Daniel Lobato Vieira Magro

BSc in Computer Science

Cache-conscious Splitting of MapReduce Tasks and its Application to Stencil Computations

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática

Orientador: Prof. Doutor Hervé Miguel Cordeiro Paulino,
Prof. Auxiliar, Universidade Nova de Lisboa

Júri

Presidente:	Prof. Doutor Miguel Pessoa Monteiro
Arguente:	Prof. Doutor João Pedro Barreto
Vogal:	Prof. Doutor Hervé Miguel Cordeiro Paulino

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

November, 2015

Cache-conscious Splitting of MapReduce Tasks and its Application to Stencil Computations

Este documento foi gerado utilizando o processador (pdf)LaTeX, com base no template "unlthesis" [1] desenvolvido no Dep. Informática da FCT-NOVA [2]. [1] https://github.com/joaomlourenco/unlthesis [2] http://www.di.fct.unl.pt

Acknowledgements

Quero agradecer ao meu orientador, Prof. Hervé Paulino, por me ter dado a possibilidade de trabalhar com ele, pela atenção, paciência e apoio na realização desta dissertação.

Também quero agradecer ao Departamento de Informática da Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa, pelas condições de trabalho oferecidas ao nível de infreastruturas e ambiente de trabalho proporcionado, pela oportunidade de participação no apoio pedagógico que me ajudou a desenvolver novas capacidades e mostrou-me o outro lado do ensino.

Para terminar, quero agradecer a todos aqueles que estiveram comigo ao longo da realização desta dissertação, família, amigos e colegas, sendo que sem o seu apoio esta tarefa teria sido muito mais díficil.

Abstract

Modern cluster systems are typically composed by nodes with multiple processing units and memory hierarchies comprising multiple cache levels of various sizes. To leverage the full potential of these architectures it is necessary to explore concepts such as parallel programming and the layout of data onto the memory hierarchy. However, the inherent complexity of these concepts and the heterogeneity of the target architectures raises several challenges at application development and performance portability levels, respectively. In what concerns parallel programming, several model and frameworks are available, of which MapReduce [16] is one of the most popular. It was developed at Google [16] for the parallel and distributed processing of large amounts of data in large clusters of commodity machines. Although being very powerful tools, the reference MapReduce frameworks, such as Hadoop and Spark, do not leverage the characteristics of the underlying memory hierarchy. This shortcoming is particularly noticeable in computations that benefit from temporal locality, such as stencil computations.

In this context, the goal of this thesis is to improve the performance of MapReduce computations that benefit from temporal locality. To that end we optimize the mapping of MapReduce computations in a machine's cache memory hierarchy by applying cache-aware tiling techniques. We prototyped our solution on top of the framework Hadoop MapReduce, incorporating a cache-awareness in the splitting stage.

To validate our solution and assess its benefits, we developed an API for expressing stencil computations on top the developed framework. The experimental results show that, for a typical stencil computation, our solution delivers an average speed-up of 1.77 while reaching a peek speed-up of 3.2. These findings allows us to conclude that cache-aware decomposition of MapReduce computations considerably boosts the execution of this class of MapReduce computations.

Keywords: Application Decomposition, Cache-Conscious, Stencil Computations, MapReduce, Hadoop

Resumo

Os sistemas de cluster modernos são tipicamente compostos por nós com várias unidades de processamento e hierarquias de memória que contêm vários níveis de cache de vários tamanhos. Para aproveitar o potencial destas arquiteturas é necessário explorar conceitos como programação paralela e o layout de dados na hierarquia de memória. No entanto, a complexidade inerente a estes conceitos e a heterogeneidade dos arquiteturas alvo levantam diversos desafios aos níveis de desenvolvimento de aplicações e ao nível da sua portabilidade de desempenho. No que diz respeito à programação paralela, existem vários modelos e ferramentas disponíveis, das quais o MapReduce [16] é um dos mais populares. Este foi desenvolvido pela Google [16] para o processamento paralelo e distribuído de grandes quantidades de dados em grandes aglomerados de máquinas. Apesar de serem muito poderosas, as ferramentas MapReduce de referência, como o Hadoop e o Spark, não aproveitam as características da hierarquia de memória subjacente. Esta lacuna é particularmente visível em computações que beneficiam da localidade temporal, tais como as computações *stencil*.

