaming operations that take advantage of the aggregate bandwidth of many disks in a cluster. Many aspects of MapReduce's design explicitly trade latency for throughput.

Hide system-level details from the application developer: According to many guides on the practice of software engineering written by experienced industry professionals, one of the key reasons why writing code is difficult is because the programmer must simultaneously keep track of many details in short term memory, ranging from the mundane (e.g., variable names) to the sophisticated (e.g., a corner case of an algorithm that requires special treatment). This imposes a high cognitive load and requires intense concentration, which leads to a number of recommendations about a programmer's environment (e.g., quiet office, comfortable furniture, large monitors, etc.). The challenges in writing distributed software are greatly compounded; the programmer must manage details across several threads, processes, or machines. Of course, the biggest headache in distributed programming is that code runs concurrently in unpredictable orders, accessing data in unpredictable patterns. This gives rise to race conditions, deadlocks, and other well-known problems. Programmers are taught to use low-level devices such as mutexes and to apply high-level "design patterns" such as producer - consumer queues to tackle these challenges, but the truth remains: concurrent programs are notoriously difficult to reason about and even harder to debug.

MapReduce addresses the challenges of distributed programming by providing an abstraction that isolates the developer from system-level details (e.g., locking of data structures, data starvation issues in the processing pipeline, etc.). The programming model specifies simple and well-defined interfaces between a small number of components, and therefore is easy for the programmer to reason about. MapReduce maintains a separation of *what* computations are to be performed and *how* those computations are actually carried out on a cluster of machines. The first is under the control of the programmer, while the second is exclusively the responsibility of the execution framework or "runtime". The advantage is that the execution framework only needs to be designed once and verified for correctness; thereafter, as long as the developer expresses computations in the programming model, code is guaranteed to behave as expected. The upshot is that the developer is freed from having to worry about system-level details (e.g., no more debugging race conditions and addressing lock contention) and can instead focus on algorithm or application design.

Chapter 2: State of Art

2.1. MapReduce Frameworks

2.1.1 Google's Implementation

Google's implementation [8] is targeted to the computing environment in wide use at Google: large clusters of commodity PCs connected together with switched Ethernet. In their environment:

(1) Machines are typically dual-processor x86 processors running Linux, with 2-4 GB of memory per machine.
(2) Commodity networking hardware is used, typically either 100 megabits/second or 1 gigabit/second at the machine level, but averaging considerably less in overall bisection bandwidth.
(3) A cluster consists of hundreds or thousands of machines, and therefore machine failures are common.
(4) Storage is provided by inexpensive IDE disks attached directly to individual machines. A distributed file system (Google file system – GFS) developed in-house is used to manage the data stored on these disks. The file system uses replication to provide availability and reliability on top of unreliable hardware.
(5) Users submit jobs to a scheduling system. Each job consists of a set of tasks, and is mapped by the scheduler to a set of available machines within a cluster.

The programming language used in this implementation is C++.

2.1.1.1. Execution Overview

The *Map* invocations are distributed across multiple machines by automatically partitioning the input data into a set of M *splits*. The input splits can be processed in parallel by different machines. *Reduce* invocations are distributed by partitioning the intermediate key space into *R* pieces using a partitioning function (e.g., *hash(key) mod R*). The number of partitions (*R*) and the partitioning function are specified by the user.

Figure 2.1 shows the overall flow of a MapReduce operation in Google's implementation. When the user program calls the MapReduce function, the following sequence of actions occurs (the numbered labels in Figure 2.1 correspond to the numbers in the list below):

1. The MapReduce library in the user program first splits the input files into M pieces of typically 16 megabytes to 64 megabytes (MB) per piece (controllable by the user via an optional parameter). It then starts up many copies of the program on a cluster of machines.
2. One of the copies of the program is special, the master. The rest are workers that are assigned work by the master. There are *M* map tasks and *R* reduce tasks to assign. The master picks idle workers and assigns each one a map task or a reduce task.
3. A worker who is assigned a map task reads the contents of the corresponding input split. It parses key/value pairs out of the input data and passes each pair to the user-defined *Map* function. The intermediate key/value pairs produced by the *Map* function are buffered in memory.

4. Periodically, the buffered pairs are written to local disk, partitioned into R regions by the partitioning function. The locations of these buffered pairs on the local disk are passed back to the master, who is responsible for forwarding these locations to the reduce workers.
5. When a reduce worker is notified by the master about these locations, it uses remote procedure calls to read the buffered data from the local disks of the map workers. When a reduce worker has read all intermediate data, it sorts it by the intermediate keys so that all occurrences of the same key are grouped together. The sorting is needed because typically many different keys map to the same reduce task. If the amount of intermediate data is too large to fit in memory, an external sort is used.
6. The reduce worker iterates over the sorted intermediate data and for each unique intermediate key encountered, it passes the key and the corresponding set of intermediate values to the user's Reduce function. The output of the Reduce function is appended to a final output file for this reduce partition.
7. When all map tasks and reduce tasks have been completed, the master wakes up the user program. At this point, the MapReduce call in the user program returns back to the user code.

After successful completion, the output of the MapReduce execution is available in the *R* output files (one per reduce task, with file names as specified by the user). Typically, users do not need to combine these R output files into one file, they often pass these files as input to another MapReduce call, or use them from another distributed application that is able to deal with input that is partitioned into multiple files.

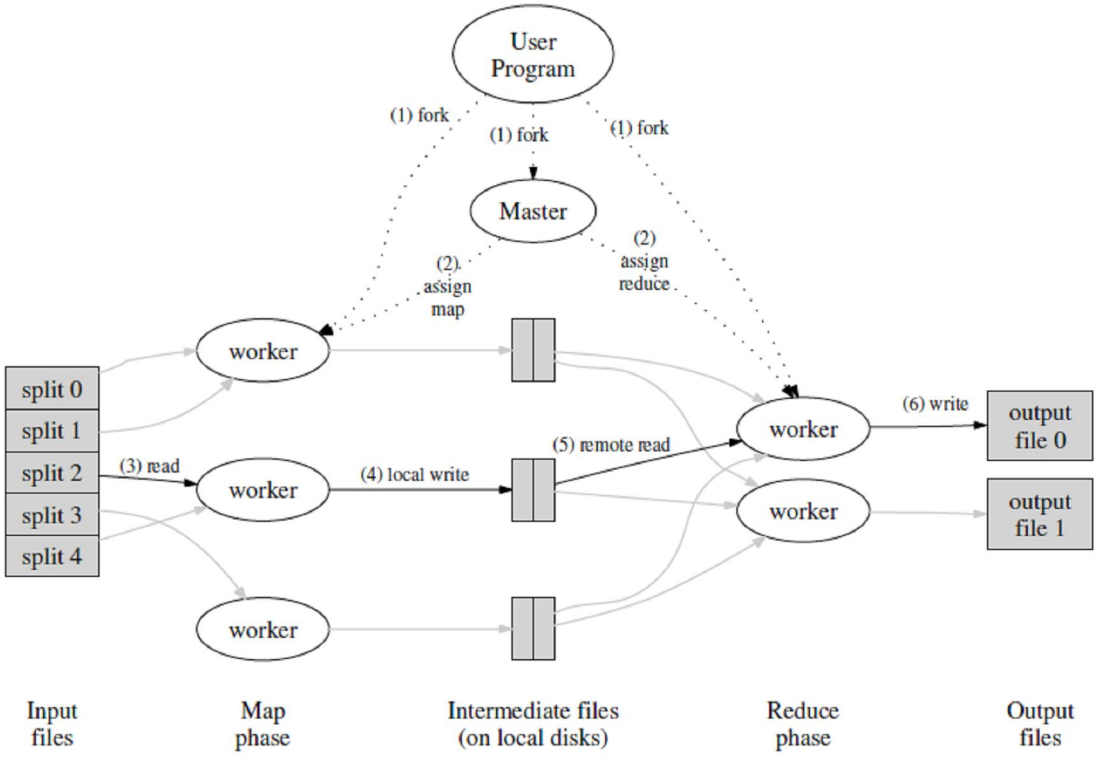


Fig. 2.1 Execution Overview [8]

2.1.1.2. Fault Tolerance

Since the MapReduce library is designed to help process very large amounts of data using hundreds or thousands of machines, the library must tolerate machine failures gracefully.

Worker Failure: The master pings every worker periodically. If no response is received from a worker in a certain amount of time, the master marks the worker as failed. Any map tasks completed by the worker are reset back to their initial *idle* state, and therefore become eligible for scheduling on other workers. Similarly, any map task or reduce task in progress on a failed worker is also reset to *idle* and becomes eligible for rescheduling.

Completed map tasks are re-executed on a failure because their output is stored on the local disk(s) of the failed machine and is therefore inaccessible. Completed reduce tasks do not need to be re-executed since their output is stored in a global file system.

When a map task is executed first by worker A and then later executed by worker B (because A failed), all workers executing reduce tasks are notified of the re-execution. Any reduce task that has not already read the data from worker A will read the data from worker B.

MapReduce is resilient to large-scale worker failures. For example, during one MapReduce operation, network maintenance on a running cluster was causing groups of 80 machines at a time to become unreachable for several minutes. The MapReduce master simply re-executed the work done by the unreachable worker machines, and continued to make forward progress, eventually completing the MapReduce operation.

Master Failure: It is easy to make the master write periodic checkpoints of the master data structures, which store the state (*idle*, *in-progress*, or *completed*), and the identity of the worker machine (for non-idle tasks) for each map and reduce task. If the master task dies, a new copy can be started from the last checkpointed state. However, given that there is only a single master, its failure is unlikely.

2.1.1.3. Backup Tasks

One of the common causes that lengthen the total time taken for a MapReduce operation is a “straggler”: a machine that takes an unusually long time to complete one of the last few map or reduce tasks in the computation. Stragglers can arise for a whole host of reasons. For example, a machine with a bad disk may experience frequent correctable errors that slow its read performance from 30 MB/s to 1 MB/s. The cluster scheduling system may have scheduled other tasks on the machine, causing it to execute the MapReduce code more slowly due to competition for CPU, memory, local disk, or network bandwidth. A recent problem was a bug in machine initialization code that caused processor caches to be disabled: computations on affected machines slowed down by over a factor of one hundred.

A general mechanism to alleviate the problem of stragglers was created. When a MapReduce operation is close to completion, the master schedules backup executions of the remaining *in-progress* tasks. The task is marked as completed whenever either the primary or the backup execution completes. This mechanism was tuned so that it typically increases the computational resources used by the operation by no more than a few percent. This significantly reduces the time to complete large MapReduce operations. As an example, the sort program described bellow takes 44% longer to complete when the backup task mechanism is disabled.

2.1.1.4. Locality

Network bandwidth is a relatively scarce resource in google's computing environment. Network bandwidth is conserved by taking advantage of the fact that the input data (managed by GFS) is stored on the local disks of the machines that make up the cluster. GFS divides each file into 64 MB blocks, and stores several copies of each block (typically 3 copies) on different machines. The MapReduce master takes the location information of the input files into account and attempts to schedule a map task on a machine that contains a replica of the corresponding input data. Failing that, it attempts to schedule a map task near a replica of that task's input data (e.g., on a worker machine that is on the same network switch as the machine containing the data). When running large MapReduce operations on a significant fraction of the workers in a cluster, most input data is read locally and consumes no network bandwidth.

2.1.1.5. Refinements

Although the basic functionality provided by simply writing *Map* and *Reduce* functions is sufficient for most needs, a few extensions are useful.

Partitioning Function: The users of MapReduce specify the number of reduce tasks/output files that they desire (R). Data gets partitioned across these tasks using a partitioning function on the intermediate key. A default partitioning function is provided that uses hashing (e.g. "*hash(key) mod R"*). This tends to result in fairly well-balanced partitions. In some cases, however, it is useful to partition data by some other function of the key. For example, sometimes the output keys are URLs, and all entries for a single host should end up in the same output file. To support situations like this, the user of the MapReduce library can provide a special partitioning function. For example, using *"hash(Hostname(urlkey)) mod R"* as the partitioning function causes all URLs from the same host to end up in the same output file.

Combiner Function: In some cases, there is significant repetition in the intermediate keys produced by each map task, and the user-specified *Reduce* function is commutative and associative. A good example of this is the word counting example. Since word frequencies tend to follow a Zipf distribution, each map task will produce hundreds or thousands of records of the form <the, 1>. All of these counts will be sent over the network to a single reduce task and then added together by the *Reduce* function to produce one number. The user could specify an optional *Combiner* function that does partial merging of this data before it is sent over the network.

The *Combiner* function is executed on each machine that performs a map task. Typically the same code is used to implement both the combiner and the reduce functions. The only difference between a reduce function and a combiner function is how the MapReduce library handles the output of the function. The output of a reduce function is written to the final output file. The output of a combiner function is written to an intermediate file that will be sent to a reduce task.

Partial combining significantly speeds up certain classes of MapReduce operations.

2.1.1.6. Performance

The performance of MapReduce is measured on two computations running on a large cluster of machines. One computation searches through approximately one terabyte of

data looking for a particular pattern. The other computation sorts approximately one terabyte of data.

These two programs are representative of a large subset of the real programs written by users of MapReduce - one class of programs shuffles data from one representation to another, and another class extracts a small amount of interesting data from a large data set.

Cluster Configuration: All of the programs were executed on a cluster that consisted of approximately 1800 machines. Each machine had two 2GHz Intel Xeon processors with Hyper-Threading enabled, 4GB of memory, two 160GB IDE disks, and a gigabit Ethernet link. The machines were arranged in a two-level tree-shaped switched network with approximately 100-200 Gbps of aggregate bandwidth available at the root. All of the machines were in the same hosting facility and therefore the round-trip time between any pair of machines was less than a millisecond.

Out of the 4GB of memory, approximately 1-1.5GB was reserved by other tasks running on the cluster. The programs were executed on a weekend afternoon, when the CPUs, disks, and network were mostly idle.

Grep: The *grep* program scans through 10^{10} 100-byte records, searching for a relatively rare three-character pattern (the pattern occurs in 92,337 records). The input is split into approximately 64MB pieces (M = 15000), and the entire output is placed in one file (R = 1).

The entire computation takes approximately 150 seconds from start to finish. This includes about a minute of start-up overhead. The overhead is due to the propagation of the program to all worker machines, and delays interacting with GFS to open the set of 1000 input files and to get the information needed for the locality optimization.

Sort: The *sort* program sorts 10^{10} 100-byte records (approximately 1 terabyte of data). This program is modelled after the TeraSort benchmark.

The sorting program consists of less than 50 lines of user code. A three-line *Map* function extracts a 10-byte sorting key from a text line and emits the key and the original text line as the intermediate key/value pair. A built-in *Identity* function was used as the *Reduce* operator. This functions passes the intermediate key/value pair unchanged as the output key/value pair. The final sorted output is written to a set of 2-way replicated GFS files (i.e., 2 terabytes are written as the output of the program).

As before, the input data is split into 64MB pieces (M = 15000). We partition the sorted output into 4000 files (R = 4000). The partitioning function uses the initial bytes of the key to segregate it into one of R pieces.

The partitioning function for this benchmark has built-in knowledge of the distribution of keys. In a general sorting program, a pre-pass MapReduce operation should be used that would collect a sample of the keys and use the distribution of the sampled keys to compute split-points for the final sorting pass.

Including start-up overhead, the entire computation takes 891 seconds. This is similar to the current best reported result of 1057 seconds for the TeraSort benchmark.

Effect of Backup Tasks: There was an execution of the sort program with backup tasks disabled. After 960 seconds, all except 5 of the reduce tasks are completed. However these last few stragglers don't finish until 300 seconds later. The entire computation takes 1283 seconds, an increase of 44% in elapsed time.

Machine Failures: Finally there was an execution of the sort program where 200 out of 1746 worker processes were intentionally killed several minutes into the computation. The underlying cluster scheduler immediately restarted new worker processes on these machines (since only the processes were killed, the machines were still functioning properly).

The re-execution of this map work happens relatively quickly. The entire computation finishes in 933 seconds including start-up overhead (just an increase of 5% over the normal execution time).

2.1.2 Hadoop

2.1.2.1. A Brief History of Hadoop

Hadoop was created by Doug Cutting, the creator of Apache Lucene, the widely used text search library. Hadoop has its origins in Apache Nutch, an open source web search engine, itself a part of the Lucene project.

Building a web search engine from scratch was an ambitious goal, for not only is the software required to crawl and index websites complex to write, but it is also a challenge to run without a dedicated operations team, since there are so many moving parts. It's expensive too: Mike Cafarella and Doug Cutting estimated a system supporting a 1- billion-page index would cost around half a million dollars in hardware, with a monthly running cost of $30,000. Nevertheless, they believed it was a worthy goal, as it would open up and ultimately democratize search engine algorithms.

Nutch was started in 2002, and a working crawler and search system quickly emerged. However, they realized that their architecture wouldn't scale to the billions of pages on the Web. Help was at hand with the publication of a paper in 2003 that described the architecture of Google's distributed filesystem, called GFS, which was being used in production at Google. GFS, or something like it, would solve their storage needs for the very large files generated as a part of the web crawl and indexing process. In particular, GFS would free up time being spent on administrative tasks such as managing storage nodes. In 2004, they set about writing an open source implementation, the Nutch Distributed Filesystem (NDFS).

In 2004, Google published the paper that introduced MapReduce to the world. Early in 2005, the Nutch developers had a working MapReduce implementation in Nutch, and by the middle of that year all the major Nutch algorithms had been ported to run using MapReduce and NDFS.

NDFS and the MapReduce implementation in Nutch were applicable beyond the realm of search, and in February 2006 they moved out of Nutch to form an independent subproject of Lucene called Hadoop. At around the same time, Doug Cutting joined Yahoo!, which provided a dedicated team and the resources to turn Hadoop into a system that ran at web scale. This was demonstrated in February 2008 when Yahoo! announced that its production search index was being generated by a 10,000-core Hadoop cluster.

In January 2008, Hadoop was made its own top-level project at Apache, confirming its success and its diverse, active community. By this time Hadoop was being used by many other companies besides Yahoo!, such as Last.fm, Facebook, and the *New York Times*.

In one well-publicized feat, the *New York Times* used Amazon's EC2 compute cloud to crunch through four terabytes of scanned archives from the paper converting them to PDFs for the Web. The processing took less than 24 hours to run using 100 machines, and the project probably wouldn't have been embarked on without the combination of Amazon's pay-by-the-hour model (which allowed the NYT to access a large number of machines for a short period), and Hadoop's easy-to-use parallel programming model.

In April 2008, Hadoop broke a world record to become the fastest system to sort a terabyte of data. Running on a 910-node cluster, Hadoop sorted one terabyte in 209 seconds (just under 3½ minutes), beating the previous year's winner of 297 seconds. In November of the same year, Google reported that its MapReduce implementation sorted one terabyte in

68 seconds. In May 2009, it was announced that a team at Yahoo! used Hadoop to sort one terabyte in 62 seconds. [9]

The Origin of the Name "Hadoop"

The name Hadoop is not an acronym; it's a made-up name. The project's creator, Doug Cutting, explains how the name came about:

> *"The name my kid gave a stuffed yellow elephant. Short, relatively easy to spell and pronounce, meaningless, and not used elsewhere: those are my naming criteria. Kids are good at generating such. Googol is a kid's term."*

2.1.2.2. How MapReduce Works in Hadoop

The whole process is illustrated in Figure 2.2. At the highest level, there are four independent entities:

- The client, which submits the MapReduce job.
- The jobtracker, which coordinates the job run. The jobtracker is a Java application whose main class is JobTracker.
- The tasktrackers, which run the tasks that the job has been split into. Tasktrackers are Java applications whose main class is TaskTracker.
- The distributed filesystem (normally HDFS), which is used for sharing job files between the other entities.

Job Submission: The runJob() method on JobClient is a convenience method that creates a new JobClient instance and calls submitJob() on it (step 1 in Figure 2.2). Having submitted the job, runJob() polls the job's progress once a second, and reports the progress to the console if it has changed since the last report. When the job is complete, if it was successful, the job counters are displayed. Otherwise, the error that caused the job to fail is logged to the console.

The job submission process implemented by JobClient's submitJob() method does the following:

1. Asks the jobtracker for a new job ID (by calling getNewJobId() on JobTracker() (step 2).
2. Checks the output specification of the job. For example, if the output directory has not been specified or it already exists, the job is not submitted and an error is thrown to the MapReduce program.
3. Computes the input splits for the job. If the splits cannot be computed, because the input paths don't exist, for example, then the job is not submitted and an error is thrown to the MapReduce program.
4. Copies the resources needed to run the job, including the job JAR file, the configuration file and the computed input splits, to the jobtracker's filesystem in a directory named after the job ID. The job JAR is copied with a high replication factor (controlled by the mapred.submit.replication property, which defaults to 10) so that there are lots of copies across the cluster for the tasktrackers to access when they run tasks for the job (step 3).
5. Tells the jobtracker that the job is ready for execution (by calling submitJob() on JobTracker) (step 4).