Neste contexto, o objetivo deste trabalho é melhorar o desempenho de computações MapReduce que beneficiem da localidade temporal. Para esse fim otimizamos o mapeamento das computações MapReduce na hierarquia de memória *cache* do uma máquina, aplicando técnicas de *tiling cache-conscious*. dado isto, desenvolvemos um protótipo da nossa solução sobre a ferramenta Hadoop MapReduce, introduzindo medidas *cache-conscious* na fase de decomposição de dados.

Para validar a nossa solução e avaliar os seus benefícios, também desenvolvemos uma API para expressar computações *stencil* sobre a estrutura desenvolvida. Os resultados experimentais mostram que para uma computação *stencil* típica, a nossa solução oferece uma média de aumento de speed-up de 1.77, chegando-se mesmo a alcançar um pico de aumento de speed-up de 3.2. Estes resultados permitem-nos concluir que a decomposição *cache-conscious* de computações MapReduce aumenta consideravelmente a execução desta classe de computações MapReduce.

Palavras-chave: Decomposição de Aplicações, Cache-Conscious, Computações Stencil, MapReduce, Hadoop

Contents

List of Figures

List of Tables

Listagens

Chapter 1

Introduction

1.1 Motivation

Multi-core processors are the current de-facto standard in commodity computers, tablets, smartphones and other devices. This approach has been the trend for some years and it is supported by the well-known Moore's Law, which observes that the number of transistors in a chip doubles every two years. With this trend, the horizontal scaling of processing cores led us to ever increasing computational power and brought us to one of the most relevant computer architecture paradigm, the multicore architectural model. Alongside with the horizontal expansion of CPU architectures, the memory hierarchy is scaled vertically resulting in more levels of cache memory to improve the data and instruction access by the processing cores. This two-dimensional scaling has flooded the market with a very heterogeneous offer in what concerns the number of cores/memory hierarchy configurations.

To benefit more from these architectures it is necessary to explore and combine concepts such as parallel programming and the mapping of applications onto the memory hierarchy. The first concept is closely linked to the usage of multiple processing units to produce faster computations, while the second is directly bonded to the leveraging of cache memories, in order to take advantage of the locality principle. The cache memory hardware available in these architectures can only guarantee spatial locality, which means that only data located close together is reused. The algorithms used to achieve this spatial locality are based only on the recent history of data access, and do not take into account the application's behaviour. Given that, the hardware only guarantees that the most recently accessed data is in the cache memory. Normally this behaviour is sufficient to improve the performance of computations that take advantage of spatial locality. However, this may not be enough for computations that benefit from temporal locality,

because this behaviour can lead to the eviction of cache lines that contain data which may be needed in a near future and, hence, important to a given application. With this, it is up to the application to express the data access patterns that minimize the eviction of useful data from the cache. The definition of these patterns can be delegated to the compiler, to the runtime or even to the programmer.

In order to explore the locality concepts and to efficiently avail the hardware in the current CPU's, a software developer must have some non-trivial knowledge about parallel programming and advanced computer architecture, which can transcend the skills of the common software developer. Moreover, with the heterogeneity of these architectures a question that may arise is related to the portability of the performance gains. The solutions developed to cope with these problems have to fulfil the performance expectations for different combinations of cores, cache memories and affinities between the two.

All these problems can add-up and produce the cumbersome task of ensuring the paralellization and the efficient mapping of applications onto the memory hierarchy. Thus, a system that could handle these tasks autonomously would be a great contribution to fully take advantage of the available hardware.

In this thesis we are particularly interested on investigating how to leverage cache hardware in the increasingly popular parallel and distributed big data processing frameworks, of which MapReduce is the basis and the most prominent model.

1.1.1 The MapReduce Programming Model and Framework

MapReduce [16] is a programming model and framework that processes large amounts of data in distributed environments. The model was developed at Google, by Jeffrey Dean and Sanjay Ghemawat, whose motivation was to conceive a resilient (in the presence of faults), scalable and efficient system for the distributing computing of Google's index for the World Wide Web search engine. The result was a very simple programming model backed up by a powerful runtime system, that subsumes the aforementioned resilience, scalability, and efficiency requirements. In this programming model a programmer does not have to hold deep knowledge of parallel or distributed programming. This model allows the development of a system that automatically parallelizes and distributes large-scale computations with good processing efficiency on different environments, such as: common multicore architectures, small clusters or large data-centers. These characteristics led to the utilization of this model in many different areas, such as distributed searching, distributed sorting, document clustering and even machine learning [15].

Crucial to this popularity has also been the Hadoop [2] open source implementation made available by Apache. Hadoop MapReduce is a widespread framework for distributed storage and processing, used by companies such as Yahoo!, Facebook, among others [6], and is present in some of the more relevant cloud services, such as Microsoft Azure and the Amazon Web Services.

As the scope of MapReduce applications has largely surpassed the one for which it

was designed for, several limitations of both the original programming and execution model were identified. This limitations are visible in paradigms such as stream processing, real time processing, or simply the mapping of some problems to a map or reduce task. To overcome such limitations, many special-purpose Hadoop-based frameworks have been proposed throughout the years. Prominent examples are: Pig [4], for data analysis programs and, for a simpler way of dealing with complex tasks related to data transformations; Twister [23], a lightweight framework with no more than 5600 lines of code for the efficient implementation of iterative MapReduce computations; Hive [3], for a better management and analysis of datasets using some relational operations allowed by an SQL-like language. Complementary, alternative systems have been proposed altogether. The most noticeable is Spark [5], a data processing framework or, as described in its website, a general processing engine that can perform batch processing as MapReduce, but it has the advantage of also supporting stream processing.

1.1.2 Motivational Example

The benefits of bearing the cache features in mind when developing applications can mean better performances, mainly through the exploitation of the locality principle. To make this concept clearer, we present a simple example of a situation where the use of a strategy that takes into account the cache features brings some benefit to the application.

We draw our example from the stencil computations that are important in several scientific fields, such as (a) image processing and scientific computations where the vector calculation of two or more dimensions is particularly important for the extraction and analysis of scientific data, and (b) for applying image manipulation masks. Generally, this kind of computations calculate the new position vector by making operations that are directly dependent of a given element's neighbourhood. These calculations have the particularity of having (often) temporal locality properties, by revisiting data over the computation. Examples of stencil computations can be found in some scientific computations such as the SOR and Jacobi method, used to solve linear equations, but also in simple image processing techniques such as a image blur. To better understand the usual behaviour of a stencil computation, consider a simple stencil example that transforms a given integer matrix M by applying the following formula:

$$M[i,j] = M[i,j] \times percentage + \sum_{ni=-r}^{r} \sum_{nj=-r}^{r} M[i+ni, j+nj] \times \frac{1 - percentage}{(2r+1)^2 - 1}$$

where i and j denote the index of the elements in the matrix, r denotes the range of the stencil and $(2r+1)^2 - 1$ gives the number of neighbours of the element to be transformed (denoted by M[i,j]). For the sake of readability we only consider the indexes inside of the matrix limits, the remainder are simply discarded.

In this formula it can be seen that the stencil has two major components, the element to be transformed or *mainElement* component (in the first part of the addition) and the

neighbourhood component. From the first component we obtain the contribution of the *mainElement* in the computation of its new value, where the *percentage* variable indicates the percentage used of the old value. In the second part of the formula, we have the sum of the contributions of neighbourhood values to the result of the *mainElement*, where these contributions depend on the remaining percentage divided by the number of neighbours. The neighbourhood of each element is defined by the elements within a range *r* of the *mainElement*.

A more concrete example can be found in a situation where we have a 6x6 matrix that is submitted to our stencil which works with a range *r=1*. This means that each elements is dependent of its immediately adjacent neighbours. If each element contributes with 50% of its own value to the final result, the remaining 50% is divided by the number of immediately adjacent neighbours and it is used to get the neighbours contributions. With this set-up and considering the element at the (0,0) matrix position, then we can say that each neighbour of that element will contribute with around 16.67% of their values.