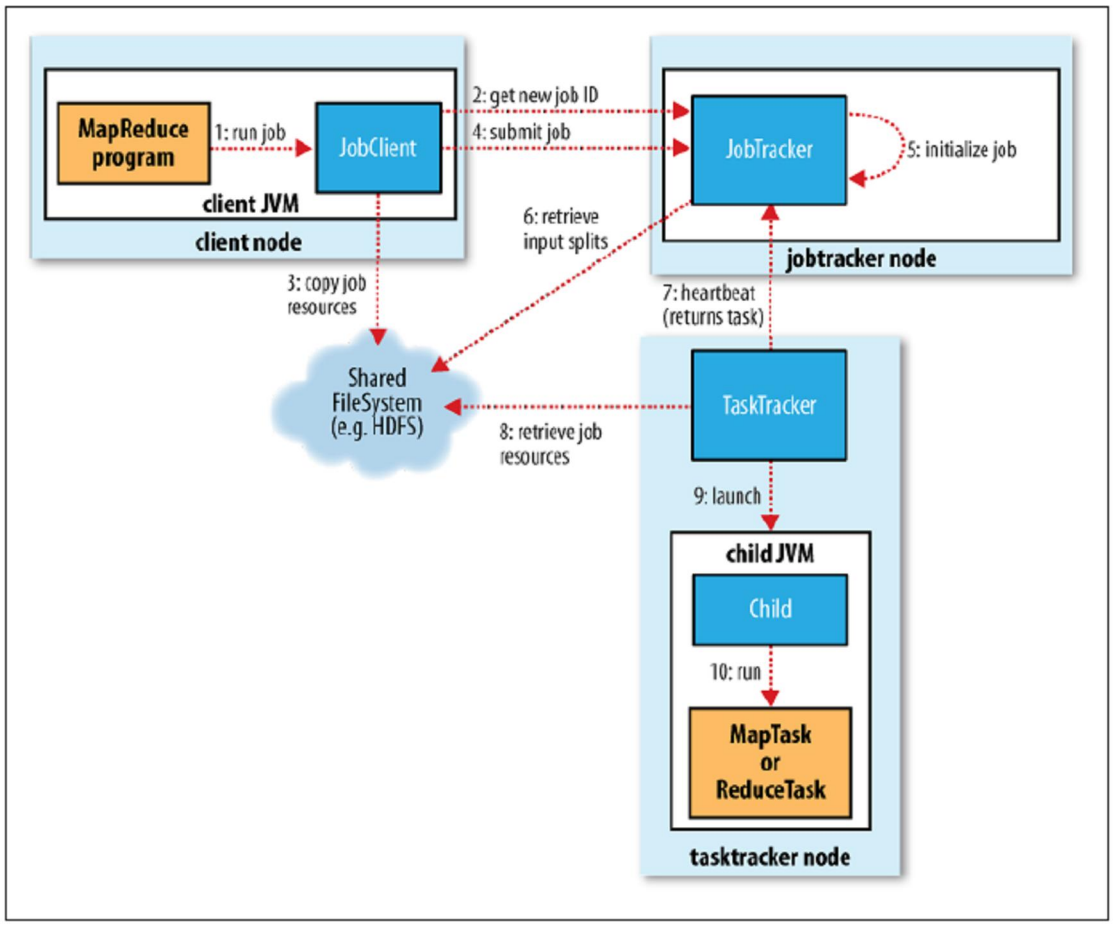


Fig. 2.2 How Hadoop runs a MapReduce job [9]

Job Initialization: When the JobTracker receives a call to its submitJob() method, it puts it into an internal queue from where the job scheduler will pick it up and initialize it. Initialization involves creating an object to represent the job being run, which encapsulates its tasks, and bookkeeping information to keep track of the tasks' status and progress (step 5).

To create the list of tasks to run, the job scheduler first retrieves the input splits computed by the JobClient from the shared filesystem (step 6). It then creates one map task for each split. The number of reduce tasks to create is determined by the mapred.reduce.tasks property in the JobConf, which is set by the setNumReduce Tasks() method, and the scheduler simply creates this number of reduce tasks to be run. Tasks are given IDs at this point.

Task Assignment: Tasktrackers run a simple loop that periodically sends heartbeat method calls to the jobtracker. Heartbeats tell the jobtracker that a tasktracker is alive, but they also double as a channel for messages. As a part of the heartbeat, a tasktracker will indicate whether it is ready to run a new task, and if it is, the jobtracker will allocate it a task, which it communicates to the tasktracker using the heartbeat return value (step 7).

Before it can choose a task for the tasktracker, the jobtracker must choose a job to select the task from. There are various scheduling algorithms, but the default one simply maintains a priority list of jobs. Having chosen a job, the jobtracker now chooses a task for the job. Tasktrackers have a fixed number of slots for map tasks and for reduce tasks: for example, a tasktracker may be able to run two map tasks and two reduce tasks simultaneously. (The precise number depends on the number of cores and the amount of

memory on the tasktracker.) The default scheduler fills empty map task slots before reduce task slots, so if the tasktracker has at least one empty map task slot, the jobtracker will select a map task; otherwise, it will select a reduce task.

To choose a reduce task the jobtracker simply takes the next in its list of yet-to-be-run reduce tasks, since there are no data locality considerations. For a map task, however, it takes account of the tasktracker's network location and picks a task whose input split is as close as possible to the tasktracker. In the optimal case, the task is *data-local*, that is, running on the same node that the split resides on. Alternatively, the task may be *rack-local*: on the same rack, but not the same node, as the split. Some tasks are neither data-local nor rack-local and retrieve their data from a different rack from the one they are running on. You can tell the proportion of each type of task by looking at a job's counters.

Task Execution: Now the tasktracker has been assigned a task, the next step is for it to run the task. First, it localizes the job JAR by copying it from the shared filesystem to the tasktracker's filesystem. It also copies any files needed from the distributed cache by the application to the local disk (step 8). Second, it creates a local working directory for the task, and un-jars the contents of the JAR into this directory. Third, it creates an instance of TaskRunner to run the task.

TaskRunner launches a new Java Virtual Machine (step 9) to run each task in (step 10), so that any bugs in the user-defined map and reduce functions don't affect the tasktracker (by causing it to crash or hang, for example). It is however possible to reuse the JVM between tasks.

The child process communicates with its parent through the *umbilical* interface. This way it informs the parent of the task's progress every few seconds until the task is complete.

Job Completion: When the jobtracker receives a notification that the last task for a job is complete, it changes the status for the job to "successful." Then, when the JobClient polls for status, it learns that the job has completed successfully, so it prints a message to tell the user, and then returns from the runJob() method.

The jobtracker also sends a HTTP job notification if it is configured to do so. This can be configured by clients wishing to receive callbacks, via the job.end.notification.url property.

Last, the jobtracker cleans up its working state for the job, and instructs tasktrackers to do the same (so intermediate output is deleted, for example).

2.1.2.3. Failures

In the real world, user code is buggy, processes crash, and machines fail. One of the major benefits of using Hadoop is its ability to handle such failures and allow your job to complete.

Task Failure: Consider first the case of the child task failing. The most common way that this happens is when user code in the map or reduce task throws a runtime exception. If this happens, the child JVM reports the error back to its parent tasktracker, before it exits. The error ultimately makes it into the user logs. The tasktracker marks the task attempt as *failed*, freeing up a slot to run another task.

Another failure mode is the sudden exit of the child JVM—perhaps there is a JVM bug that causes the JVM to exit for a particular set of circumstances exposed by the Map-Reduce user code. In this case, the tasktracker notices that the process has exited, and marks the attempt as failed.

Hanging tasks are dealt with differently. The tasktracker notices that it hasn't received a progress update for a while, and proceeds to mark the task as failed. The child JVM process will be automatically killed after this period. The timeout period after which tasks are considered failed is normally 10 minutes, and can be configured on a per-job basis (or a cluster basis) by setting the mapred.task.timeout property to a value in milliseconds.

When the jobtracker is notified of a task attempt that has failed (by the tasktracker's heartbeat call) it will reschedule execution of the task. The jobtracker will try to avoid rescheduling the task on a tasktracker where it has previously failed. Furthermore, if a task fails more than four times, it will not be retried further. This value is configurable: the maximum number of attempts to run a task is controlled by the mapred.map.max.attempts property for map tasks, and mapred.reduce.max.attempts for reduce tasks. By default, if any task fails more than four times (or whatever the maximum number of attempts is configured to), the whole job fails.

For some applications it is undesirable to abort the job if a few tasks fail, as it may be possible to use the results of the job despite some failures. In this case, the maximum percentage of tasks that are allowed to fail without triggering job failure can be set for the job. Map tasks and reduce tasks are controlled independently, using the mapred.max.map.failures.percent and mapred.max.reduce.failures.percent properties.

A task attempt may also be *killed*, which is different from it failing. A task attempt may be killed because it is a speculative, or because the tasktracker it was running on failed, and the jobtracker marked all the task attempts running on it as killed. Killed task attempts do not count against the number of attempts to run the task (as set by mapred.map.max.attempts and mapred.reduce.max.attempts), since it wasn't the task's fault that an attempt was killed.

Users may also kill or fail task attempts using the web UI or the command line (type hadoop job to see the options). Jobs may also be killed by the same mechanisms.

Tasktracker Failure: Failure of a tasktracker is another failure mode. If a tasktracker fails by crashing, or running very slowly, it will stop sending heartbeats to the jobtracker (or send them very infrequently). The jobtracker will notice a tasktracker that has stopped sending heartbeats (if it hasn't received one for 10 minutes, configured via the mapred.tasktracker.expiry.interval property, in milliseconds) and remove it from its pool of tasktrackers to schedule tasks on. The jobtracker arranges for map tasks that were run and completed successfully on that tasktracker to be rerun if they belong to incomplete jobs, since their intermediate output residing on the failed tasktracker's local filesystem may not be accessible to the reduce task. Any tasks in progress are also rescheduled.

A tasktracker can also be *blacklisted* by the jobtracker, even if the tasktracker has not failed. A tasktracker is blacklisted if the number of tasks that have failed on it is significantly higher than the average task failure rate on the cluster. Blacklisted tasktrackers can be restarted to remove them from the jobtracker's blacklist.

Jobtracker Failure: Failure of the jobtracker is the most serious failure mode. Currently, Hadoop has no mechanism for dealing with failure of the jobtracker—it is a single point of failure— so in this case the job fails. However, this failure mode has a low chance of occurring since the chance of a particular machine failing is low. It is possible that a future release of Hadoop will remove this limitation by running multiple jobtrackers, only one of which is the primary jobtracker at any time.

2.1.2.4. Speculative Execution

The MapReduce model is to break jobs into tasks and run the tasks in parallel to make the overall job execution time smaller than it would otherwise be if the tasks ran sequentially. This makes job execution time sensitive to slow-running tasks, as it takes only one slow task to make the whole job take significantly longer than it would have done otherwise. When a job consists of hundreds or thousands of tasks, the possibility of a few straggling tasks is very real.

Tasks may be slow for various reasons, including hardware degradation, or software mis-configuration, but the causes may be hard to detect since the tasks still complete successfully, albeit after a longer time than expected. Hadoop doesn't try to diagnose and fix slow-running tasks; instead, it tries to detect when a task is running slower than expected and launches another, equivalent, task as a backup. This is termed *speculative execution* of tasks.

It's important to understand that speculative execution does not work by launching two duplicate tasks at about the same time so they can race each other. This would be wasteful of cluster resources. Rather, a speculative task is launched only after all the tasks for a job have been launched, and then only for tasks that have been running for some time (at least a minute), and have failed to make as much progress, on average, as the other tasks from the job. When a task completes successfully, any duplicate tasks that are running are killed since they are no longer needed. So if the original task completes before the speculative task then the speculative task is killed; on the other hand, if the speculative task finishes first, then the original is killed.

2.1.3. Phoenix

Phoenix is a shared-memory implementation of Google's MapReduce model for data-intensive processing tasks. Phoenix can be used to program multi-core chips as well as shared-memory multiprocessors (SMPs and ccNUMAs). Phoenix was developed as a class project for the Advanced Processor Architecture course at Stanford. Original Phoenix code was developed by Colby Ranger, Ramanan Raghuraman, and Arun Penmetsa under the supervision of Christos Kozyrakis [12] and released in the April of 2007. Since then it has been significantly updated to improve scalability and portability. Compared to the original release, version 2.0 includes:

- Linux (x86_64) support
- Enhanced NUMA support for Solaris environment
- Improved performance and stability

Phoenix 2.0 was developed by Richard Yoo and Anthony Romano [13].

Phoenix++ is a C++ reimplementation of Phoenix 2. It was designed to provide a modular, extensible pipeline that can be easily adapted by the user to the characteristics of a particular workload. Phoenix++ was developed by Justin Talbot and Richard Yoo [14]. For the purposes of this thesis, Phoenix++ won't be presented.

2.1.3.1. The Phoenix API

The Phoenix implementation provides an application-programmer interface (API) for C and C++. The API includes two sets of functions summarized in Table 2.1. The first set is provided by Phoenix and is used by the programmer's application code to initialize the

system and emit output pairs (1 required and 2 optional functions). The second set includes the functions that the programmer defines (3 required and 2 optional functions). Apart from the Map and Reduce functions, the user provides functions that partition the data before each step and a function that implements key comparison. Note that the API is quite small compared to other models. The API is type agnostic. The function arguments are declared as void pointers wherever possible to provide flexibility in their declaration and fast use without conversion overhead. In contrast, the Google implementation uses strings for arguments as string manipulation is inexpensive compared to remote procedure calls and file accesses.

Function Description	R/O
Functions Provided by Runtime	
`int phoenix_scheduler (scheduler_args_t * args)` Initializes the runtime system. The `scheduler_args_t` struct provides the needed function & data pointers	R
`void emit_intermediate(void *key, void *val, int key_size)` Used in Map to emit an intermediate output <key, value> pair. Required if the Reduce is defined	O
`void emit(void *key, void *val)` Used in Reduce to emit a final output pair	O
Functions Defined by User	
`int (*splitter_t)(void *, int, map_args_t *)` Splits the input data across Map tasks. The arguments are the input data pointer, the unit size for each task, and the input buffer pointer for each Map task	R
`void (*map_t)(map_args_t*)` The Map function. Each Map task executes this function on its input	R
`int (*partition_t)(int, void *, int)` Partitions intermediate pair for Reduce tasks based on their keys. The arguments are the number of Reduce tasks, a pointer to the keys, and a the size of the key. Phoenix provides a default partitioning function based on key hashing	O
`void (*reduce_t)(void *, void **, int)` The Reduce function. Each reduce task executes this on its input. The arguments are a pointer to a key, a pointer to the associated values, and value count. If not specified, Phoenix uses a default *identity* function	O
`int (*key_cmp_t)(const void *, const void*)` Function that compares two keys	R

Table 2.1 The functions in the Phoenix API [12]

The data structure used to communicate basic function information and buffer allocation between the user code and runtime is of type scheduler_args_t. Its fields are summarized in Table 2.2. The basic fields provide pointers to input/output data buffers and to the user-provided functions. They must be properly set by the programmer before calling phoenix scheduler(). The remaining fields are optionally used by the programmer to control scheduling decisions by the runtime. There are additional data structure types to facilitate communication between the Splitter, Map, Partition, and Reduce functions. These types use pointers whenever possible to implement communication without actually copying significant amounts of data.

The API guarantees that within a partition of the intermediate output, the pairs will be processed in key order. This makes it easier to produce a sorted final output which is often desired. There is no guarantee in the processing order of the original input during the Map stage. These assumptions did not cause any complications with the programs we examined. In general it is up to the programmer to verify that the algorithm can be expressed with the Phoenix API given these restrictions.

Field	Description
Basic Fields	
Input_data	Input data pointer; passed to the Splitter by the runtime
Data_size	Input dataset size
Output_data	Output data pointer; buffer space allocated by user
Splitter	Pointer to Splitter function
Map	Pointer to Map function
Reduce	Pointer to Reduce function
Partition	Pointer to Partition function
Key_cmp	Pointer to key compare function
Optional Fields for Performance Tuning	
Unit_size	Pairs processed per Map/Reduce task
L1_cache_size	L1 data cache size in bytes
Num_Map_workers	Maximum number of threads (workers) for Map tasks
Num_Reduce_workers	Maximum number of threads (workers) for Reduce tasks
Num_Merge_workers	Maximum number of threads (workers) for Merge tasks
Num_procs	Maximum number of processors cores used

Table 2.2 The *scheduler_args_t* data structure type. [12]

The Phoenix API does not rely on any specific compiler options and does not require a parallelizing compiler. However, it assumes that its functions can freely use stack-allocated and heap-allocated structures for private data. It also assumes that there is no communication through shared-memory structures other than the input/output buffers for these functions.

2.1.3.2. The Phoenix Runtime

The Phoenix runtime was developed on top of Pthreads, but can be easily ported to other shared memory thread packages.

Basic Operation and Control Flow: Figure 2.3 shows the basic data flow for the runtime system. The runtime is controlled by the scheduler, which is initiated by user code. The scheduler creates and manages the threads that run all Map and Reduce tasks. It also manages the buffers used for task communication. The programmer provides the scheduler with all the required data and function pointers through the scheduler args t structure. After initialization, the scheduler determines the number of cores to use for this computation. For each core, it spawns a worker thread that is dynamically assigned some number of Map and Reduce tasks.

To start the Map stage, the scheduler uses the Splitter to divide input pairs into equally sized units to be processed by the Map tasks. The Splitter is called once per Map task and returns a pointer to the data the Map task will process. The Map tasks are allocated dynamically to workers and each one emits intermediate <key,value> pairs. The Partition function splits the intermediate pairs into units for the Reduce tasks. The function ensures all values of the same key go to the same unit. Within each buffer, values are ordered by key to assist with the final sorting. At this point, the Map stage is over. The scheduler must wait for all Map tasks to complete before initiating the Reduce stage.

Reduce tasks are also assigned to workers dynamically, similar to Map tasks. The one difference is that, while with Map tasks we have complete freedom in distributing pairs across tasks, with Reduce we must process all values for the same key in one task. Hence, the Reduce stage may exhibit higher imbalance across workers and dynamic scheduling is more important. The output of each Reduce task is already sorted by key. As the last step, the final output from all tasks is merged into a single buffer, sorted by keys. The merging takes place in $\log_2(P/2)$ steps, where P is the number of workers used.

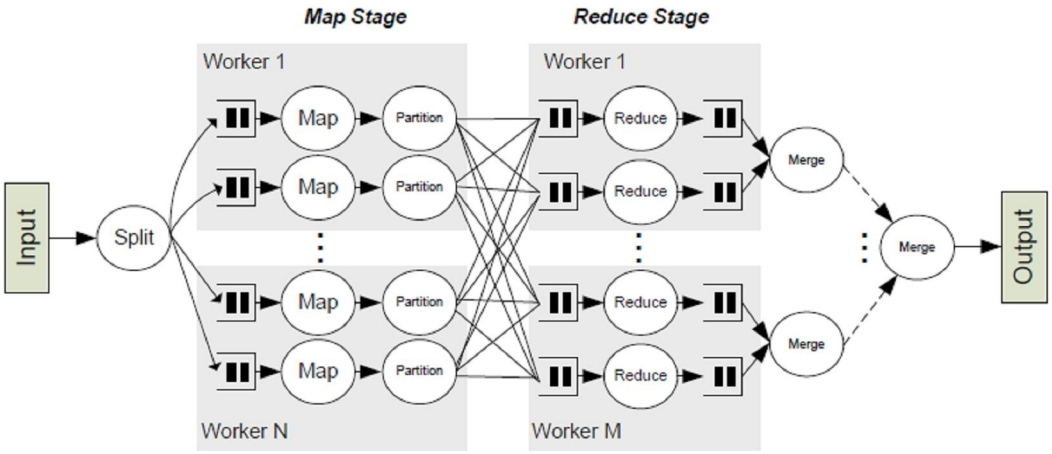


Fig. 2.3 The basic data flow for the Phoenix runtime [12]

Buffer Management: Two types of temporary buffers are necessary to store data between the various stages. All buffers are allocated in shared memory but are accessed in a well specified way by a few functions. Whenever buffers have to be re-arranged (e.g., split across tasks), pointers are manipulated instead of the actual pairs, which may be large in size. The intermediate buffers are not directly visible to user code.

Map-Reduce buffers are used to store the intermediate output pairs. Each worker has its own set of buffers. The buffers are initially sized to a default value and then resized dynamically as needed. At this stage, there may be multiple pairs with the same key. To accelerate the Partition function, the Emit intermediate function stores all values for the same key in the same buffer. At the end of the Map task, we sort each buffer by key order. Reduce-Merge buffers are used to store the outputs of Reduce tasks before they are sorted. At this stage, each key has only one value associated with it. After sorting, the final output is available in the user allocated Output data buffer.

Fault Recovery: The runtime provides support for fault tolerance for transient and permanent faults during Map and Reduce tasks. It focuses mostly on recovery with some limited support for fault detection.

Phoenix detects faults through timeouts. If a worker does not complete a task within a reasonable amount of time, then a failure is assumed. The execution time of similar tasks on other workers is used as a yardstick for the timeout interval. Of course, a fault may cause a task to complete with incorrect or incomplete data instead of failing completely. Phoenix has no way of detecting this case on its own and cannot stop an affected task from potentially corrupting the shared memory. To address this shortcoming, one should combine the Phoenix runtime with known error detection techniques. Due to the functional nature of the MapReduce model, Phoenix can actually provide information that simplifies error detection. For example, since the address ranges for input and output buffers are known, Phoenix can notify the hardware about which load/store addresses to shared structures should be considered safe for each worker and which should signal a potential fault.

Once a fault is detected or at least suspected, the runtime attempts to re-execute the failed task. Since the original task may still be running, separate output buffers are allocated for the new task to avoid conflicts and data corruption. When one of the two tasks completes successfully, the runtime considers the task completed and merges its result with the rest of the output data for this stage. The scheduler initially assumes that the fault was a transient one and assigns the replicated task to the same worker. If the task fails a few times or a

worker exhibits a high frequency of failed tasks overall, the scheduler assumes a permanent fault and no further tasks are assigned to this worker.

The current Phoenix code does not provide fault recovery for the scheduler itself. The scheduler runs only for a very small fraction of the time and has a small memory footprint, hence it is less likely to be affected by a transient error. On the other hand, a fault in the scheduler has more serious implications for the program correctness. We can use known techniques such as redundant execution or checkpointing to address this shortcoming.

Google's MapReduce system uses a different approach for worker fault tolerance. Towards the end of the Map or Reduce stage, they always spawn redundant executions of the remaining tasks, as they proactively assume that some workers have performance or failure issues. This approach works well in large clusters where hundreds of machines are available for redundant execution and failures are more frequent. On multi-core and symmetric multiprocessor systems, the number of processors and frequency of failures are much smaller hence this approach is less profitable.

Concurrency and Locality Management: The runtime makes scheduling decisions that affect the overall parallel efficiency. In general, there are three scheduling approaches one can employ: 1) use a default policy for the specific system which has been developed taking into account its characteristics; 2) dynamically determine the best policy for each decision by monitoring resource availability and runtime behaviour; 3) allow the programmer to provide application specific policies. Phoenix employs all three approaches in making the scheduling decisions described below.

Number of Cores and Workers/Core: Since MapReduce programs are data-intensive, workers are spawned to all available cores. In a multi-programming environment, the scheduler can periodically check the system load and scale its usage based on system-wide priorities. The mechanism for dynamically scaling the number of workers is already in place to support fault recovery. In systems with multithreaded cores (e.g., UltraSparc T1), one worker per hardware thread is spawned. This typically maximizes the system throughput even if an individual task takes longer.

Task Assignment: To achieve load balance, Map and Reduce task are assigned to workers dynamically. Since all Map tasks must execute before Reduce tasks, it is difficult to exploit any producer-consumer locality between Map and Reduce tasks.

Task Size: Each Map task processes a unit of the input data. Given the size of an element of input data, Phoenix adjusts the unit size so that the input and output data for a Map task fit in the L1 data cache. Note that for some computations, there is little temporal locality within Map or Reduce stages. Nevertheless, partitioning the input at L1 cache granularity provides a good trade-off between lower overheads (few larger units) and load balance (more smaller units). The programmer can vary this parameter given specific knowledge of the locality within a task, the amount of output data produced per task, or the processing overheads.

Partition Function: The partition function determines the distribution of intermediate data. The default partition function partitions keys evenly across tasks. This may be suboptimal since keys may have a different number of values associated with them. The user can provide a function that has application-specific knowledge of the values' distribution and reduces imbalance.