Consider this stencil example in a MapReduce context. To express stencil computations with the current MapReduce model, we have to decompose these computations into the map and reduce phases of this model. As mentioned earlier in this section, to compute the new value of a given element we must have all the elements that compose the element's neighbourhood. However, the most common way to receive the input in MapReduce is in the form of a line of data, which can be a simple line of text, a line of numbers or a line of other data type. In either case, a line of data for the majority of stencils does not contain all the elements required for a element's computation. For example, considering an input matrix, the input of the *map* phase will probably be a matrix line that limits the work in this phase. This approach lead us to the regular model for MapReduce, where the *map* phase receives a given input and, after some optional pre-processing, distributes the input data by the *reducers*, in order for them to perform the actual computations. The left-hand side of Figure 1.1 depicts the execution of a MapReduce stencil computation where the *mapper* receives as its input a matrix line. With this line, the *mapper* determines to which *reducers* are sent the line's elements and finally sends them. The part of determining to which *reducers* an element has to be sent is related to the "construction" of a element's neighbourhood. For example, considering this situation for a given element *A*, the *mapper* must send this element to the *reducer* responsible for its stencil computation, and to send this element to the *reducers* which are responsible for the elements of *A's* neighbourhood, so that the neighbour's computations can be completed. The *reducers* have the task of receiving an element and its neighbourhood, and apply the stencil to the received elements to produce the final results.

As in many MapReduce applications, a stencil computation has its number of partitions predefined (may have been set by the programmer) without any regard about the memory hierarchy. A consequence that may arise from this predefined number of partitions is related to the size of each partition, which can be too big for the cache and, hence, can lead to workers of different cores competing for shared cache space. This