2.1.3.3. Performance

Shared Memory Systems: Phoenix was run on the two shared-memory systems described in Table 2.3. Both systems are based on the Sparc architecture. Nevertheless, Phoenix should work without modifications on any architecture that supports the Pthreads library. The CMP system is based on the UltraSparc T1 multi-core chip with 8 multithreaded cores sharing the L2 cache. The SMP system is a symmetric multiprocessor with 24 chips. The use of two drastically different systems allows us to evaluate if the Phoenix runtime can deliver on its promise: the same program should run as efficiently as possible on any type of shared-memory system without any involvement by the user.

	CMP	SMP
Model	Sun Fire T1200	Sun Ultra-Enterprise 6000
CPU Type	UltraSparc T1 single-issue in-order	UltraSparc II 4-way issue in-order
CPU Count	8	24
Threads/CPU	4	1
L1 Cache	8KB 4-way SA	16KB DM
L2 Size	3MB 12-way SA shared	512KB per CPU (off chip)
Clock Freq.	1.2 GHz	250 MHz

Table 2.3 The characteristics of the CMP and SMP systems used to evaluate Phoenix. [12]

Applications: 8 benchmarks were used. They represent key computations from application domains such as enterprise computing (Word Count, Reverse Index, String Match), scientific computing (Matrix Multiply), artificial intelligence (Kmeans, PCA, Linear Regression), and image processing (Histogram). Three datasets were used for each benchmark (S, M, L) to test locality and scalability issues. The sequential code for all benchmarks serves as the baseline for speedups. From that, a MapReduce version using Phoenix and a conventional parallel version using P-threads were developed. The P-threads code is statically scheduled.

The following are brief descriptions of the main mechanisms used to code each benchmark with Phoenix.

Word Count: It counts the frequency of occurence for each word in a set of files. The Map tasks process different sections of the input files and return intermediate data that consist of a word (key) and a value of 1 to indicate that the word was found. The Reduce tasks add up the values for each word (key).

Reverse Index: It traverses a set of HTML files, extracts all links, and compiles an index from links to files. Each Map task parses a collection of HTML files. For each link it finds, it outputs an intermediate pair with the link as the key and the file info as the value. The Reduce task combines all files referencing the same link into a single linked-list.

Matrix Multiply: Each Map task computes the results for a set of rows of the output matrix and returns the (x,y) location of each element as the key and the result of the computation as the value. The Reduce task is just the identity function.

String Match: It processes two files: the “encrypt” file contains a set of encrypted words and a “keys” file contains a list of non-encrypted words. The goal is to encrypt the words in the “keys” file to determine which words were originally encrypted to generate the “encrypt file”. Each Map task parses a portion of the “keys” file and returns a word in the “keys” file as the key and a flag to indicate whether it was a match as the value. The reduce task is just the identity function.

KMeans: It implements the popular kmeans algorithm that groups a set of input data points into clusters. Since it is iterative, the Phoenix scheduler is called multiple times until it converges. In each iteration, the Map task takes in the existing mean vectors and a subset of the data points. It finds the distance between each point and each mean and assigns the point to the closest cluster. For each point, it emits the cluster id as the key and the data vector as the value. The Reduce task gathers all points with the same cluster-id, and finds their centriod (mean vector). It emits the cluster id as the key and the mean vector as the value.

PCA: It performs a portion of the Principal Component Analysis algorithm in order to find the mean vector and the covariance matrix of a set of data points. The data is presented in a matrix as a collection of column vectors. The algorithm uses two MapReduce iterations. To find the mean, each Map task in the first iteration computes the mean for a set of rows and emits the row numbers as the keys, and the means as the values. In the second iteration, the Map task is assigned to compute a few elements in the required covariance matrix, and is provided with the data required to calculate the value of those elements. It emits the element row and column numbers as the key, and the covariance as the value. The Reduce task is the identity in both iterations.

Histogram: It analyzes a given bitmap image to compute the frequency of occurrence of a value in the 0-255 range for the RGB components of the pixels. The algorithm assigns different portions of the image to different Map tasks, which parse the image and insert the frequency of component occurrences into arrays. The reduce tasks sum up these numbers across all the portions.

Linear Regression: It computes the line that best fits a given set of coordinates in an input file. The algorithm assigns different portions of the file to different map tasks, which compute certain summary statistics like the sum of squares. The reduce tasks compute these statistics across the entire data set in order to finally determine the best fit line.

Basic Performance Evaluation: Phoenix provides significant speedups with both systems for all processor counts and across all benchmarks. In some cases, such as MatrixMultiply, superlinear speedups were observed due to caching effects (beneficial sharing in the CMP, increased cache capacity in the SMP with more cores). At high core counts, the SMP system often suffers from saturation of the bus that interconnects the processors (e.g., PCA and Histogram). With a large number of cores, some applications suffered from load imbalance in the Reduce stage (e.g., Word Count). Reverse Index achieves the highest speedups due to a number of reasons. Its code uses array based heaps to track indices. As work is distributed across more cores, the heaps accessed by each core are smaller and operations on them become significantly faster. Another contributor to the superlinear speedup is that it spends a signficant portion of its execution time on the final merging/sorting of the output data. The additional cores and their caches reduce the merging overhead.

In general, the applications can be classified into two types. The key-based structure that MapReduce uses fits well the algorithm of WordCount, MatrixMultiply, StringMatch, and LinearRegression. Hence, these applications achieve significant speedups across all system sizes. On the other hand, the key-based approach is not the natural choice for Kmeans, PCA, and Histogram. Hence, fitting these algorithms into the MapReduce models leads to significant overheads compared to sequential code and reduces the overall speedup.

Dependency to Dataset Size: It is clear that increasing the dataset leads to higher speedups over the sequential version for most applications. This is due to two reasons. First, a larger dataset allows the Phoenix runtime to better amortize its overheads for task

management, buffer allocation, data splitting and sorting. Such overheads are not dominant if the application is truly data intensive. Second, caching effects are more significant when processing large datasets and load imbalance is rarer. StringMatch and LinearRegression perform similarly across all dataset sizes. This is because even their small datasets contain a large number of elements. Moreover, they perform a significant amount of computation per element in their dataset. Hence, even the small datasets are sufficient to fully utilize the available parallel resources and hide the runtime overheads.

Dependency to Unit Size: Each Map task processes a unit of the input data. Hence, the unit size determines the number of number of Map tasks, their memory footprint, and how well their overhead is amortized. Many applications perform similarly with all unit sizes as there is little temporal locality in the data access. Larger units can lead to better performance for some applications as they reduce significantly the portion of time spent on spawning tasks and merging their outputs (fewer tasks). Histogram benefits from larger units because it reduces the number of intermediate values to merge across tasks. On the other hand, applications with short term temporal locality in their access patterns (e.g. Kmeans and MatrixMultiply) perform better with smaller units as they allow tasks to operate on data within their L1 cache or the data for all the active tasks to fit in the shared L2.

The current implementation of Phoenix uses the user-supplied unit size or determines the unit size based on the input dataset size and the cache size. A better approach is to use a dynamic framework that discovers the best unit size for each program. At the beginning of a data intensive program, the runtime can vary the unit size and monitor the trends in the completion time or other performance indicators (processor utilization, number of misses, etc.) in order to select the best possible value.

2.1.3.4 Optimizing Phoenix for Large-Scale & NUMA (Phoenix 2.0)

The most popular form of large-scale, shared-memory machines today are multi-socket servers that use two to four multi-core chips, with caches and main memory channels physically distributed across those chips. Such a system can readily support hundreds of threads in a single unit, but exhibits variable memory access latencies. Next generation multi-core chips with hundreds of cores on a single die will likely exhibit similar NUMA characteristics.

The original Phoenix runtime targeted CMP and SMP systems with uniform memory access characteristics and 24 to 32 hardware threads. The Sun SPARC Enterprise T5440 system, summarized in Table 2.5 is the new target. Each of the four T2+ chips supports 64 hardware contexts for a total of 256 in the whole system. Each chip has 4 channels of locally attached main memory (DRAM) as well. Notice that the accesses to remote DRAM are 33% slower than the accesses to locally attached memory; any program that uses more than 64 threads will experience such non-uniform latency.

Hardware Settings	
CPU	4 UltraSPARC T2+ chips 8 cores per chip, 8 HW contexts per core total of 256 hardware contexts on system
Per Core L1 Cache	8 KB, 16 B lines, 4-way assoc write through, physically tagged / indexed
Shared L2 Cache	4 MB, 64 B lines, 8-banked, 16-way assoc write back, physically tagged / indexed
Memory	128 GB over 16 FBDIMM channels 4 channels local to each chip 300-cycle latency for local access 100-cycle overhead for remote access 85 GB/sec for read, 42 GB/sec for write
Interconnect	single external coherence hub 130 GB/sec bisection for coherence
Software Settings	
Operating System	Sun Solaris 5.10
Compiler	GCC 4.2.1 with -O3 optimization

Table 2.4 The Sun SPARC Enterprise T5440 [13]

Figure 2.4 shows the scalability of the original Phoenix runtime measured on T5440 with the released applications. Despite the parallelism available in these applications, none of them scales beyond 64 threads, and most of them actually slow down when more threads are involved. This is the result of the increased memory latency, loss of locality, and high contention when utilizing threads across multiple chips.

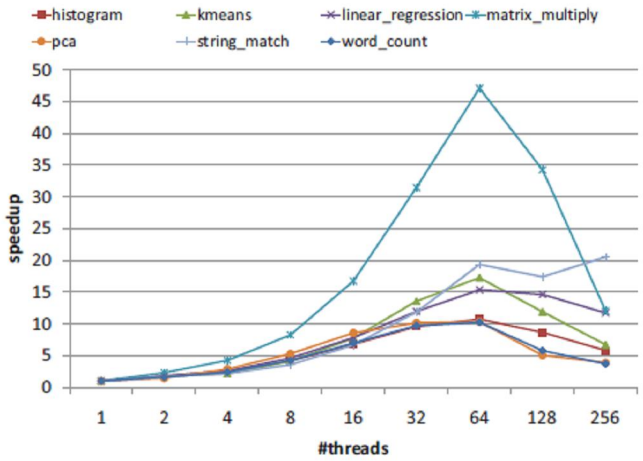


Fig. 2.4 Application speedup for the original Phoenix system [13]

A parallel runtime such as Phoenix continuously interacts with the user application and the operating system. Therefore, it is natural that the optimization strategies for large-scale, NUMA systems are multi-layered. The approach we propose comprises three layers: algorithm, implementation, and OS interaction.

A. Algorithmic Optimizations

To perform well on a NUMA machine, the basic algorithms used in the runtime must be scalable and NUMA aware. For instance, the original Phoenix algorithm is not NUMA aware in that local and remote worker threads are indistinguishable at the algorithm level. While this is not an issue for smallscale systems, it becomes important for locality-aware task distribution in a large-scale, NUMA environment. When the input data for the application is brought into memory via mmap(), Solaris distributes the necessary physical

frames across multiple *locality groups*, i.e., chips and their separate memory channels. If Phoenix blindly assigns map tasks to threads, it could end up having a local thread continuously work on remote data chunks, thus causing additional latency and unnecessary remote traffic.

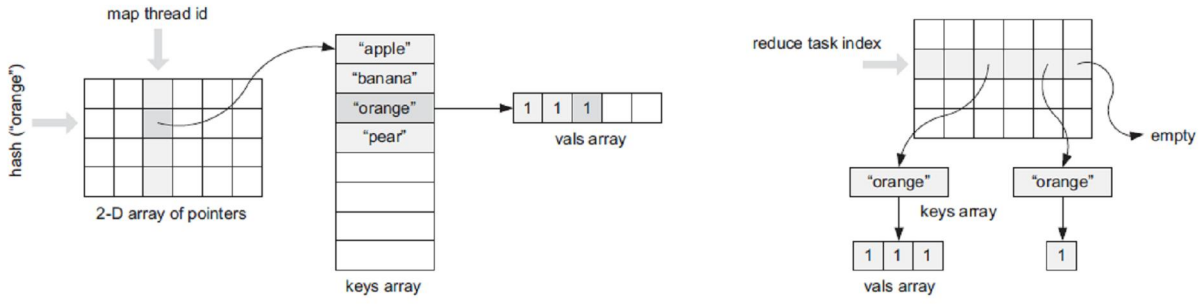
This issue is resolved by introducing a task queue per locality group, and by distributing tasks according to the location of the pertaining data chunk. Map threads retrieve tasks from their local task queue first, and when the queue runs out, they start stealing tasks from remote task queues. Although similar in spirit to the work stealing algorithm utilized in other runtimes, the difference here is that one task queue is maintained for each locality group (instead of each thread), hence creating a load balancing approach that is compatible with NUMA memory hierarchies.

B. Implementation Optimizations

To fully utilize a large-scale machine, applications must include a non-trivial amount of work. This typically implies large input datasets that the runtime system must handle efficiently. Meeting this requirement in MapReduce is quite challenging, since a typical MapReduce application generates a commensurate amount of intermediate and output data as well. Unlike the original MapReduce where data storage and retrieval are limited by the raw bandwidth of the network and the disk subsystem, in Phoenix, the design and performance of the in-memory data structures used to store and retrieve intermediate data are crucial in overall system performance.

Figure 2.5 gives a simplistic view of the core data structure of the original Phoenix, which is used to store the intermediate key / value pairs generated from the map phase. In particular, Figure 2.5(a) depicts the typical access pattern to this structure in the map phase, where a worker thread is storing an <“orange”, 1> intermediate pair. Phoenix internally maintains a 2-D array of pointers to *keys array*, where the width is determined as the number of map workers, and the height is fixed by a default value (256). During the map phase, each map worker uses its thread id to index into this array column-wise. Once the column is determined, the element is indexed by the hash value of the key. All the threads use the same hash function. Therefore, for each thread, a column of pointers acts as a fixed-sized hash table, and one keys array amounts to a hash bucket.

Specifically, each keys array is implemented as a contiguous buffer, and keys are stored sorted to facilitate binary search. Each entry in the keys array also has a pointer to a *vals array*; this structure stores all the values associated with a particular key. To maximize locality, vals array is implemented as a single contiguous array as well. During the map phase, both the keys array and vals array are thread local.



(a) Access pattern during the map phase. (b) Access pattern during the reduce phase.

Fig. 2.5 Phoenix data structure for intermediate key / value pairs [13]

Next, during the reduce phase (Figure 2.5(b)), each row amounts to a reduce task; a reduce thread grabs a row from the matrix and invokes the reduce function over all the keys array in that row. Notice the disparity between how the threads access the 2-D array structure during the map phase (column-wise) and the reduce phase (row-wise); pictorially, every 'crossing' in the access pattern signifies that a worker thread has to access data structures prepared by another thread, which could require remote memory accesses in the NUMA environment. On the other hand, since Phoenix is implemented in C, the 2-D array structure is represented as a row-major array in memory. Hence, the above design optimizes for the locality of reduce phase; map phase exhibits little locality since the sequence of keys confronted during the phase is close to random.

With the medium-sized datasets used on small-scale systems, this data structure design was acceptable. But when the input was significantly increased on the 256-thread NUMA system, some performance pathologies were observed. As described earlier, the keys array structure was implemented as a sorted array to utilize binary search. Although it provided fast lookup, the downside of this implementation was that when the buffer ran out of space, the entire array had to be reallocated. Even worse, when a new key was inserted in the middle of the array, all the keys coming lexicographically after the new key had to be moved. As we increased the input dataset, this problem became more prominent. Similar buffer reallocation issues occurred on the vals array as well.

Improving the keys array structure was more complicated than first expected. Since massive amounts of intermediate pairs were being inserted into the hash table, slight latency increases such as pointer indirection obliterated any performance gains. For example, a design that implemented keys array as a linked list was considered, but failed to improve performance. Replacing the structure with a tree was not an option either, since frequent rebalancing would have been costly. Instead, the number of hash buckets was significantly increased so that on average only one key resides in a keys array.

For the vals array, an iterator interface to the buffer was implemented and exposed this interface to the user reduce function. This allowed to have a buffer that is comprised of few disjoint memory chunks (good for locality purposes) while still maintaining the sequential accessibility through the iterator interface. This design completely removed the reallocation issue. Additionally, to tolerate the increased memory latencies in the NUMA system, prefetching functionality was implemented behind the interface. Note that unlike the map phase where we can avoid accessing remote memory by assigning tasks based on locality, in the reduce phase a worker might not be able to avoid performing remote accesses since the intermediate pairs with the same key may be produced by workers across all the locality groups. Therefore, sequential accesses and prefetching to vals array become important.

The use of combiners was also considered, where each thread invoked the combiner at the end of the map phase to decrease the amount of reduce phase remote memory traffic. However, with the prefetching in place for the reduce phase, combiners made little difference.

After the data structure was improved, other minor factors such as task generation time were affected by the increased input size as well.

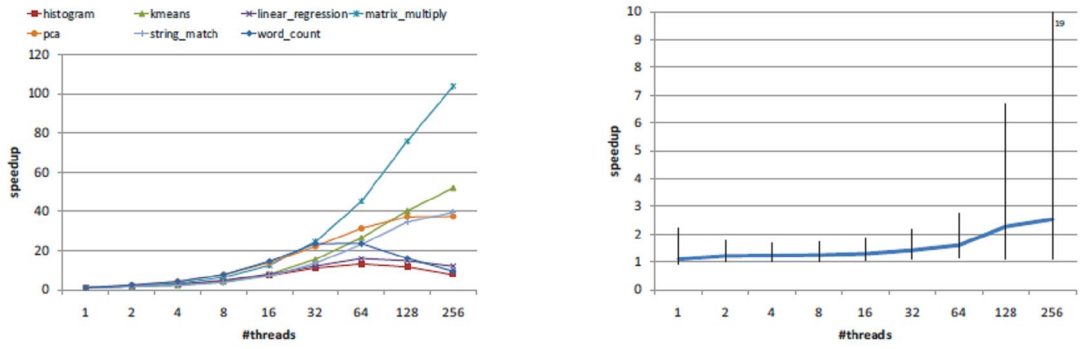
C. OS Interaction Optimizations

Once the algorithm and the implementation are optimized, interactions with the OS are apt to become the next bottleneck. Specifically, Phoenix frequently uses two OS services: memory allocation and I/O.

Phoenix exerts significant pressure on memory allocators due to its large memory footprint as well as its peculiar allocation pattern. First, Phoenix generates a significant amount of intermediate and output data. Even worse, the memory needs per key are usually unpredictable. As for the allocation pattern, in Phoenix, the thread that allocates memory (e.g., a map thread) rarely is also the thread that deallocates the memory (e.g., a reduce thread). This mismatch can incur contention on the per-thread heap locks when multiple threads try to manipulate the same thread heap. We experimented with a sizable number of memory allocators to compare their performance. However, at high thread count, we noticed that the parallel allocator performance was limited by the scalability of the sbrk() system call.

2.1.3.5. Optimized Phoenix Performance

Figure 2.6(a) summarizes the scalability results for the optimized runtime. Specifically, workloads matrix multiply, kmeans, pca, and string match scaled up to 256 threads. However, some of the workloads still did not scale particularly well. We discuss their bottlenecks in detail in Section V, but in the remainder of this section we first focus on the optimizations that turned out to be successful.



(a) Application speedup with the optimized version of Phoenix.

(b) Relative speedup over the original Phoenix system.

Fig. 2.6 Performance improvements summary [13]

Figure 2.6(b) measures the relative speedup of the new runtime over the original. It essentially compares Figure 2.6(a) and Figure 2.5 using the same dataset. In the figure, the top and bottom of each vertical bar denotes the maximum and minimum performance improvement achieved at a particular thread count, respectively; the horizontal line that connects those bars represent the harmonic means of speedups obtained from the entire workload. The optimized runtime led to improvements across all thread counts. For less than 64 threads (single chip), the average improvement was 1.5×, and the variation across applications was rather small (maximum of 2.8×). For largescale, NUMA configurations with 128 or 256 threads, the optimizations were significantly more effective, reaching 19× in maximum and 2.53× on average. [13]

2.2. Other parallelization Algorithms

2.2.1. FREERIDE

Agrawal's group at Ohio State has developed a middleware system for cluster-based data-intensive processing [15], which shared many similarities with the mapreduce framework. However, there are some subtle but important differences in the API offered by these two systems. One, FREERIDE allows developers to explicitly declare a reduction object, and perform updates to its elements directly, while in Hadoop/map-reduce, reduction object is implicit and not exposed to the application programmer. While explicit reduction object can help with performance gains, there is also a possibility of incorrect usage by application developer. Other important distinction is that, in Hadoop/map-reduce, all the data elements are processed in the map step and the intermediate results are then combined in the reduce step, where as, in FREERIDE, both map and reduce steps are combined into a single step where each data element is processed and reduced before next data element is processed. This choice of design avoids the overhead due to sorting, grouping, and shuffling, which can be significant costs in a map-reduce implementation.

The FREERIDE system was motivated by the difficulties in implementing and performance tuning parallel versions of data mining algorithms. FREERIDE is based upon the observation that parallel versions of several well-known data mining techniques share a relatively similar structure, which is that of a *generalized reduction*. Though the *map-reduce* paradigm is built on a similar observation, it should be noted that the first work on FREERIDE was published in 2001, prior to the map-reduce paper by Dean and Ghemawat in 2004.

The interface exploits the similarity among parallel versions of several data mining algorithms. The following functions need to be written by the application developer using the middleware.

Reduction: The data instances owned by a processor and belonging to the subset specified are read. A reduction function specifies how, after processing one data instance, a *reduction object* (initially declared by the programmer), is updated. The result of this processing must be independent of the order in which data instances are processed on each processor. The order in which data instances are read from the disks is determined by the runtime system.

ProcessNextIteration: This is an optional function that can be implemented by an application programmer. Typically, this function should include a program logic specific to an application that would control the number of iterations for which the application should run. This adds more flexibility to the programmer.

Finalize: After final results from multiple nodes are combined into a single reduction object, the application programmer can read and perform a final manipulation on the reduction object to summarize the results specific to an application.

Throughout the execution of the application, the reduction object is maintained in main memory. After every iteration of processing all data instances, the results from multiple threads in a single node are combined locally depending on the shared memory technique chosen by the application developer. After local combination, the results produced by all nodes in a cluster are combined again to form the final result, which is the global combination phase. The global combination phase can be achieved by a simple all-to-one reduce algorithm. If the size of the reduction object is large, both local and global combination phases perform a parallel merge to speed up the process. The local combination and the communication involved in global combination phase are handled internally by the middleware and transparent to the application programmer.

Fig. 2.7 further illustrates the distinction in the processing structure enabled by FREERIDE and map-reduce. The function Reduce is an associative and commutative function. Thus, the iterations of the for-each loop can be performed in any order. The data-structure *RObj* is referred to as the reduction object.

```
{* Outer Sequential Loop *}
While() {
    {* Reduction Loop *}
    Foreach(element e) {
        (i, val)    =   Process(e) ;
        RObj(i)     =   Reduce(RObj(i),val) ;
    }
    Global Reduction to Combine RObj
}
```

```
{* Outer Sequential Loop *}
While() {
    {* Reduction Loop *}
    Foreach(element e) {
        (i, val)    =   Process(e) ;
    }
    Sort (i,val) pairs using i
    Reduce to compute each RObj(i)
}
```

Fig. 2.7 Processing Structure: FREERIDE (left) and Map-Reduce (right) [16]

2.2.1.1. Case Studies

K-means Clustering: The first application described is the k-means clustering, which is one of the commonly used data mining algorithms. This was one of the earliest algorithms implemented in FREERIDE. The clustering problem is as follows. We consider transactions or data instances as representing points in a high-dimensional space. Proximity within this space is used as the criterion for classifying the points into clusters. Four steps in the sequential version of k-means clustering algorithm are as follows:

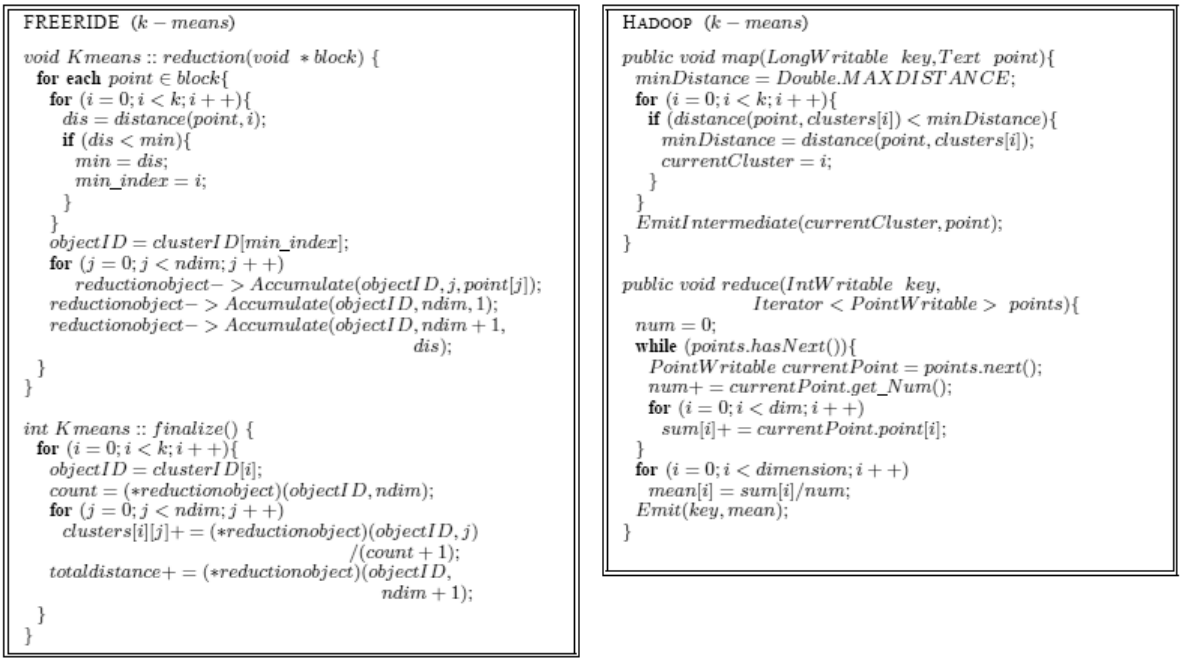
1) start with k given centers for clusters;
2) scan the data instances, for each data instance (point), find the center closest to it and assign this point to the corresponding cluster,
3) determine the k centroids from the points assigned to the corresponding centers, and
4) repeat this process until the assignment of points to clusters does not change.

Fig. 2.8 gives the psedo-code of k-means using FREERIDE API. In FREERIDE, first, the programmer is responsible for creating and initializing the *reductionobject*. The k cluster centroids are initialized as well. For k-means, the *reductionobject* is an array with k×(ndim + 2) elements, where ndim is the number of dimensions for the coordinate space. Each cluster center occupies (ndim+2) consecutive elements in the reductionobject array. Among the (ndim+2) elements for each cluster, the first ndim elements are used to store the sum of data points assigned to each cluster. The other two elements represents the number of data points belonging to each cluster and the accumulated overall *distances*.

In each iteration, the *reduction* operation reads a block of data points, and for each point, it will compute the distance between the point and the k cluster centers and find the closest. Then, we can obtain the *objectID* in the *reductionobject* for the closest cluster and update the particular elements correspondingly in the *reductionobject*. A *global reduction* phase is also required to merge the results from all nodes after the *reduction* operation is applied on all data points.

Fig. 2.8 also gives the pseudo-code of k-means implemented using Hadoop API. The input datasets are partitioned into a number of splits first. Then, the *Map* tasks will be launched. In each iteration, the *Map* task reads the existing cluster centroids from files and then processes a split of the input data points using the *map* function. For each point, *map* finds the closest cluster center, assigns the point to it, and then emits the cluster center id as the key and the point as the value. The *Reduce* task collects all the points with the same cluster center id which is the key, and computes their mean as the new centroid of the

cluster. Finally, the output of *Reduce* tasks will be written into files and taken as new cluster centers by next iteration.



```
FREERIDE (k − means)

void Kmeans :: reduction(void * block) {
  for each point ∈ block{
    for (i = 0; i < k; i++){
      dis = distance(point, i);
      if (dis < min){
        min = dis;
        min_index = i;
      }
    }
    objectID = clusterID[min_index];
    for (j = 0; j < ndim; j++)
        reductionobject− > Accumulate(objectID, j, point[j]);
    reductionobject− > Accumulate(objectID, ndim, 1);
    reductionobject− > Accumulate(objectID, ndim + 1,
                                  dis);
  }
}

int Kmeans :: finalize() {
  for (i = 0; i < k; i++){
    objectID = clusterID[i];
    count = (*reductionobject)(objectID, ndim);
    for (j = 0; j < ndim; j++)
      clusters[i][j]+ = (*reductionobject)(objectID, j)
                                  /(count + 1);
    totaldistance+ = (*reductionobject)(objectID,
                                  ndim + 1);
  }
}
```

```
HADOOP (k − means)

public void map(LongWritable key,Text point){
  minDistance = Double.MAXDISTANCE;
  for (i = 0; i < k; i++){
    if (distance(point, clusters[i]) < minDistance){
      minDistance = distance(point, clusters[i]);
      currentCluster = i;
    }
  }
  EmitIntermediate(currentCluster, point);
}

public void reduce(IntWritable key,
                   Iterator < PointWritable > points){
  num = 0;
  while (points.hasNext()){
    PointWritable currentPoint = points.next();
    num+ = currentPoint.get_Num();
    for (i = 0; i < dim; i++)
      sum[i]+ = currentPoint.point[i];
  }
  for (i = 0; i < dimension; i++)
    mean[i] = sum[i]/num;
  Emit(key, mean);
}
```

Fig. 2.8 Pseudo-code for k-means using: FREERIDE (left) and Hadoop (right) [16]

The *combine* function is optional in the Hadoop API, but to improve the performance, we implemented this function. It is similar to the *reduce* function. It just accumulates the coordinates of data points belonging to the same cluster without computing the mean. The k cluster centers are initialized and written into files and then copied to the HDFS before we run the program. Since k-means is iterative, a sequence of iterations is represented by the same number of *map/reduce* jobs. But due to the constraints of the Hadoop implementation, the configuration and initialization of *map/reduce* tasks for each job needs to occur every time a job is submitted. Hadoop also provides the *run* function, where the programmer can control the number of iterations to run k-means.

We can compare the codes shown and understand the similarities and differences between the two systems. In FREERIDE, the *reductionobject* is used to store the sum of the data points belonging to the same cluster center. It has to be created and initialized first. Then, for each point, the *reductionobject* is updated in the *reduction* operation, after we know which cluster it belongs to. When all data points are processed, the *finalize* operation will compute the new center vector for each cluster. In Hadoop, *map* computes the closest cluster center id for each point and emits it with the point value. The *reduce* function will gather the data points with the same cluster center id. So, to summarize, FREERIDE adds the coordinates of points belonging to each cluster point-by-point, and computes the mean after all points are read. Whereas, Hadoop computes the cluster center id point-by-point, without performing the sum. It performs the accumulation and computes the mean in the *Reduce* Phase. Besides, the *intermediate* pairs produced by the Map tasks are written into files and read by the Reduce tasks through HTTP requests. One observation we can make is as follows. In FREERIDE, *reductionobject*, which is allocated in memory, is used to communicate between the *reduction* and the *finalize* functions. In comparison, HDFS is used to connect the *Map* and the *Reduce* functions in Hadoop. So, Hadoop requires extra sorting, grouping, and shuffling for this purpose.

Word Count: The second application we use is the word-count program, which is a simple application, often used to describe the map-reduce API.

Fig. 2.9 shows the pseudo-code for this application using FREERIDE API. This application does not naturally fit into the FREERIDE framework. This is because we do not know the exact size of the *reductionobject*, as it depends on the number of distinct words in the dataset. So, the *reductionobject* for this application is allocated dynamically id. We used a word map to store the string value of each unique word and its id. The *reduction* operation takes a block of words, and for each word, searches for it first in the word map. If we can find the word, then retrieve its id from the word map, and update its frequency. Otherwise, this is a new word, and we need to insert one entry before updating the reduction object. A *global reduction* phase is also needed to merge the results from all nodes.

Fig. 2.9 gives the pseudo-code for this application using Hadoop API, which is smaller in code size. The *map* function takes a line of words and for each word, emits the word and its associated frequency which is 1 in all cases. The *reduce* function gathers all the counts for each unique word, and compute the sums before the final output.

```
FREERIDE (wordcount)

void Wordcount :: reduction(void * block) {
  for each word ∈ block{
    map < string, int >:: iteratoriter = wordmap.begin();
    iter = wordmap.find(word);
    if (iter! = wordmap.end() ){
      wordID = iter− > second;
      reductionobject− > Accumulate(wordID, 1, 1);
    }
  else {
      newID = reductionobject− > alloc(2);
      wordmap.insert(pair < string, int > (word, newID));
      reductionobject− > Accumulate(newID, 0, newID);
      reductionobject− > Accumulate(newID, 1, 1);
    }
  }
}

int Wordcount :: finalize() {
  size = wordmap.size();
  for (i = 0; i < size; i + +){
    id = (∗reductionobject)(i, 0);
    count = (∗reductionobject)(i, 1);
    output(get_word(id), count);
  }
  return 0;
}
```

```
HADOOP (wordcount)

public void map(LongWritable key, Text words){
  for each word ∈ words
    EmitIntermediate(word, "1");
}

public void reduce(Text key,
                   Iterator < IntWritable > values){
  sum = 0;
  while (values.hasNext()){
    sum+ = values.next.get();
  }
  Emit(key, newIntWritable(sum));
}
```

Fig. 2.9 Pseudo-code for wordcount using: FREERIDE (left) and Hadoop (right) [16]

Experimental Evaluation

Since the main focus of this study is on data mining applications, three popular data mining algorithms were chosen. They are, kmeans clustering, apriori associating mining, and k-nearest neighbor search. K-means clustering was described earlier. Apriori is a well accepted algorithm for *association mining*, which also forms the basis for many newer algorithms. Association rule mining is the process of analyzing a set of transactions to extract *association rules* and is a commonly used and well-studied data mining problem. Given a set of transactions, each of them being a set of items, the problem involves finding subsets of items that appear frequently in these transactions. k-nearest neighbour classifier is based on learning by analogy. The training samples are described by an n-dimensional numeric space. Given an unknown sample, the k-nearest neighbour classifier searches the pattern space for k training samples that are closest, using the Euclidean distance, to the unknown sample. Among these three algorithms, k-means and apriori both require multiple passes over data, whereas kNN is a single pass algorithm.

To complete the experimental study, a fourth application was included, which is quite different from the other three applications. Word-count is a simple search application, and can be viewed as a representative of the class of applications that motivated the development of the mapreduce framework. In comparison, FREERIDE was motivated by data mining algorithms, including the three applications used in this study.

Experiments were conducted on a cluster of multicore machines. Each node in the cluster is an Intel xeon CPU E5345, comprising two quad-core CPUs. Each core has a clock frequency of 2.33GHz and each node has a 6 GB main memory. The nodes in the cluster are connected by Infiniband. These applications were excecuted with FREERIDE and Hadoop, on 4, 8, and 16 nodes of the cluster.

Results showed that for the three data mining applications, FREERIDE outperforms Hadoop significantly. The differences in performances ranged from a factor of 2.5 to about 20. We can also see that the scalability on increasing number of nodes is almost linear with FREERIDE. With Hadoop, speedups in going from 4 nodes to 16 is about 2.2 for k-means and wordcount, and only about 1.1 for apriori. Overall, Hadoop has high overhead for data mining applications with a significant amount of computations.

The results are very different for word-count. On 4 nodes, Hadoop is faster by a factor of about 2. With increasing number of nodes, FREERIDE is more competitive, though still slower by about 10% on 16 nodes. There are two possible reasons for this performance limitation. First, for such an algorithm, FREERIDE's shared memory techniques are not most appropriate. Second, the specialized file system in Hadoop may be improving the performance for this application where there is very little computation.

2.2.2. MATE

MATE system has been developed on top of Phoenix by Wei Jiang, Vignesh T. Ravi, and Gagan Agrawal [17]. The key distinctive aspect of this API is that it allows the programmers to explicitly declare a *reduction object*. The *map* and *reduce* functions are replaced by a single *reduction* function that updates such a reduction object, as opposed to emitting (key, value) pairs, and then reducing the values that have the same key. Race conditions can be avoided by replicating the reduction object. A *combination* function combines different copies of the reduction object to create the final results. The runtime system automatically maps the *Reduction/Combination* tasks to threads that are spawned on processor cores. Also, the runtime scheduler dynamically partitions the dataset and assigns splits to processor cores. The MATE system's generalized reduction based API is motivated by FREERIDE.

The API supported by MATE (and FREERIDE) exploits the similarity among parallel versions of several data mining and scientific data processing algorithms. The following are some important functions, which need to be written by the application developer.

Reduction: A reduction function specifies how, after processing one data instance, a *reduction object* (initially declared by the programmer), is updated. The result of this processing must be independent of the order in which data instances are processed on each processor. The order in which data instances are processed is determined by the runtime system.

Combination: In this function, final results from multiple copies of a reduction object are combined into a single reduction object. A user can choose from one of the several common combination functions already implemented in the system, or could provide one of their own.

2.2.1. Execution Overview

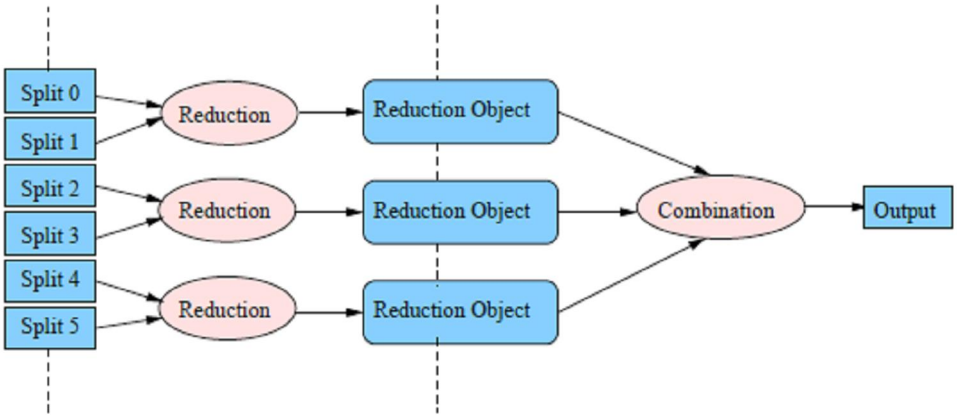


Fig. 2.10 One-Stage Execution Overview [17]

Fig. 2.10 gives an overview of the execution of an application. Some of the key aspects are as follows:

Scheduler Initialization: To initialize the scheduler, the programmer needs to specify all required data pointers and functions. Particularly, the programmer should pass the input data pointers and implement the *splitter*, *reduction*, and *combination* functions. Also, the programmer has to declare a *reduction object*.

Reduction: After initialization, the scheduler checks the availability of processor cores, and for each core, it creates one worker thread. Before the *Reduction* phase starts, the *splitter* function is used to partition the data into equal-sized splits. Each worker thread will then invoke the *splitter* to retrieve one split and process it with the *reduction* function.

To avoid load imbalance, *Reduction* tasks are assigned to worker threads dynamically instead of statically partitioning the data. Using this approach, each thread worker will repeatedly retrieve one split of the data and then process it as specified in the *reduction* function until no more splits are available. The split size, however, must be set appropriately to balance the lower overheads (few larger splits) and the load balancing (more smaller splits). To make use of temporary locality, by default, the runtime adjusts the split size such that the input data for a *Reduction* task can fit in the L1 cache. The programmer can also vary this parameter to achieve better performance given the knowledge of a specific application.

The *reduction object* is updated correspondingly in the *reduction* function as specified by the user code. Each thread will repeat this step until all data instances have been processed. At this point, the *Reduction* stage is over. The scheduler must wait for all *reduction* workers to finish before initializing the *Combination* stage.

Combination: The processing structure of Generalized Reduction enables us to develop general but efficient shared memory parallelization schemes. Consider again the loop in Fig. 2.7. The main correctness challenge in parallelizing a loop like this on a shared-memory machine arises because of possible race conditions (or *access conflicts*) when multiple processors update the same element of the reduction object. Two obvious approaches for avoiding race conditions are: *full replication* and *full locking*. In the full replication approach, each processor can update its own reduction object and these copies are merged together later. In the full locking approach, one lock or latch is associated with each aggregated value.

Currently, the *full replication* approach is used to implement the parallelization. So a *Combination* phase is required to merge the *reduction-object*s of multiple threads when all data is processed.

In the *Combination* phase, the scheduler will spawn a combination-worker thread to merge the *reduction object* copies of all *Reduction* threads. The result is a single copy of the final reduction object after every iteration. If the application involves multiple iterations, this object is stored by the scheduler and can be accessed by the future stages.

Finalize: This phase occurs after all other stages of the computation have been performed, i.e., each *Combination* stage has its own copy of the *reduction object*, which is combined from multiple threads. The programmer is then able to perform a manipulation on the *reduction-object*s to summarize the results specific to an application. The scheduler also needs to free the allocated space in this phase, but this is transparent to the programmer.

Buffer Management: The runtime system handles two types of temporary buffers. One is the *reduction-object buffer* allocated for each thread to do its own computation over different splits. The other is the *combination buffer* created for storing the intermediate output results of each stage. The *combination buffer* is of type *reduction object* in this implementation. The buffers are initially sized to a default value and then grown dynamically if needed.

Fault Tolerance: Fault detection and recovery have been an important aspect of map-reduce implementations, as data intensive applications can be very long running, and/or may need to be executed on a large number of nodes or cores. The current support for fault-tolerance is based on the Phoenix system. It supports fault-recovery for the *Reduction* tasks. If a *Reduction* worker does not finish within a reasonable time limit, the scheduler assumes that a failure of this task has occurred. The failed task is then re-executed by the runtime. In this implementation, separate buffers are allocated for the new task to avoid data access conflicts.

Experimental Evaluation

Since the main focus of this study [17] is on data mining applications, three popular data mining algorithms were chosen. They are kmeans clustering, apriori association mining, and principal components analysis (PCA). Apriori and k-means were described in the FREERIDE section. PCA is a popular dimensionality reduction method that was developed by Pearson in 1901. Its goal is to compute the mean vector and the covariance matrix for a set of data points that are represented by a matrix.

The experiments were conducted on two distinct multicore machines. One system uses Intel Xeon CPU E5345, comprising two quad-core CPUs (8 cores in all). Each core has a clock frequency of 2.33GHz and the system has a 6 GB main memory. The other system uses AMD Opteron Processor 8350 with 4 quad-core CPUs (16 cores in all). Each core has a clock frequency of 1.00GHz and the system has a 16 GB main memory.

The aforementioned three applications were executed with both MATE and Phoenix systems on the two multi-core machines. Since MATE uses the same scheduler as Phoenix, the main performance difference is likely due to the different processing structure.

Results for k-means showed that MATE outperforms Phoenix on both the machines. MATE is almost twice as fast as Phoenix with 8 threads on the 8-core machine, and almost thrice as fast with 16 threads on the 16-core machine. We also see a very good scalability with increasing number of threads or cores with MATE. On the 8-core machine, the speedup with MATE in going from 1 to 8 threads is about 7.8, whereas, Phoenix can only achieve a speedup of 5.4. The result is similar on the 16-core machine: MATE can get a speedup of about 15.0 between 1 and 16 threads, whereas, it is 5.1 for Phoenix. With more

than 100 million points in the k-means dataset, Phoenix was slower because of the high memory requirements associated with creating, storing, and accessing a large number of intermediate pairs. MATE, in comparison, directly accumulates values into the *reduction object*, which is of much smaller size.

PCA does not scale as well with increasing number of cores. This is because some segments of the program are either not parallelized, or do not have regular parallelism. However, we can still see that, with 8 threads on the 8-core machine and 16 threads on the 16-core machine, MATE was twice as fast as Phoenix. Besides the overheads associated with the intermediate pairs, the execution-time breakdown analysis for Phoenix shows that the *Reduce* and *Merge* phases account for a non-trivial fraction of the total time. This overhead is non-existent for the MATE system, because of the use of the *reduction object*.

Finally, the results for apriori showed that for 1 and 2 threads, MATE is around 1.5 times as fast as Phoenix on both the machines. With 4 threads and more, the performance of Phoenix was relatively close to MATE. One of the reasons is that, compared to k-means and PCA, the *Reduce* and *Merge* phases take only a small fraction of time with the apriori dataset. Thus, the dominant amount of time is spent on the *Map* phase, leading to good scalability for Phoenix. For the MATE system, the *reduction object* can be of large size and needs to be grown and reallocated for all threads as the new candidate itemsets are generated during the processing.

Overall, these experiments show that the MATE system has between reasonable to significant speedups for all the three datamining applications.

2.3. Limitations of MapReduce

Solutions to many interesting problems do not require global synchronization. As a result, they can be expressed naturally in MapReduce, since map and reduce tasks run independently and in isolation.

However, there are many examples of algorithms that depend crucially on the existence of shared global state during processing, making them difficult to implement in MapReduce (since the single opportunity for global synchronization in MapReduce is the barrier between the map and reduce phases of processing).

The first example is online learning. Both EM and the gradient-based learning algorithms are instances of what are known as batch learning algorithms. This simply means that the full “batch" of training data is processed before any updates to the model parameters are made. On one hand, this is quite reasonable: updates are not made until the full evidence of the training data has been weighed against the model. An earlier update would seem, in some sense, to be hasty. However, it is generally the case that more frequent updates can lead to more rapid convergence of the model (in terms of number of training instances processed), even if those updates are made by considering less data. Thinking in terms of gradient optimization, online learning algorithms can be understood as computing an approximation of the true gradient, using only a few training instances. Although only an approximation, the gradient computed from a small subset of training instances is often quite reasonable, and the aggregate behaviour of multiple updates tends to even out errors that are made. In the limit, updates can be made after every training instance.

Unfortunately, implementing online learning algorithms in MapReduce is problematic. The model parameters in a learning algorithm can be viewed as shared global state, which must be updated as the model is evaluated against training data. All processes performing the evaluation (presumably the mappers) must have access to this state. In a batch learner, where updates occur in one or more reducers (or, alternatively, in the driver code), synchronization of this resource is enforced by the MapReduce framework. However,

with online learning, these updates must occur after processing smaller numbers of instances. This means that the framework must be altered to support faster processing of smaller datasets, which goes against the design choices of most existing MapReduce implementations. Since MapReduce was specifically optimized for batch operations over large amounts of data, such a style of computation would likely result in inefficient use of resources. In Hadoop, for example, map and reduce tasks have considerable start-up costs. This is acceptable because in most circumstances, this cost is amortized over the processing of many key-value pairs. However, for small datasets, these high start-up costs become intolerable. An alternative is to abandon shared global state and run independent instances of the training algorithm in parallel (on different portions of the data). A final solution is then arrived at by merging individual results. Experiments, however, show that the merged solution is inferior to the output of running the training algorithm on the entire dataset.

A related difficulty occurs when running what are called Monte Carlo simulations, which are used to perform inference in probabilistic models where evaluating or representing the model exactly is impossible. The basic idea is quite simple: samples are drawn from the random variables in the model to simulate its behaviour, and then simple frequency statistics are computed over the samples. This sort of inference is particularly useful when dealing with so-called nonparametric models, which are models whose structure is not specified in advance, but is rather inferred from training data. For an illustration, imagine learning a hidden Markov model, but inferring the number of states, rather than having them specified. Being able to parallelize Monte Carlo simulations would be tremendously valuable, particularly for unsupervised learning applications where they have been found to be far more effective than EM-based learning (which requires specifying the model). Although recent work has shown that the delays in synchronizing sample statistics due to parallel implementations do not necessarily damage the inference, MapReduce offers no natural mechanism for managing the global shared state that would be required for such an implementation.

The problem of global state is sufficiently pervasive that there has been substantial work on solutions. One approach is to build a distributed data store capable of maintaining the global state. However, such a system would need to be highly scalable to be used in conjunction with MapReduce. Google's BigTable, which is a sparse, distributed, persistent multidimensional sorted map built on top of GFS, fits the bill, and has been used in exactly this manner. Amazon's Dynamo, which is a distributed key-value store (with a very different architecture), might also be useful in this respect, although it wasn't originally designed with such an application in mind. Unfortunately, it is unclear if the open-source implementations of these two systems (HBase and Cassandra, respectively) are sufficiently mature to handle the low-latency and high-throughput demands of maintaining global state in the context of massively distributed processing.

2.4. Objectives and Contribution

Our work focuses on processing data sets by evenly distributing workload on a many-core embedded platform. We introduce a MapReduce framework to address the problems of:

1. Providing a scalable solution for data processing on a Many-core system
2. Providing a method for dividing the input and workload among the available cores.

3. Exploiting platform's characteristics in order to provide synchronization and communication between the cores in a Distributed Shared Memory (DSM) environment.

The main contributions of the proposed MapReduce framework are:

1. **HPC on Embedded:** Proves that it is feasible for traditional High Performance Computing (HPC) methods to be used on embedded Many-core systems.
2. **Many-core functionality**: Provides an easy to program data processing method that utilizes Many-core computational power without exposing the programmer to parallel programming.
3. **Evaluation:** Evaluation of the proposed frameworks is conducted through extensive experimentation and explorative results.

Chapter 3: Proposed Methodology

The proposed methodology includes a multi-core NoC as presented by Xiaowen et al. in [18]. The framework runs on this NoC taking advantage of the mechanism explained in the next section.

3.1 The DME

The DME is a dual microcoded controller, which supports distributed shared memory (DSM) functions in NoCs (Network on Chips). A DME is a hardware module that connects to the CPU core, local memory, and the network. Each DME features two mini-processors to be able to concurrently deal with requests from the local core and remote cores via the network. The execution in the mini-processors is triggered by memory requests. Figure 3.1 (a) shows an example of a DSM based multi-core NoC architecture. The system is composed of 16 Processor – Memory (PM) nodes interconnected via a packet – switched network. The network topology is a mesh, which is the most popular NoC topology today. As shown in figure 3.1 (b), each PM node contains a processor (LEON 3), hardware modules connected to the local bus and a local memory.

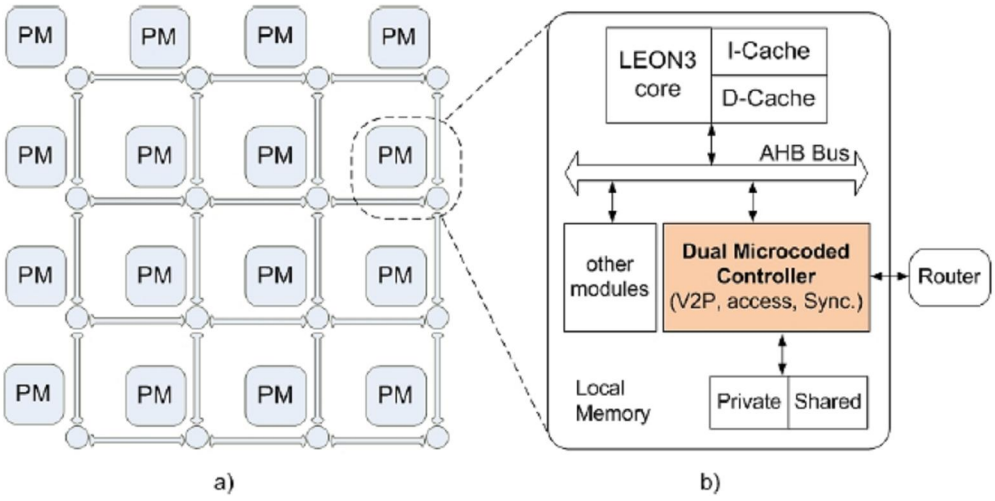


Fig. 3.1 (a) A 16-node mesh multi-core NoC, (b) Processor – Memory node [18]

3.1.1. Architectural Design

As shown in Figure 3.2, the controller contains six main parts, namely, *Core Interface Control Unit* (CICU), *Network Interface Control Unit* (NICU), *Control Store*, *Mini-processor A* and *B*, and the *synchronization supporter*. As the names suggest, the CICU provides a hardware interface to the local core, the NICU a hardware interface to the network. The two mini-processors are the central processing engine, Microprogram is initially stored in the Local Memory and dynamically uploaded into the Control Store on demand during the program execution. The Synchronization Supporter coordinates the two mini-processors to avoid simultaneous accesses to the same memory address and guarantees atomic read-and-modify operations. Both the Local Memory and the control store are dual

ported, port A and B, which connect to the mini-processor A and B, respectively. The functions of each module are detailed as follows:

Core Interface Control Unit: The CICU connects with the core, mini-processor A, the NICU, the Control Store and the Local Memory. Its main functions are: (I) it receives local requests in form of command from the local core and triggers the operation of the mini-processor A accordingly; (II) it uploads the microcode from the Local Memory to the Control Store through port A; (III) it receives results from the mini-processor A; (IV) it accesses the private memory directly using physical addressing if the memory access is private; (V) it sends results back to the local core.

Network Interface Control Unit: The NICU connects the network, the mini-processor B, the CICU, the Control Store and the Local Memory. Its main functions are: (I) it receives remote requests in form of command from the network and triggers the operation of the mini-processor B accordingly; (II) it also can upload the microcode from the Local Memory to the Control Store through port B; (III) it sends remote requests from the mini-processor A or B to remote destination nodes in format of message via the network; (IV) it receives the format of message from remote destination nodes via the network and forwards them to the mini-processor A or B.

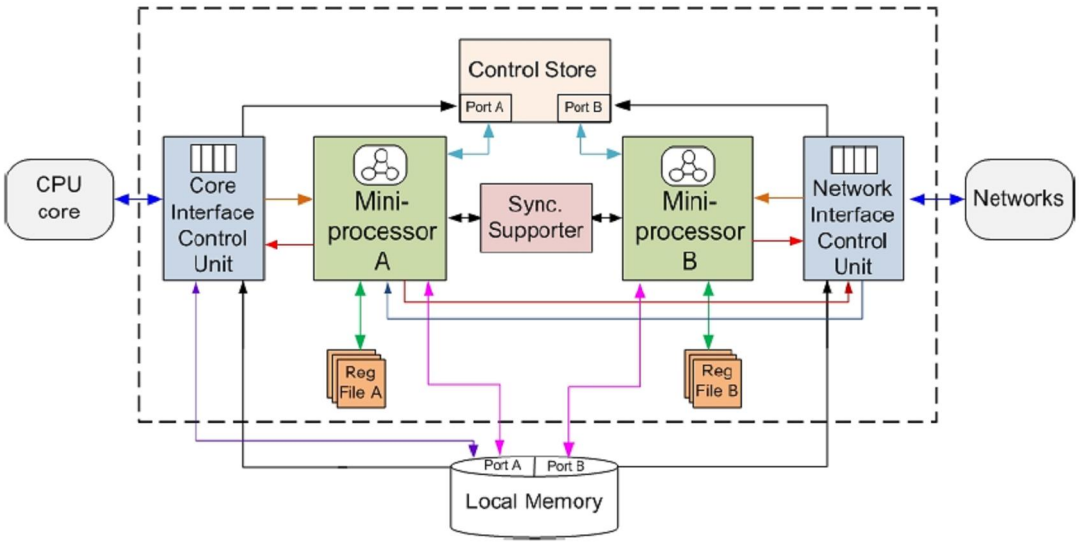


Fig. 3.2 Architecture of the Dual Microcoded Controller [18]

Mini-processor A: The mini-processor A connects with the CICU, the register file A, the Synchronization Supporter, the Control Store, and the Local Memory. Its operation is triggered by a command from the local core. It executes microcode from the Control Store through port A, uses register file A for temporary data storage, and accesses the Local Memory through port A.

Mini-processor B: The mini-processor B connects with the NICU, the register file B, the Synchronization Supporter, the Control Store, and the Local Memory. Its operation is triggered by a command from remote cores via the network. It executes microcode from the Control Store through port B, uses register file B for temporary data storage and accesses the Local Memory through port B.

The two mini-processors feature a five-stage pipeline and four function units: *Load/Store Unit (LSU), Adder Unit (AU), Condition Unit* and *Message Passing Unit*

(MPU), to provide operations of memory access, addition, conditional branch and message-passing.

Synchronization Supporter: The Synchronization Supporter, which connects with the mini- processor A and B is a hardware module to support atomic read- and-modify operation. This is necessary when two synchronization requests try to access the same lock at the same time.

Control Store: The Control Store, which connects with the CICU, the NICU, the mini-processor A and B and the Local Memory, is a local storage for microcode; like an instruction cache. It dynamically uploads microcode from the Local Memory. It feeds microcode to the mini-processor A through port A, and the mini-processor B through port B. This uploading and feeding is controlled by the CICU for commands from the local core and the NICU for commands from remote cores via the network.

3.1.2. Operation Mechanism

For the DMC, the execution of the mini-processors is triggered by requests (in form of command) from the local and remote cores. This is called *command-triggered microcode execution.* As shown in Fig. 3.3, a command is related to a certain function, which is implemented by a microcode. A microcode is a sequence of microinstructions with an **end** microoperation at the end. A microprogram is a set of microcodes. Fig. 3.4 shows how the DMC works (Microprogram is initially stored in the Local Memory). This procedure is iterated over the entire execution of the system.

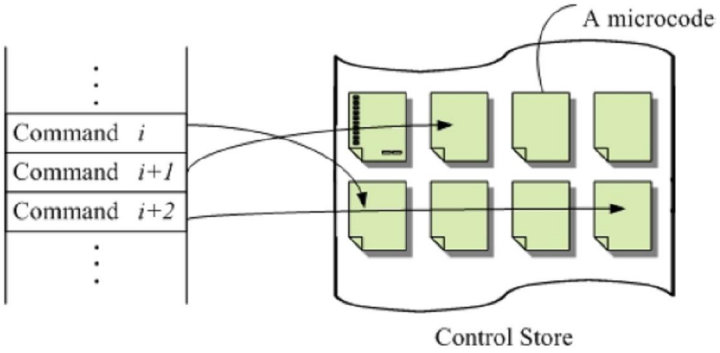


Fig. 3.3 One command triggers a microcode [18]

3.1.3. DSM Functions

Using microcode, Xiaowen et al. [18] implemented DSM functions, namely: V2P address translation, shared memory access and synchronization.

Virtual-to-Physical (V2P) Address Translation: To maintain a Distributed Shared Memory environment, each time the request (in form of command) from the local core or a remote core comes, the Virtual-to-Physical address translation is always performed at first to obtain the physical address. And then, the target microcode related to this command will be executed.

Shared Memory Access: Shared memory access is implemented by microcode. We categorize it into two types: (1) Local shared access: (2) Remote shared access. Because shared memory access uses logical addressing, it implies a V2P translation overhead.

Synchronization: The Synchronization Supporter provides underlying hardware support for synchronization. It works with a pair of special microoperations (ll and sc) to guarantee atomic operation. Based on them, various synchronization primitives can be built. The primitive implemented was a *test-and-set()* lock.

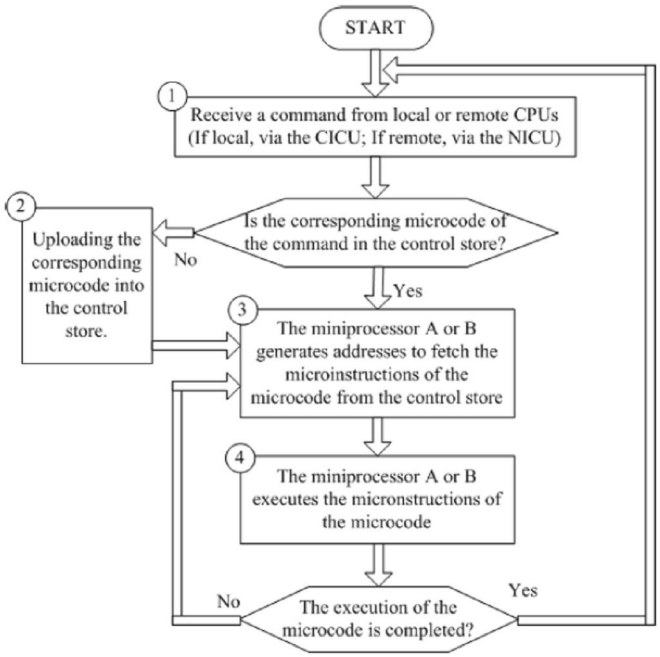


Fig. 3.4 Work flow of the DMC [18]

3.1.4. Distributed Shared Memory Space

Memories are distributed in each node and tightly integrated with processors. All local memories can logically form a single global memory address space. However, we do not treat all memories as shared.

As illustrated in Figure 3.1 and Figure 3.5, the local memory is partitioned into two parts: private and shared. And two addressing schemes are introduced: physical addressing and logic (virtual) addressing.

The private memory can only be accessed by the local processor and it's physical. All of shared memories are visible to all nodes and organized as a Distributed Shared Memory (DSM) and are virtual.

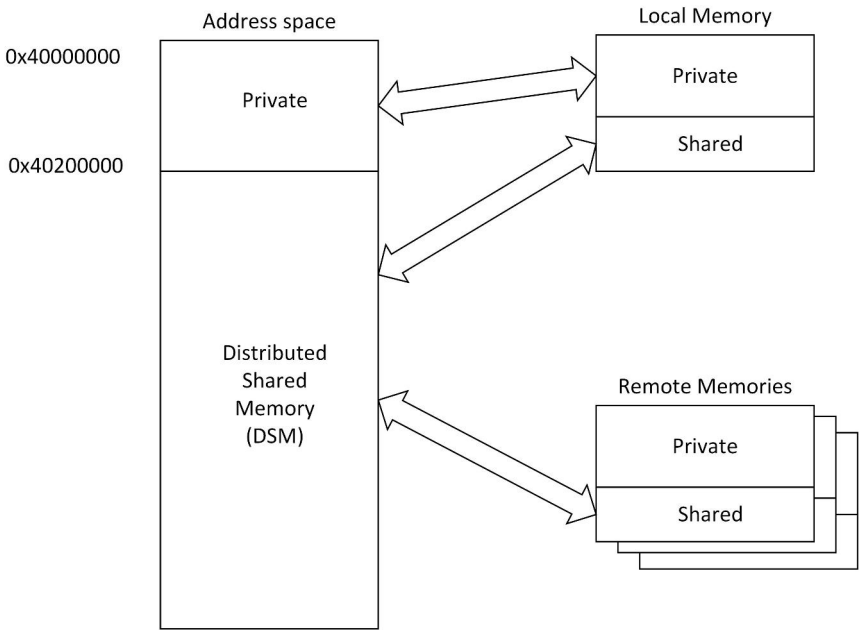


Fig. 3.5 DSM Organization

The boundary address between the private memory space and the shared memory space is 0x40200000.

Memory Space	Information
0x40000000 ~ 0x401FFFFF	Private Memory
0x40200000 ~ 0x4021FFFF	Shared Memory #0
0x40220000 ~ 0x4023FFFF	Shared Memory #1
0x40240000 ~ 0x4025FFFF	Shared Memory #2
…	…

Table 3.1. DSM Address Space

3.2 The proposed MapReduce Framework

The proposed MapReduce framework is loosely based on the original phoenix implementation. The framework runs on a DSM based multi-core NoC that utilizes the DMC architecture described in the section above and is implemented in C.

3.2.1. Proposed Framework Overview

The nodes in the presented framework are distinguished in two categories. There is a master node and worker nodes. The functionality that distinguishes the two types of nodes is summarized below.

Master node: It is the first node in any mesh. It handles the synchronization needed between the different execution phases of our framework. The master node ensures that all map tasks have ended and all the intermediate data have been written to shared memory before the reduce phase can begin.

Worker nodes: They are the frameworks processing nodes. They initially apply the map function on the given data splits to create intermediate data. Then, they write these data to the shared memory. After map phase is over, workers apply the reduce function on the

intermediate data to create the final data (also located in the shared memory). The workers run the same code, but each worker has a unique identification number which depends on the location of the node on the mesh.

Figure 3.6 shows an example of the topology proposed. In this figure we see in a 3x3 mesh of nodes, which has the characteristics of the NoC presented in section 3.1. The first node, which occupies the position (0,0) is the master node. The other 8 nodes are worker nodes, and are given IDs using the formula: $ID = x * y\_\max + y - 1$, where y_max stands for the number of nodes in the y direction.

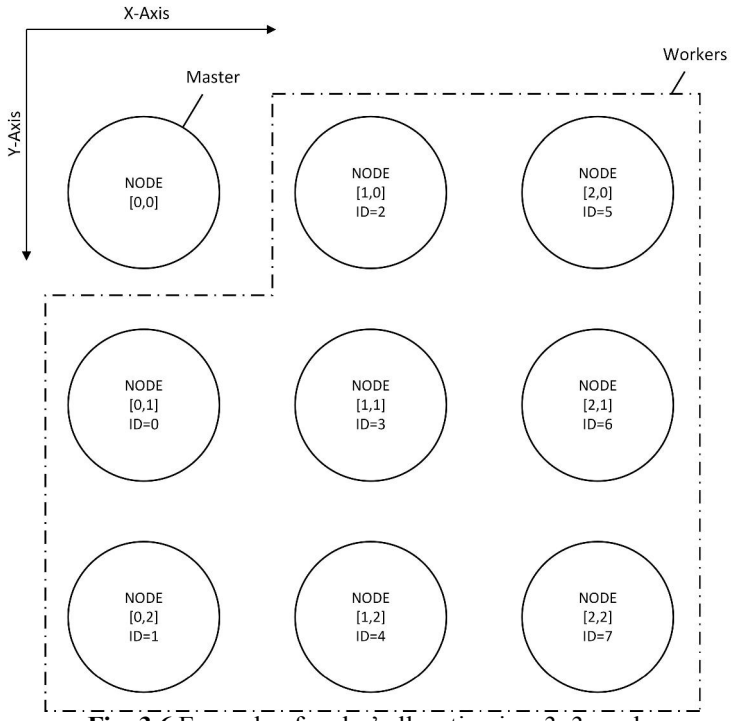


Fig. 3.6 Example of nodes' allocation in a 3x3 mesh

An overview of the proposed methodology framework is presented in figure 3.7. Once the execution begins, all nodes work concurrently. Master node initializes all locks to 0 and then acquires all the worker nodes' locks. Meanwhile, worker nodes initialize their local combiners' arrays and acquire the Map lock (one of master node's locks). By doing this, master node locally polls his lock, until map phase is over and all worker nodes have unlocked the Map lock. During Map phase, worker nodes consecutively take a portion of the input data and process it using the map function. They repeat this operation until all input data have been processed. The intermediate data pairs produced are stored locally, aggregated by the combiner. When all map tasks are over in each node, the node stores its intermediate data to its part of the shared memory, unlocks the Map lock and start polling its lock. By the time all nodes have unlocked Map lock, the master node stops polling. That concludes the Map phase. To enter Reduce phase, master node unlocks all worker nodes' locks. Then worker nodes acquire the Reduce lock (the other master node's lock), select a key from the intermediate data and use the Reduce function on the values associated with that key. They repeat this operation until all keys have been processed. The final data pairs produced are stored in the shared memory. When all reduce tasks are over in each node, the node unlocks the Reduce lock exits. The master node stops polling, prints the results and then exits as well.

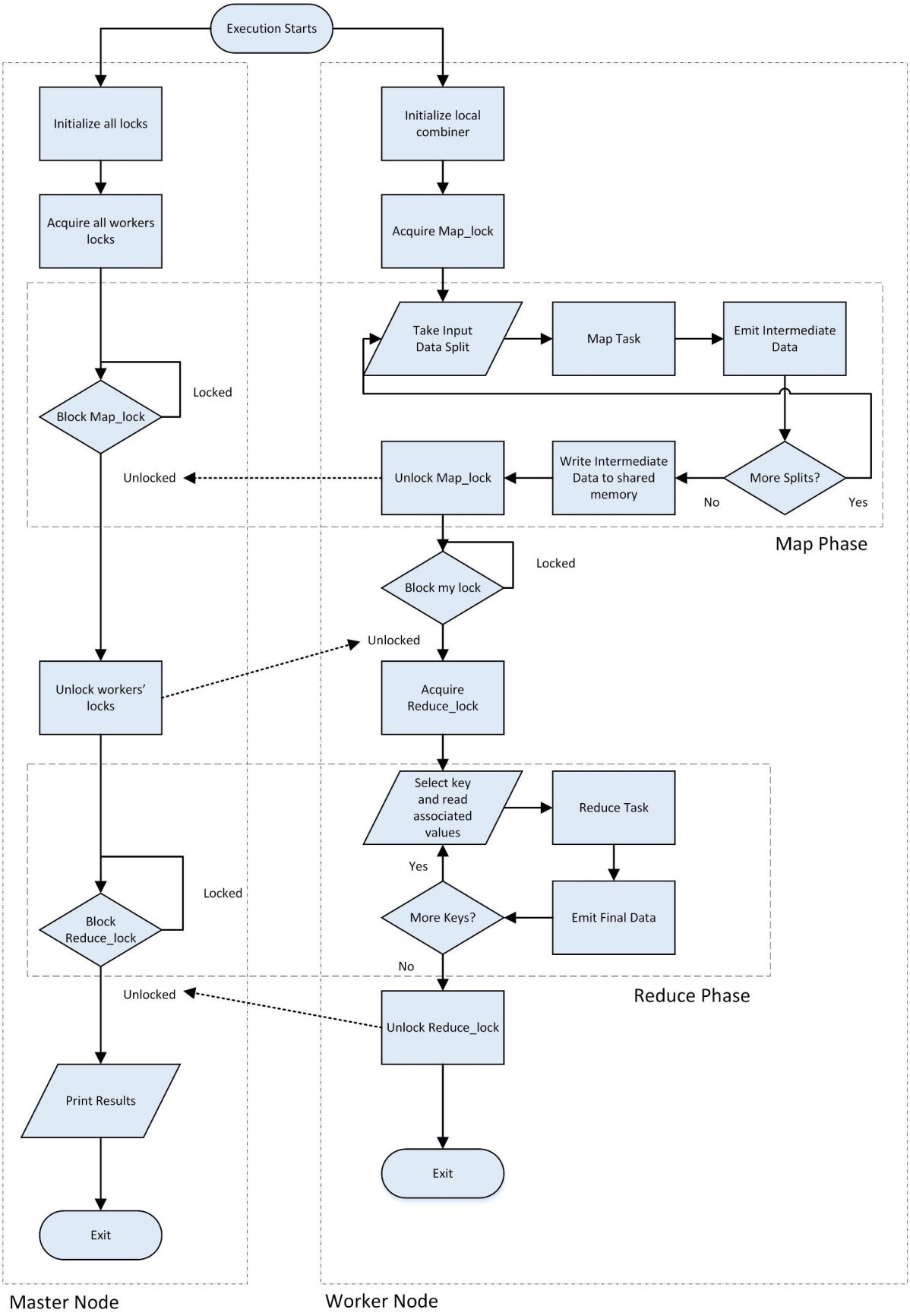


Fig. 3.7 Overall flow of the proposed methodology

Figure 3.8 shows an overview of the code used by the master and the worker nodes. As presented in fig. 3.8 (a), the master node, after a short initialization, tries to acquire the map_lock. When the map_lock is unlocked by all workers, master node unlocks all worker nodes, and then tries to acquire the red_lock. After it is unlocked, the merger function displays the results. The worker node, after an initialization phase, begins the map phase. Each map task, takes a split of the data with size equal to the unit_size provided by the user, until all data have been taken. Then, each worker writes its combiner data to the shared memory unlocks map_lock and tries to acquire its lock. When master unlocks all nodes, they acquire red_lock and begin the reduce phase. Each worker takes a key, according to their id number, emits the result given by the reduce function and chooses the next key, until all keys are reduced. Finally, unlocks the red_lock and quits. The functions and data structures used are presented in the next sections.

```
main()
{
    Initialize();

    lock(map_lock);

    for (k=0; k<NUM_WORKERS; k++)
    {
        unlock(locks[k]);
    }

    lock(red_lock);

    merger();
}
```

(a)

```
main()
{
    Initialize();

    lock(map_lock);

    //Map Phase
    for (i=CPU_ID*unit_size; i<INPUT_SIZE; i+=NUM_WORKERS*unit_size)
    {
        map(i, (i+unit_size)<INPUT_SIZE ? unit_size : INPUT_SIZE-i);
    }

    Write_inter_data();

    unlock(map_lock);

    block(my_lock);

    lock(red_lock);

    //Reduce Phase
    for (i=CPU_ID; i<NUM_KEYS; i+=NUM_WORKERS)
    {
        emit(i,reduce(i));
    }

    unlock(red_lock);
}
```

(b)

Fig. 3.8 Overview of the code used by the master (a) and the worker (b) nodes

3.2.2. Synchronization

We relied on locks for the synchronization of the Map and Reduce phases. Two types of locks were used, a *test-and-set* and a *fetch-and-increase* lock. Both locks were hardware supported by the DME.

The test-and-set lock checks if the content of a specified memory address is set to 0 and if it is set, it acquires the lock by setting it to 1. If it is already set to 1 then the code polls locally until the lock is set to 0. The microcode used for this lock is presented in figure 3.9 (a) and could be called by the *block(locks_address)* command.

The fetch-and-increase lock simply increases the content of a specified address by 1. The microcode used for this lock is presented in figure 3.9 (b) and could be called by the *fetch_and_set(locks_address)* command.

Finally, a microcoded unlock function was implemented to unlock both locks. The unlock function decreased the content of the lock's address by 1. The microcode used for this lock is presented in figure 3.9 (c) and could be called by the *unlock(locks_address)* command.

In order to work, the contents of the memory addresses used as locks must be initialized to 0, before being used.

```
;Optimized TEST and SET lock
            ll *A6,A0
            nop
            nop
            bneqz A0, FAILED_2
            nop
SUCCESS_2:   sl *A6, 1
            nop
            end 1
FAILED_2:   sl *A6, A0
            nop
            end 2
```

(a)

```
;FETCH and INC lock
            ll *A6,A0
            nop
            nop
            add A0,A0,1
            nop
            sl *A6, A0
            nop
            end 1
```

(b)

```
;Unlock
            ll *A6,A0
            nop
            nop
            sub A0,A0,1
            nop
            sl *A6,A0
            nop
            end 1
```

(c)

Fig. 3.9 Microcode for (a) test-and-set lock, (b) fetch and increase lock and (c) unlock

In the proposed implementation, each worker node has a unique lock located in its shared memory (memory address generated automatically using the formula: $0x40223820 + (0x020000 \cdot ID)$). Those locks are initially obtained by the master before the beginning of the Map Phase. Each node locally polls its lock, until the Master releases it, after the end of Map phase. In this way, we ensure that no individual node will enter the Reduce phase before all map tasks are done.

Master node has two locks located in its shared memory (addresses 0x40203838 and 0x40203840). Both locks are used with the fetch-and-increase mechanism. The first lock, called Map_lock, is fetched and increased by every node at the beginning of the Map phase. The master locally polls, until all nodes have unlocked this lock. In this way, we ensure that every node will finish its map tasks before the Reduce phase can begin. The second lock, called Reduce_lock, is fetched and increased by every node in the beginning of the Reduce phase. Again, the master locally polls, until all nodes have unlocked this lock. When all reduce tasks are done, every node unlocks the Master, which, in its turn, finalizes the framework execution. The timeline of this operation is shown in figure 3.10.

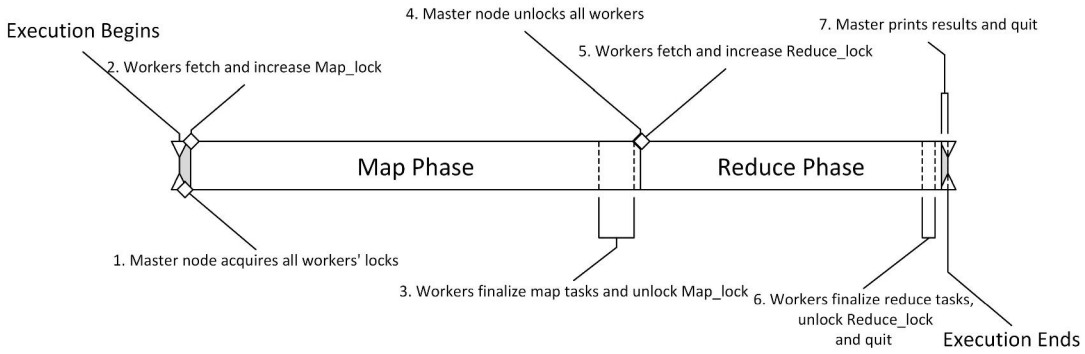


Fig. 3.10 Timeline of the synchronization

3.2.3. Data Storage

There are two main types of data that need to be stored. These are the *intermediate data* and the *final data*.

The *intermediate data* are in form of <key, value> pairs and are the product of map tasks. Aggregated locally by the combiner, intermediate data from each node are written in their shared memory in form of an array. The indexing of this array matches the indexing of the keys. Each value associated with a key will be later accessed by the node that reduces that key. The intermediate data arrays are presented in figure 3.11, which shows the dataflow during the map phase. As we can see each worker node stores a copy of its intermediate data in its shared memory. For example, worker node 1 will have an intermediate data array that begins in memory address 0x40230000.

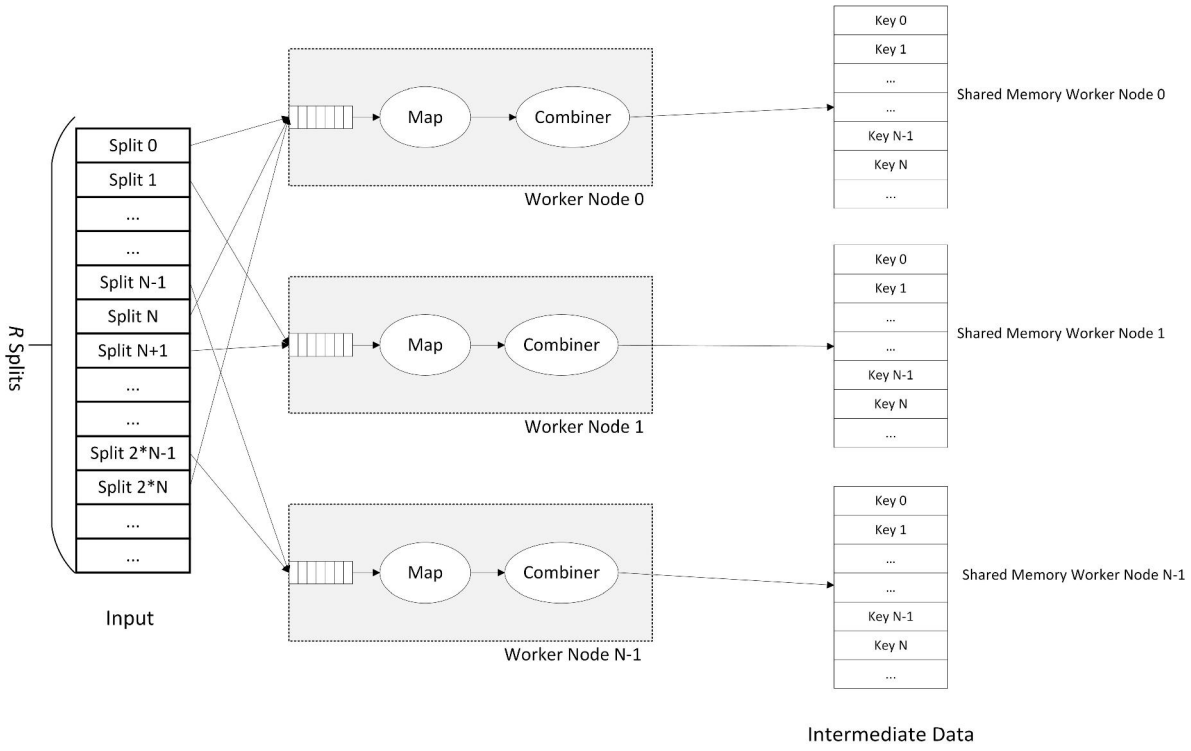


Fig. 3.11 Dataflow during the Map phase

The *final data* are also in form of <key, value> pairs and are the product of reduce tasks. Contrary to the intermediate data, final data are stored in a single array located in the shared memory of the master node. The array begins in memory address 0x40203860 and follows the same indexing scheme as before. The final data array is presented in figure 3.12.

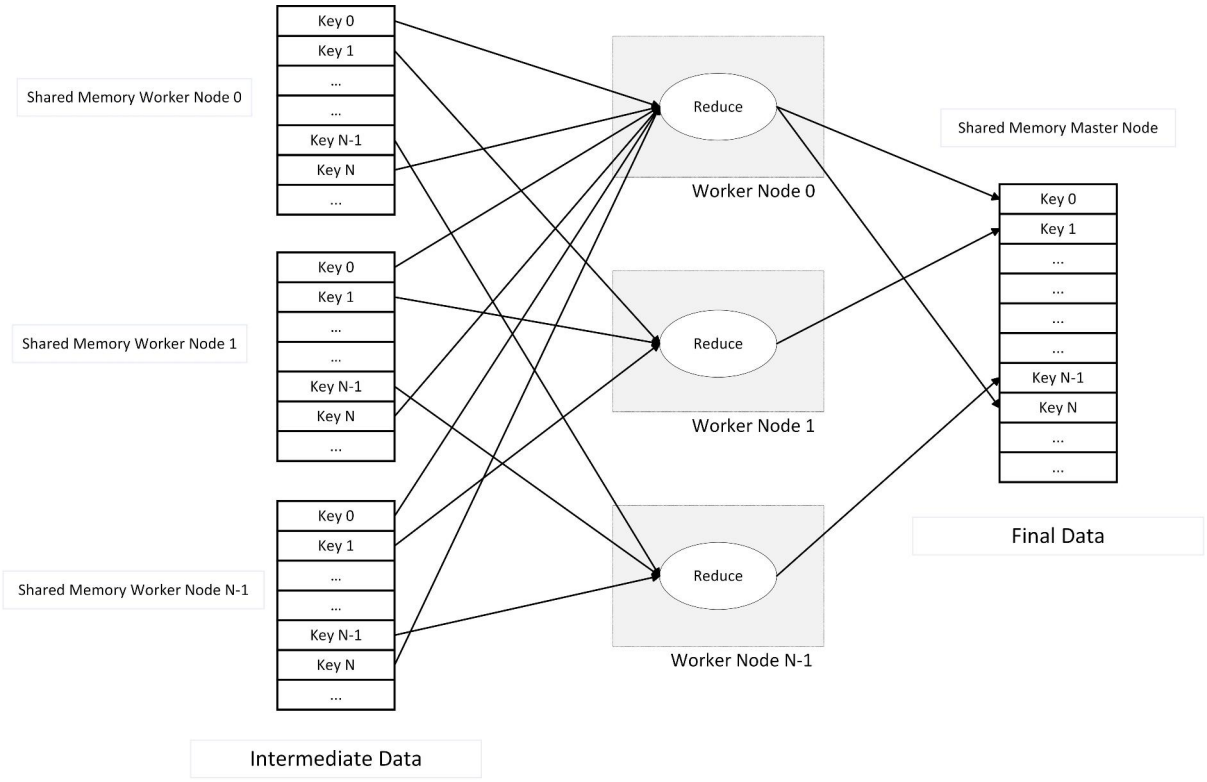


Fig. 3.12 Dataflow during the Reduce phase

3.2.4. Functions and Dataflow

At the beginning of every operation, there is an initialization phase for both Master and Worker nodes. As shown in figure 3.13(a), master node initializes all locks to 0 and creates a pointer to the final data (results). Worker nodes create pointers to all nodes intermediate data and the final data. Then initiate their local combiner data to 0 (fig. 3.13(b)).

To start the Map Phase, input data are statically split to each worker node, as shown in figure 3.11. The number (R) of these splits depends on a user given parameter called unit size. The meaning of unit size may differ from one workload to another. For example in a text processing MapReduce operation, it may dictate how many words will each map task process.

```
*map_lock=0;

for (k=0; k<NUM_WORKERS; k++)
{
    locks[k]= (0x40223820+k*0x20000);
    *locks[k]=0;
}

*red_lock=0;
results=  (0x40203860);
```

(a)

```
my_lock = (0x40223820+CPU_ID*0x20000);

for (k=0; k<NUM_WORKERS; k++)
{
    inter_data[k]= (0x40223860+k*0x20000);
}

results=  (0x40203860);

for (i=0; i<NUM_KEYS; i++)
    my_inter_data[i]=0;
```

(b)

Fig. 3.13 Initialization in Master (a) and Worker (b) nodes

Then, each worker takes a number of splits, which depend on the number of splits *R* and the number of the worker nodes *N*. For example, worker node 0 will process splits 0, N, 2N, etc. Each split will be processed individually by a map task on the node to create pairs of intermediate data, in accordance with the <key, value> structure used in MapReduce. Those intermediate data are stored locally to each node. A local combiner is used to aggregate all the values associated with one key. This technique helps us to minimize the amount of intermediate data that will be later written in the shared memory. As a rule, we tried to avoid remote memory accesses as they are very time consuming. Each map task uses a user provided map function. Figure 3.14 shows an example of such a map function. This map function belongs to a string match paradigm, widely used in MapReduce. The map function takes two arguments, the current position to the input data and the amount of data that will be processed by this call of the function. The amount of data is usually equal to the unit size. In this example, the map function compares the number of words, determined by the size argument, with the keywords and whenever there is a match, it emits the keyword's index and the value 1. The intermediate data are locally combined by the emit_intermediate_data() function which takes as arguments, the index of the key and a value. This function is also customizable by the user. After the worker node processes all splits assigned to it, it writes the intermediate data produced to its shared memory.

```
void map  (cur_pos, size)
{
    for (i=0; i<size; i++)
        for (j=0; j<NUM_KEYS; j++)
            if (strcmp(keys[jj],data_in[cur_pos+i])==0)
                emit_intermediate(j,1);
}

void emit_intermediate(key_ind, value)
{
    my_inter_data[key_ind]+=value;
}
```

Fig. 3.14 String match: Example of map() and emit_intermediate_data() functions

When all worker nodes finish their operations (map tasks and memory transactions), the master node wakes up to initiate the reduce phase. In the reduce phase, each worker node takes a number of keys, all the values associated with those keys and uses the reduce function to create the final data. Figure 3.15 shows an example of a reduce function, as well

as the emit function. This reduce function also belongs to the string match paradigm. The map function takes one argument, the index of the key that will be processed by this function, and produces one resulting value. In this example, the reduce function simply aggregates the values associated with a key and returns the sum. The final data are stored by the emit() function which takes as arguments the index of the key and a value. The number of keys that will be processed by each node depends on the overall number of keys and the number of worker nodes *N*. For example, worker node 0 will handle keys 0, N, 2N, etc, as shown in figure 3.12. To obtain the values associated with a key, every node has to access the intermediate data arrays located on all worker nodes. The final data are immediately written to the final data array. When all reduce tasks are over, worker nodes unlock master node and then quit. Master node wakes up, prints all the results and quits.

```
int reduce (key_index)
{

      for (i=0; i<NUM_PROCS; i++)
                temp+=*(inter_data[i]+key_index);
      return temp;
}

void emit (key_index, value)
{
      *(res+key_index)=value;
}
```

Fig. 3.15 String match: Example of reduce() and emit () functions

Chapter 4: Experimental Evaluation

4.1 Benchmarks

Three different benchmarks were used for the experimental evaluation of the proposed framework. Those are *string match, histogram* and *average*.

String Match: It processes a text and counts the occurrences of ten keywords provided in a key matrix. Each Map task parses a portion of the text and returns a key and a value, as a flag to indicate whether it was a match or not. The reduce task sums all the occurrences across all the processors.

Histogram: It analyzes a given bitmap image to compute the frequency of occurrence of a value in the 0-255 range for the RGB components of the pixels. The algorithm assigns different portions of the image to different Map tasks, which parse the image and insert the frequency of component occurrences into arrays. The reduce tasks sum up these numbers across all the processors.

Average: It computes the average rating per user using anonymized movies rating data which is of the form <movie_id: list{rater_id, rating}>. Each Map task parses a portion of the input and returns rater ids and the ratings associated with them. The reduce task sums all ratings associated with one rater and divides by the number of ratings.

Those three benchmarks are inspired by different sectors of computing, namely, Enterprise Computing (String Match, Average) and Image Processing (Histogram).

4.2 Test Configuration

Four NoC configurations with 9, 16, 25 and 36 nodes were used to test the proposed framework. The nodes were arranged in 3x3, 4x4, 5x5 and 6x6 meshes, accordingly. Each node consisted of a DMC, a Leon3 processor and memory, as presented in the previous chapter. Each Leon3 processor has 4KB of data cache and operates at 500MHz. The DMC also operates at 500MHz.

Moreover, the framework was tested for three different input sizes: a small (80KB), a medium (10MB) and a large (100MB) input. The small input was set to 80KB, because it was the maximum input that could be given to the platform statically. To create larger inputs, we processed the same data multiple times. For example, for the medium input we processed the small input 125 times. Unfortunately, most results were taken with the small input, because the duration of the simulations was prohibiting, as shown in table 4.1. Especially with the large input, we assume that simulations for String Match and Histogram would need more than a month to finish.

Finally, the framework was tested for four different unit sizes across all NoC configurations. Those unit sizes are 512, 1024, 2048 and 4096 bytes. They were chosen as fractions of the Leon's data cache size.

Benchmark	**Average Duration per Simulation**		
	80KB	10MB	100MB
String Match	3h 17min	19d 5h 44min	*
Histogram	1h 6min	3d 9h 44min	*
Average	25min	15h 42min	6d 3h 59min

Table 4.1. Average Duration per Simulation

4.3 Basic Performance Evaluation

The performance of the proposed framework was evaluated in terms of core and input scalability. It was tested on grids with 9, 16, 25 and 36 cores for all inputs. The speedup for each application was calculated compared to sequential code as we scale the number of available cores. The sequential code was tested on a core identical to those in our grids.

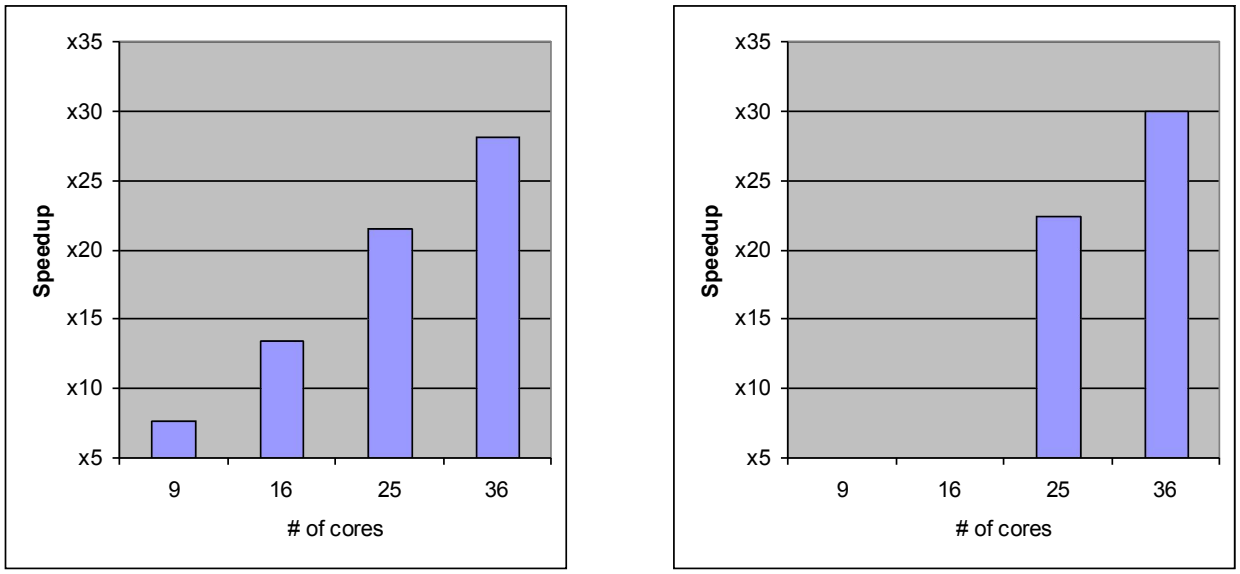


Fig. 4.1 String match: Speedup for 9, 16, 25 and 36 cores with small (a) and medium (b) input

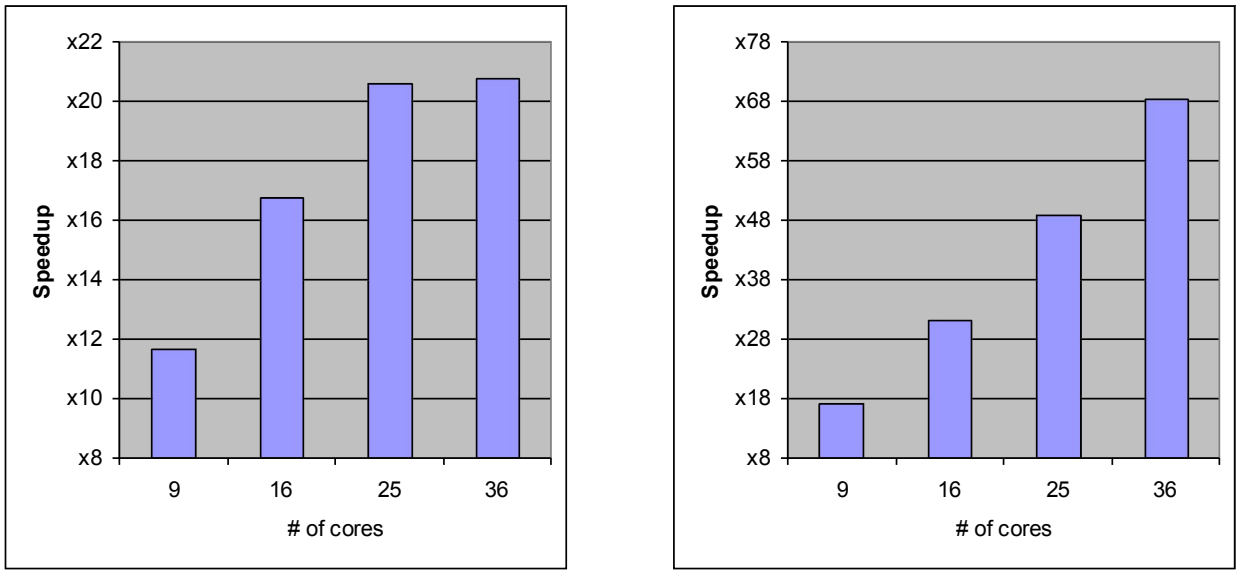


Fig. 4.2 Histogram: Speedup for 9, 16, 25 and 36 cores with small (a) and medium (b) input

Figure 4.1(a) presents the results for the string match benchmark with the small input. The string match benchmark shows good scalability, achieving speedups from x7.7 for 9 cores up to x28.1 for 36 cores. With the medium input, the string match benchmark

exhibits similar results with x22.4 speedup for 25 cores and x29.9 for 36 cores, as shown in figure 4.2(b).

Figure 4.2 presents the results for the histogram benchmark with the small and the medium input. The Histogram benchmark also scales well. With the small input the framework achieved speedups from x11.7 for 9 cores up to x20.7 for 36 cores, as shown in figure 4.2 (a). It is interesting to notice that there is almost no speedup as we move from 25 to 36 cores. This shows us that the cost of synchronization for the extra cores surpasses the gain from the extra computational power, and will be further analyzed in section 4.5. Figure 4.2 (b) exhibits the results with the medium input. The framework achieved speedups from x17.3 for 9 cores up to x68.4 for 36 cores, thus being consistent with the nature of MapReduce, which makes it more efficient as inputs grow larger. The gain in speedup varies from 47% for 9 cores to 230% for 36 cores.

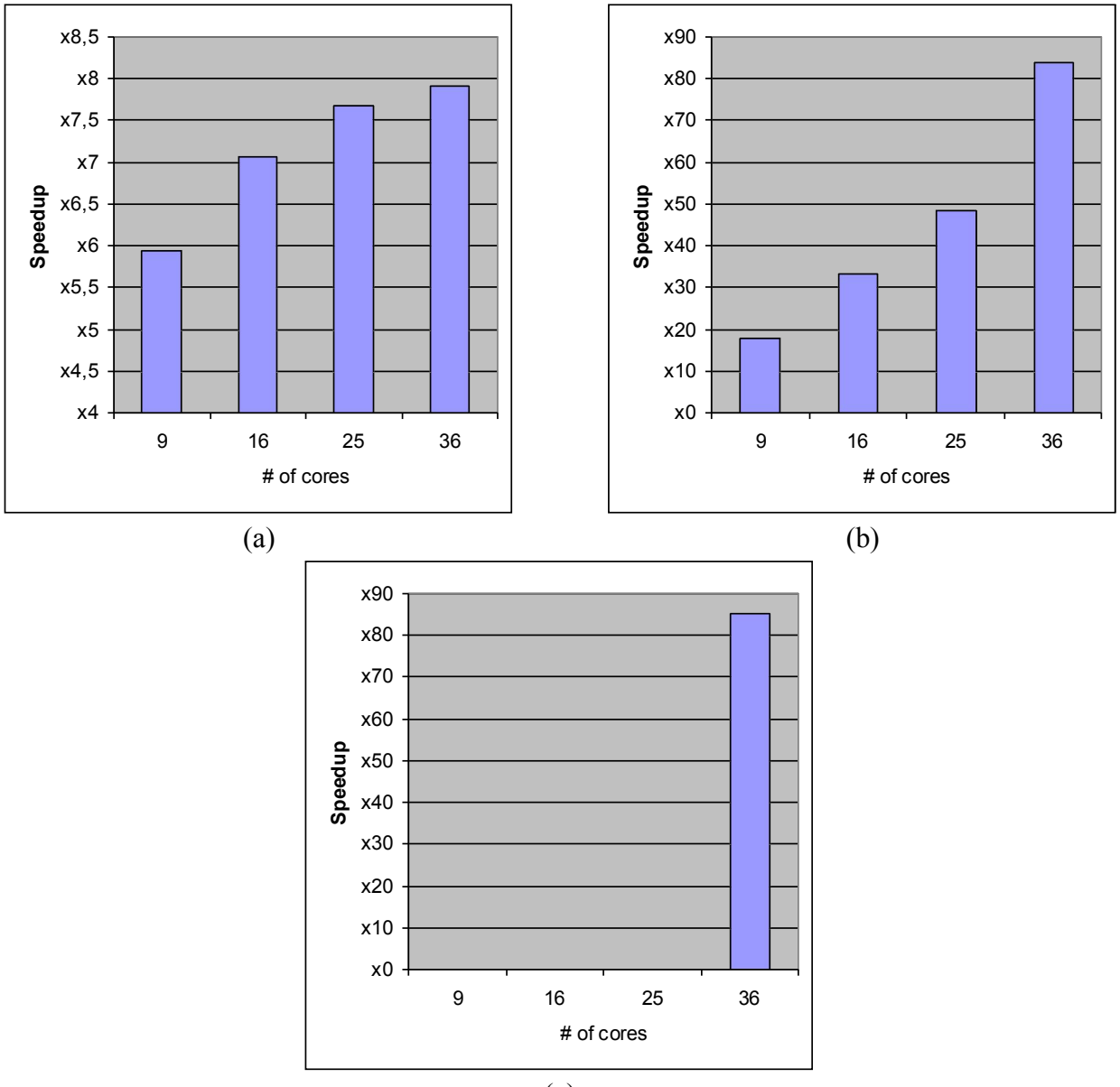


Fig. 4.3 Average: Speedup for 9, 16, 25 and 36 cores with small (a), medium (b) and large (c) input

Finally, figure 4.3 shows the results for the average benchmark with the small and medium input. With the small input the framework does not scale well with speedup varying

from x5.9 for 9 cores up to x7.9 for 36 cores, as shown in figure 4.3 (a). This shows us that synchronization is more time consuming than computation itself, and will be further analyzed in section 4.5. Figure 4.3 (b) exhibits far better results with the medium input. The framework achieved speedups from x17.9 for 9 cores up to x83.6 for 36 cores, thus supporting our previous remark about the cost of synchronization. The gain in speedup varies from 200% for 9 cores to 957% for 36 cores. With the large input, average benchmark achieves x85.2 speedup for 36 cores (fig. 4.3(c)), exhibiting a small gain from x83.6 with the medium input.

4.4 Dependency to Unit Size

The proposed framework was tested for four different unit sizes across all configurations with the small input.

Figure 4.4 shows the results for the string match benchmark. As we can see, unit size has small impact for configurations with less than 36 cores. For 36 cores, large unit sizes (2 and 4KB) have negative impact on the speedup. Surprisingly, the performance for 36 cores with large unit sizes is the same as the performance for 25 cores. The reason behind this performance loss hides in the meaning of large unit sizes. A large unit size dictates that less, but larger map tasks, will be performed. This means that if some cores delay for any reason the impact in the overall execution will be greater, as they have larger tasks to complete.

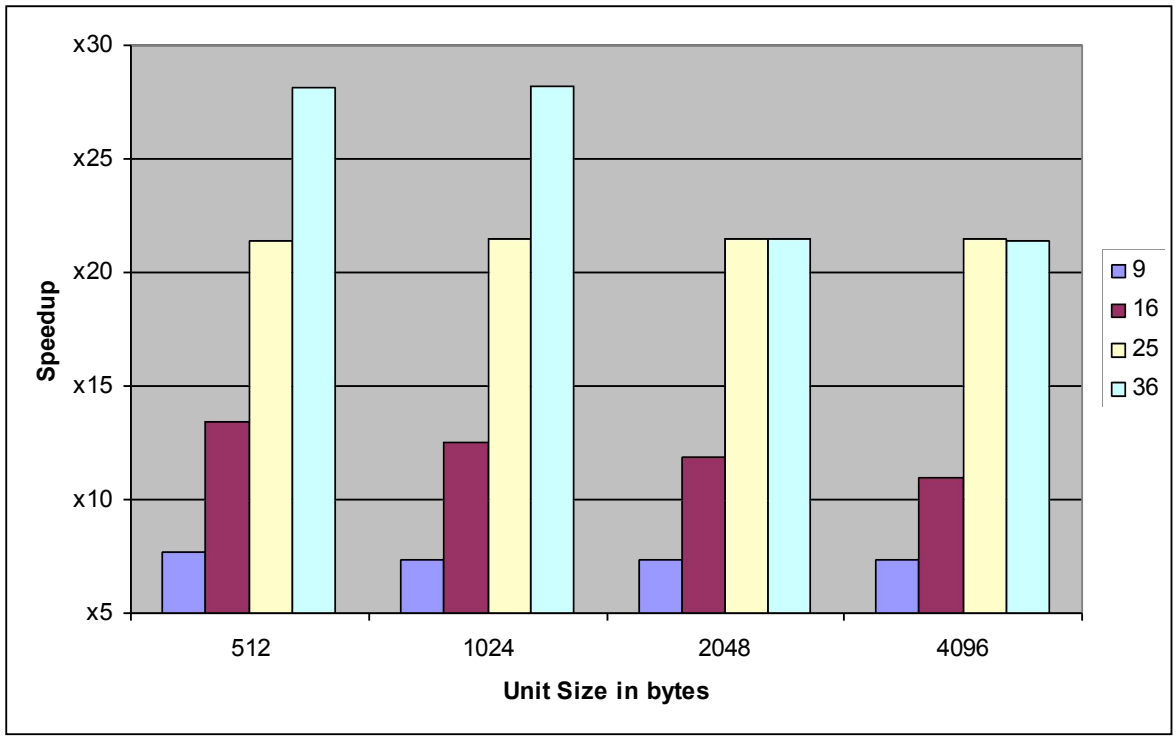


Fig. 4.4 String match: Speedup for 4 different Unit Sizes

Figure 4.5 shows the results for the histogram benchmark. Here the impact of large unit sizes is even greater. For 36 cores, the performance of unit sizes larger than 512 bytes is worse than the performance for 25 cores. The reason behind this tremendous performance loss might lie in the nature of the input. Images exhibit spatial correlation, meaning that pixels in a neighbourhood have similar RGB values. A small unit size means that each core

will process a small part of the image and, because of the spatial correlation, will reuse the same or neighbouring memory addresses over and over again.

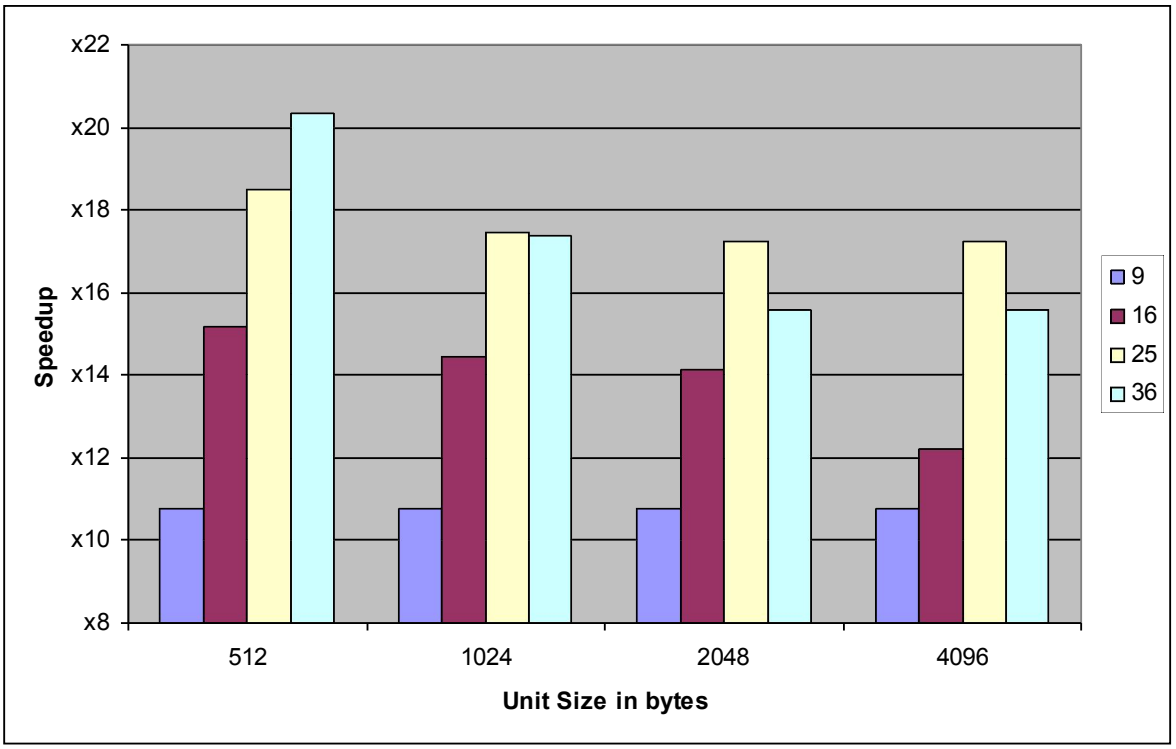


Fig. 4.5 Histogram: Speedup for 4 different Unit Sizes

Finally, Figure 4.6 shows the results for the average benchmark. As we can see, the average benchmark exhibits similar behaviour as the string match benchmark.

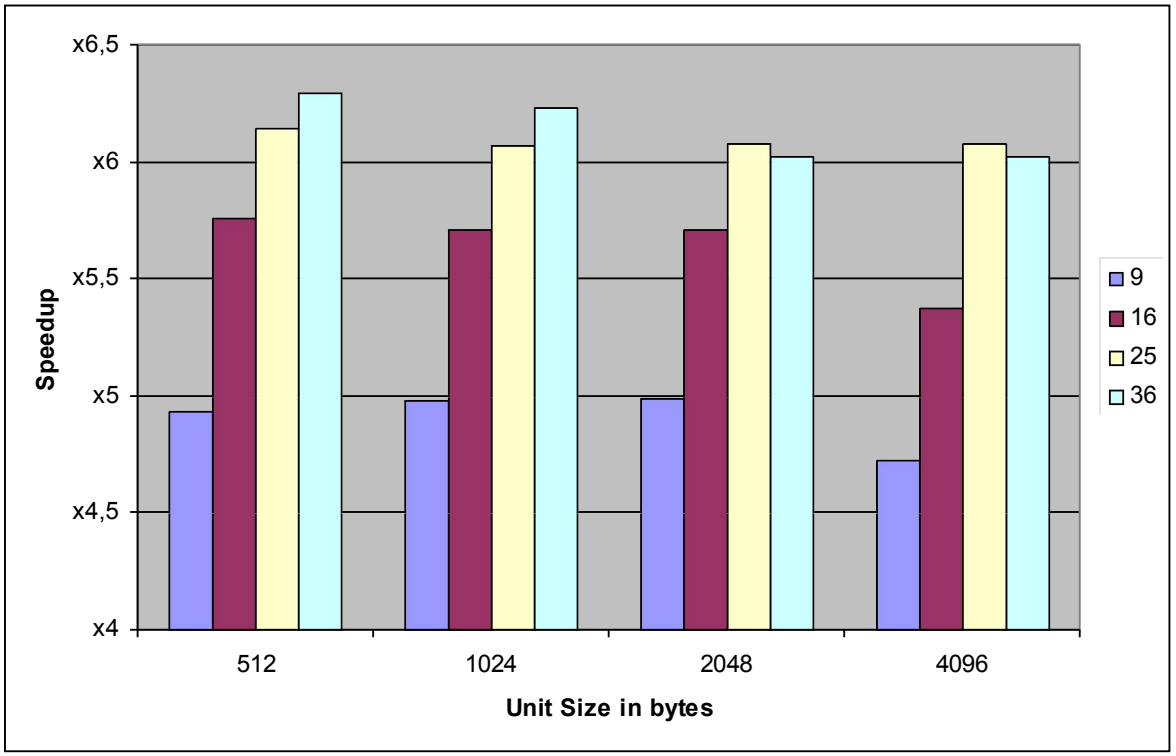


Fig. 4.6 Average: Speedup for 4 different Unit Sizes

4.5 Execution time breakdown

Figures 4.7 - 4.11 show the execution time breakdown for all the benchmarks. There are four discrete phases in every execution. At the beginning, there is an initialization phase, indicated in blue. Then, there is the Map phase during which all map tasks take place, indicated in red. Finally, there is the Reduce phase indicated in light blue. In between, there is a phase when cores finalize their memory transactions, indicated in yellow.

As we can see in figure 4.7, string match spends most of the execution time in the Map phase, which reduces in length as we scale the number of cores. This is natural, because there were only ten keywords and therefore only ten reduce tasks. Moreover, each map task consists of several function calls to compare the given word with the keywords, making each task time consuming.

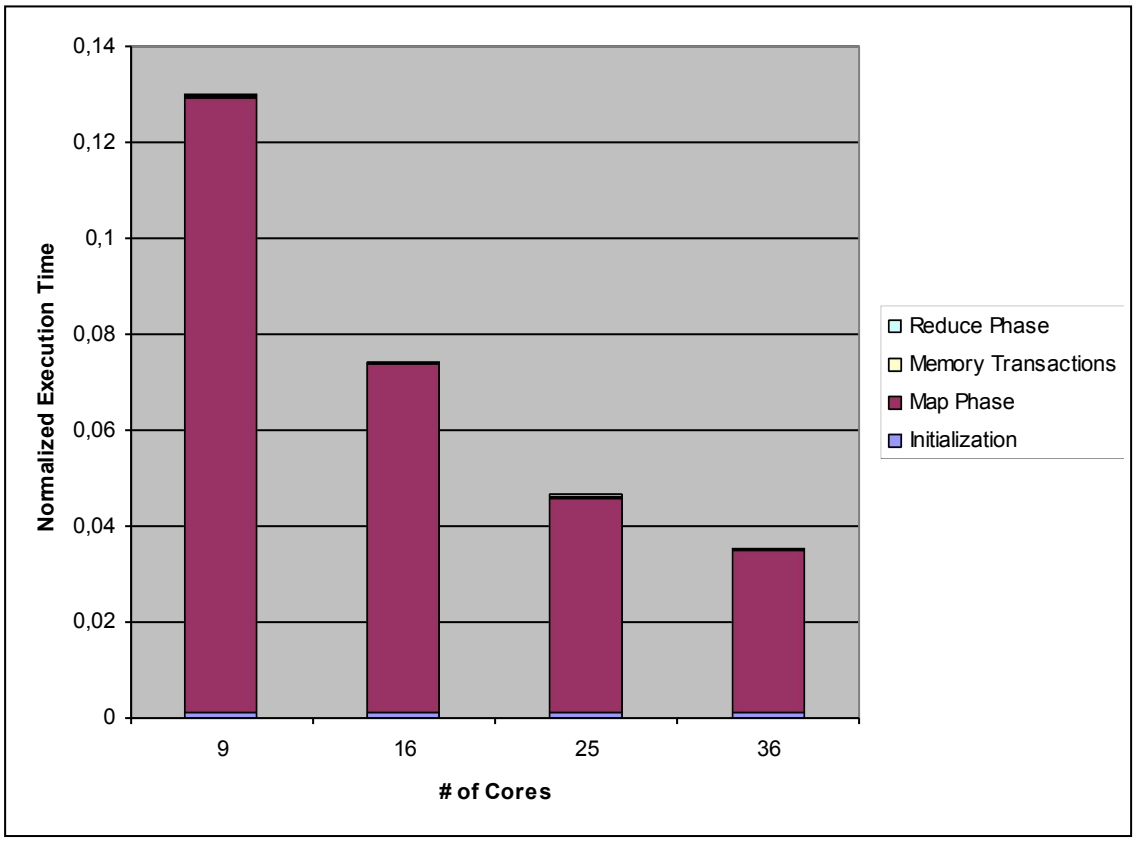


Fig. 4.7 String match: Execution time breakdown for 9, 16, 25 and 36 cores with the small input

Figure 4.8 shows that histogram with the small input spends most time in Map phase, but as we scale the number of cores Map phase's duration becomes comparable with the other phases. A closer look at figure 4.8 reveals two very important observations. Firstly, figure shows that the gain in Map phase's duration from the transition from 25 to 36 cores is diminished by an overhead in memory transactions. Consequently, there is no speedup in this transition. Secondly, we can see that reduce phase's duration does not decrease as we scale the number of cores, as it would be expected. To find the reason behind this behaviour, we have to analyze the shared memory accesses for one core during the reduce phase for two different configurations, as load/store operations, especially remote ones, are more time consuming than other operations. For example, in a 36 cores

configuration a single core would have to make 34 remote loads, 1 local load and 1 remote store for each key and it would process 22 keys, coming to a total of 748 remote loads, 21 local loads and 21 remote stores. In a 16 cores configuration a single core would have to make 14 remote loads, 1 local load and 1 remote store for each key and it would process 51 keys, coming to a total of 714 remote loads, 14 local loads and 14 remote stores. It is obvious that there is no gain for the reduce phase as we scale the cores.

Figure 4.9 shows the execution time breakdown for histogram with the medium input. The duration of the Map phase is overwhelming for this input. This is expected as the other phases depend only on the number of keys that is the same, whereas the size of the input that needs to be parsed is 125 times larger.

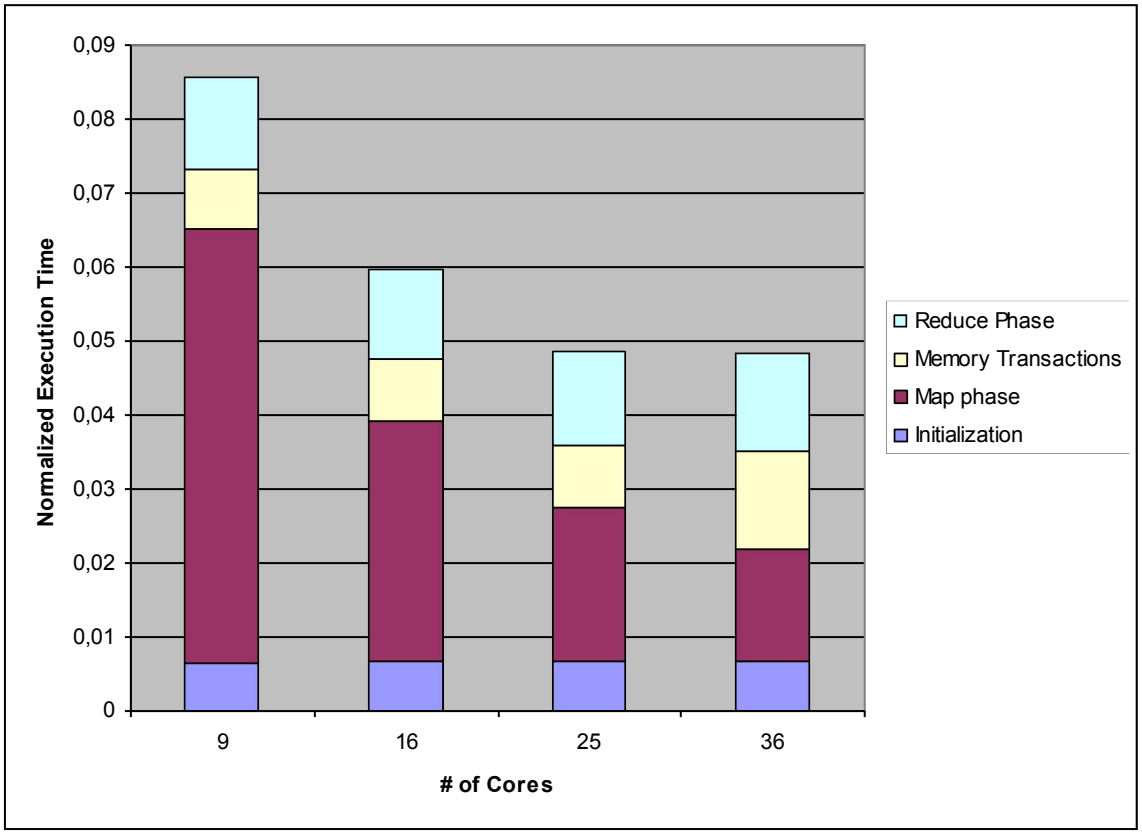


Fig. 4.8 Histogram: Execution time breakdown for 9, 16, 25 and 36 cores with the small input

Figures 4.10-4.11 show the execution time breakdown for the average benchmark with the small and medium input. The average benchmark exhibits similar behaviour as the histogram benchmark. With the small input map phase takes most of the time, but as core number scales its duration decreases and reduce phase accounts for most of the execution time. Reduce phase's duration does not scale, for the same reasons that were analyzed before. With the medium input, map phase dominates almost all of the execution time.

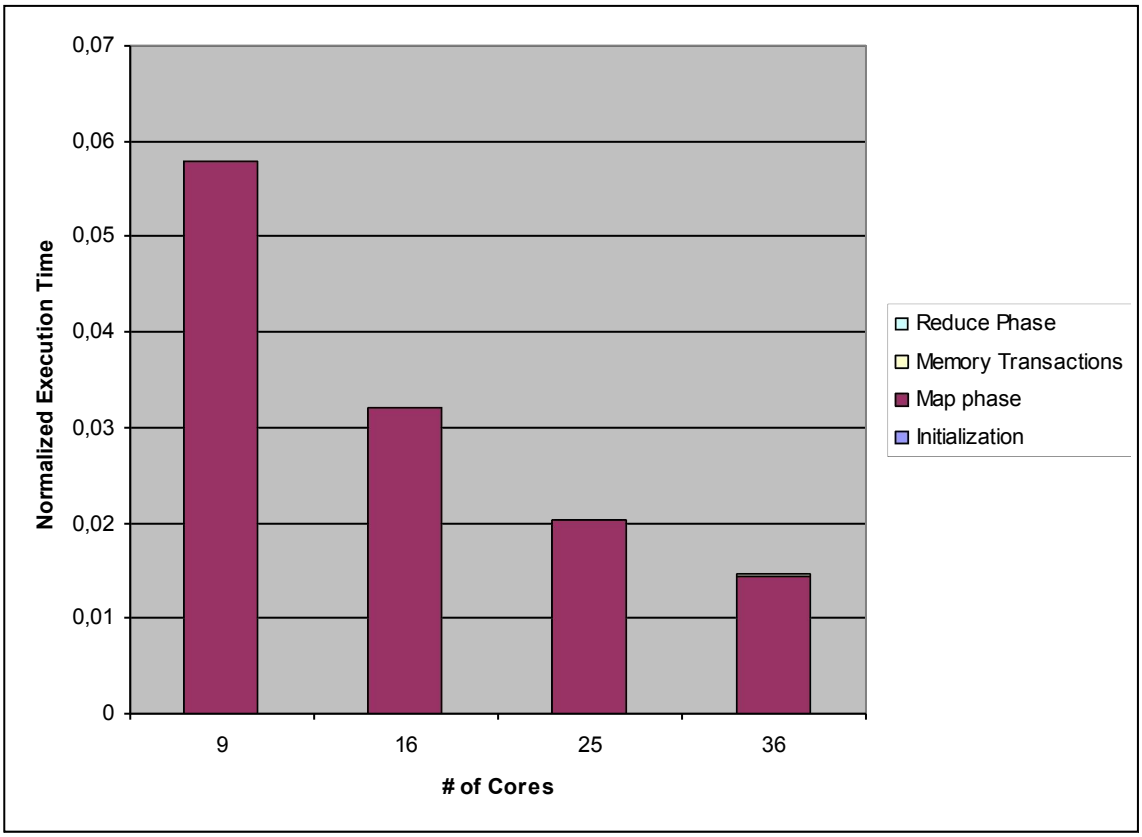


Fig. 4.9 Histogram: Execution time breakdown for 9, 16, 25 and 36 cores with the medium input

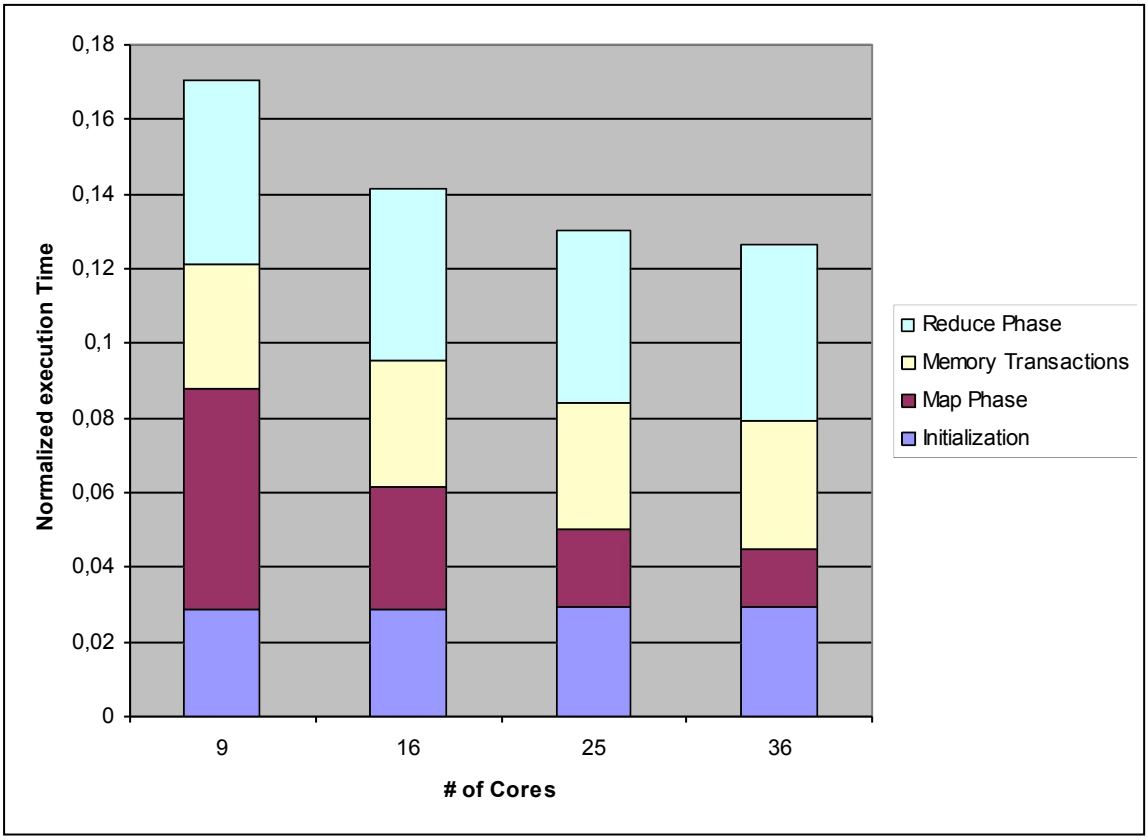


Fig. 4.10 Average: Execution time breakdown for 9, 16, 25 and 36 cores with the small input

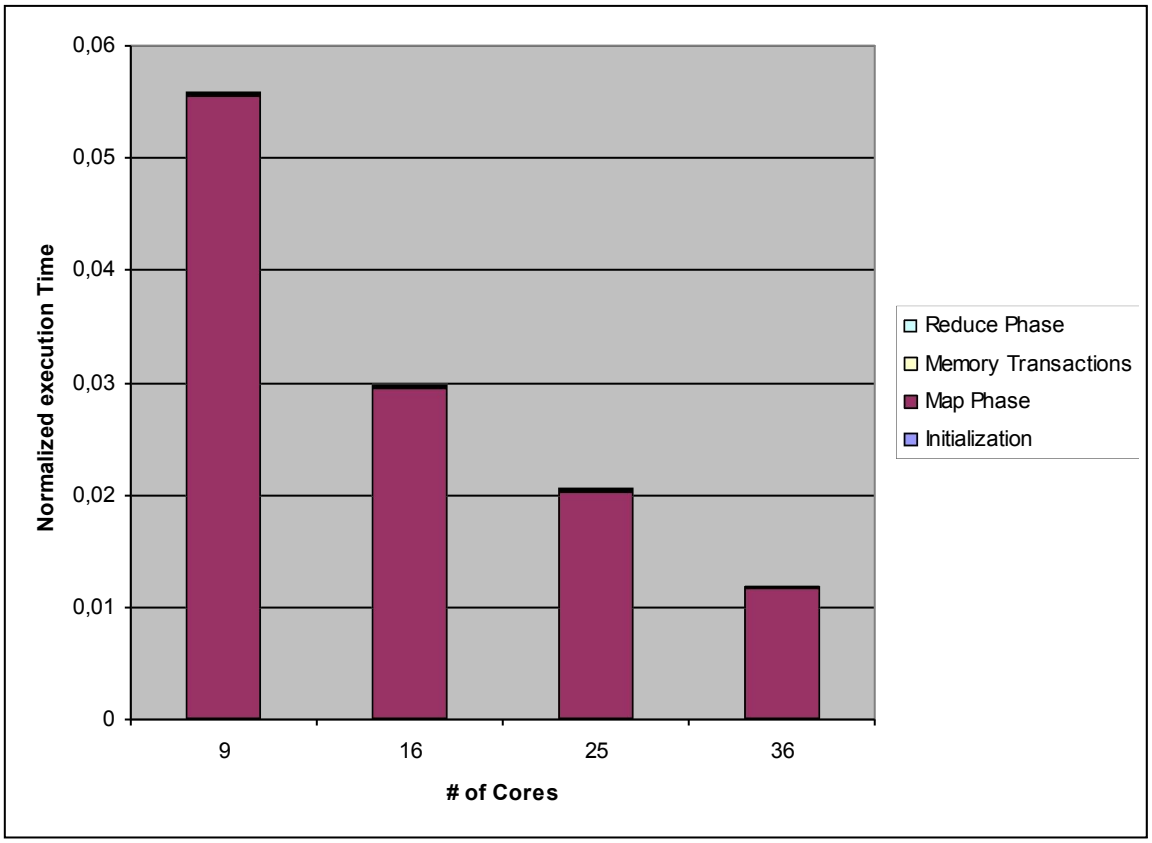


Fig. 4.11 Average: Execution time breakdown for 9, 16, 25 and 36 cores with the medium input

4.6 Summary

To sum up, the proposed framework shows good scalability for all benchmarks. It exhibits impressive speedups, achieving up to x85.2 for the average benchmark with the large input. It is, also, clear that it favours larger inputs, as it would be expected from a MapReduce framework. The cost of synchronization seems to have no impact on performance for the medium and large input, as shown in section 4.5. Moreover, results show that performance depends on the nature of each benchmark, as it was, also, concluded by previous research on MapReduce [12][17]. Overall, MapReduce seems like a logical choice for data processing on Many-core platforms with DSM characteristics.

Chapter 5: Conclusions

5.1 Achievements of the proposed framework

Embedded systems have come a long way in the last decade and have a bright future in front of them. Mobility and low power consumption have become prime design principles. Modern Embedded systems come with great processing power, a power that has to be unlocked and fully utilized. To that end, it is very important for embedded software designers to explore and learn from existing general purpose paradigms. The proposed framework proves that today's embedded systems can benefit greatly from existing HPC methods.

MapReduce is a dominant method for scalable performance on shared-memory systems with simple parallel code. The proposed MapReduce framework provided a scalable solution for data processing in a Many-core system, while fully utilizing the platform's characteristics. In this way, it achieved all the objectives set in section 2.4. Moreover, it can be easily programmed by someone who's familiar with MapReduce and does not require any knowledge of parallel programming.

5.2 Future Work

During our time working on the NoC platform presented in section 3.1, it became obvious that there are certain ways in which it could be enhanced.

First and foremost, future programmers would benefit greatly from the implementation of a coherence protocol on the DSM. A paradigm could be found in modern coherence protocols used in general purpose shared memory multi-core systems. A MESI protocol would be a good and simple match. This protocol could be either implemented as a separate module in VHDL or in assembly microcode running on DMEs, taking advantage of the special synchronization hardware.

Furthermore, the existence of a file system on the simulator would greatly benefit any programming attempt on the application platform level. At the moment there is no higher programming level I/O capability available.

Finally, after the implementation of the coherence protocol, the proposed framework could be enhanced in a number of ways. Those are:

1. The implementation of a dynamic work scheduler, which will dynamically distribute the remaining tasks to the available workers.
2. The implementation of a fault tolerance system on the framework. This system could monitor each worker and if any of them stops responding, the scheduler would redistribute the worker's task.
3. The exploration of the generalized reduction presented in [17].

A dynamic work scheduler would undoubtedly give a great boost to the framework's overall performance. Static scheduling would exhibit optimal performance, if all tasks presented equal workload. But, even with the same amount of data, each task differs from another in terms of execution time. That makes workload balancing problematic when using static scheduling. Thus, a dynamic scheduler would thrive over the static scheduler used. Still, a dynamic scheduler would require more inter-core communication, which is achieved in the NoC through the DSM, therefore requiring it to be coherent in order to work properly.

A fault tolerance system could be easily created after the implementation of the dynamic scheduler. The scheduler will periodically request workers to respond. If any worker stops responding, the scheduler will reschedule the task performed in another worker. The fault tolerance system could, also, be used to deal with “stranglers”, aka workers which take more time than usual to perform a task. Their task could be scheduled to another free worker and keep the results from the node that finishes first.

Last, but not least, generalized reduction exhibited better results opposed to traditional MapReduce method for data-mining operations. It would be beneficial if it was tested on the presented NoC in terms of scalability and general performance. The generalized reduction could be built on the existing framework or, even better, on an enhanced version of the framework by the introduction of a dynamic scheduler.

Bibliography

[1]. H. Sutter, “The free lunch is over: a fundamental turn toward concurrency in software,” Dr. Dobb's Journal, vol. 30, no. 3, pp. 202–210, 2005.

[2]. A. A. Vajda, Programming Many-Core Chips, Springer, 2011. doi: 10.1007/978-1-4419-9739-5

[3]. http://www.tilera.com/products/processors/TILE-Gx_Family

[4]. Agarwal, A.; Levy, M., "The KILL Rule for Multicore," Design Automation Conference, 2007. DAC '07. 44th ACM/IEEE , vol., no., pp.750,753, 4-8 June 2007

[5]. Robert Hilbrich, “How to Safely Integrate Multiple Applications on Embedded Many-Core Systems by Applying the “Correctness by Construction” Principle,” Advances in Software Engineering, vol. 2012, Article ID 354274, 14 pages, 2012. doi:10.1155/2012/354274

[6]. http://newsroom.intel.com/community/intel_newsroom/blog/2012/11/12/intel-delivers-new-architecture-for-discovery-with-intel-xeon-phi-coprocessors

[7]. J. Gantz and D. Reinsel (IDC), “THE DIGITAL UNIVERSE IN 2020: Big Data, Bigger Digital Shadows, and Biggest Growth in the Far East,” http://idcdocserv.com/1414, 2012, last accessed: Mar 2013.

[8]. J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In OSDI’04, 6th Symposium on Operating Systems Design and Implementation, Sponsored by USENIX, in cooperation with ACM SIGOPS, pages 137–150, 2004.

[9]. Tom White, “Hadoop: The Definite Guide,” O’Reilly Media, Inc., 2009.

[10]. Lin, J., Dyer, C., “Data-Intensive Text Processing with MapReduce,” Synthesis Lectures on Human Language Technologies. Morgan & Claypool Publishers, 2010.

[11]. G. M. Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” in Proceedings of the Spring Joint Computer Conference, pp. 483–485, ACM, April 1967.

[12]. Ranger, C.; Raghuraman, R.; Penmetsa, A.; Bradski, G.; Kozyrakis, C., "Evaluating MapReduce for Multi-core and Multiprocessor Systems," High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on , vol., no., pp.13,24, 10-14 Feb. 2007 doi: 10.1109/HPCA.2007.346181

[13]. Yoo, R.M.; Romano, A.; Kozyrakis, C., "Phoenix rebirth: Scalable MapReduce on a large-scale shared-memory system," Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on , vol., no., pp.198,207, 4-6 Oct. 2009 doi: 10.1109/IISWC.2009.5306783

[14]. Talbot, J.; Yoo, R.M.; Kozyrakis, C., “Phoenix++: modular MapReduce for shared-memory systems,” in Proceedings of the second international workshop on MapReduce and its applications (MapReduce '11). 2011. doi:10.1145/1996092.1996095

[15]. R. Jin and G. Agrawal, “A middleware for developing parallel data mining implementations,” in Proceedings of the first SIAM conference on Data Mining, Apr. 2001.

[16]. Wei Jiang; Ravi, V.T.; Agrawal, G., "Comparing map-reduce and FREERIDE for data-intensive applications," Cluster Computing and Workshops, 2009.

CLUSTER '09. IEEE International Conference on , vol., no., pp.1,10, Aug. 31 2009-Sept. 4 2009 doi: 10.1109/CLUSTR.2009.5289199

[17]. Wei Jiang, Vignesh T. Ravi, and Gagan Agrawal. “A Map-Reduce System with an Alternate API for Multi-core Environments”. In Proceedings of the 10th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, pages 84–93, May 2010.

[18]. Xiaowen Chen; Zhonghai Lu; Jantsch, A.; Shuming Chen, "Supporting Distributed Shared Memory on multi-core Network-on-Chips using a dual microcoded controller," Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010 , vol., no., pp.39,44, 8-12 March 2010 doi:10.1109/DATE.2010.5457240