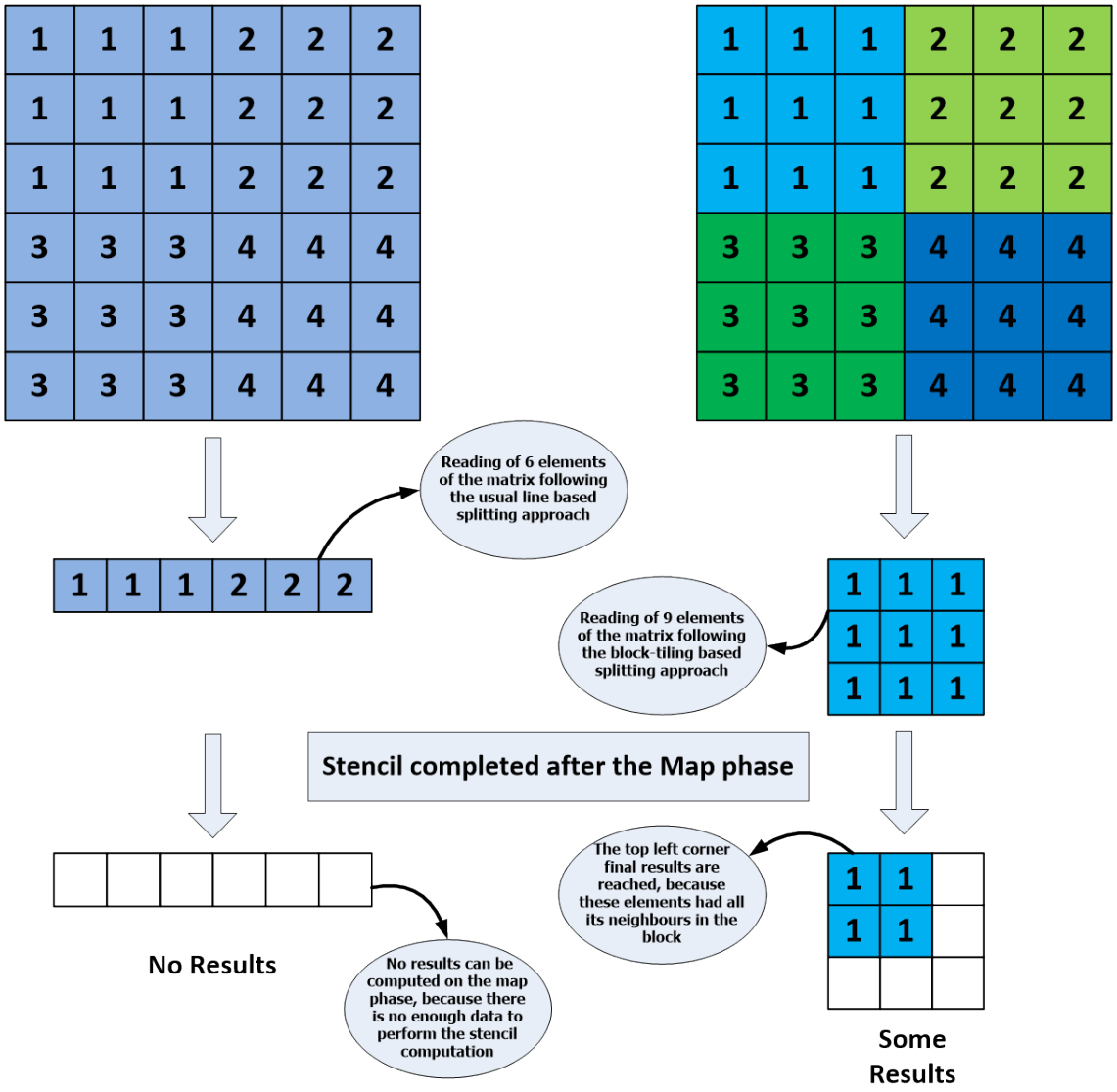

Figure 1.1: Simple stencil with range = 1. Comparison between using lines (on the left) and using blocks (on the right) as input in the *map* phase.

situation can be disregarded if the computation does not make use of temporal locality, as in situations such as sequential iteration of an array or reading a file. However, this is not true for stencil computations that iterate multiple times the same data. In this example, the temporal locality is of great importance, because, as its definition says, there is a need of reusing data in a short span of time. If the input matrix data cannot remain long enough in cache because of its size, then this can result in a higher number of cache misses. This indicates that the cache hardware is not being fully used and consequently the computations are slower.

There are techniques to improve temporal locality in these situations, which are called blocking/tiling strategies. In our stencil example, blocking/tiling can be applied to keep

the matrix data longer in cache by dividing the matrix into smaller blocks/tiles, as depicted in the right-hand side of Figure 1.1, allowing more reuse of data. A measure that can improve further more this optimization, is the inclusion of cache related information in the process of choosing the block/tile size. Adapting the size of the blocks/tiles to the cache dimension can lead to better temporal locality by reducing the eviction of data from cache, derived from data location conflicts.

1.2 Problem

The efficient mapping of MapReduce tasks onto the memory hierarchy of a given CPU is a problem with some complexity that if treated correctly has the potential to be rewarded with significant performance gains. This problem is mainly caused by the lack of support of MapReduce for cache friendly strategies, by that we mean that this model does not take into account any cache feature that could be used to better map its tasks onto the memory hierarchy.

The efficient mapping of computations onto the underlying memory hierarchy is a concern that has driven a considerable amount of research. The proposed strategies may be divided into the following three groups: (i) loop transformations [10, 24] which consists of loop manipulation at the compiler level to improve cache locality and performance of a sequential or parallel programs; (ii) cache-oblivious algorithms [21] which try to improve the mapping of applications to memory hierarchy by determining the workloads and data movements of cache levels without depending or taking into account any cache feature (cache size, cache-line length and others) to tune the application for better performance, and; (iii) explicit programming of the memory hierarchy [18, 20] which abstracts the memory hierarchy of some given machine for the programmer, in order to let him/her choose where the computations are to be done in the hierarchy.

None of these strategies can be seamlessly integrated in the MapReduce execution model. The loop transformations are confined to loops, and parallelism in MapReduce computations are not, normally, found at the cycle level, but in the parallel implementation of a computation on disjoint partitions of input dataset. The same goes for the cache-oblivious and explicit memory hierarchy programming. These two apply divide-and-conquer strategies in which the domain partition is close tied with the expression of the computation. Moreover, both these approaches require modifications in the application's original code, which is something that was out of this thesis context.

To address this application mapping onto the memory hierarchy in the context of MapReduce, it is important to understand or at least know the two main phases of this programming model, being these the *map* phase where a computation is applied to the multiple partitions in parallel, generating some results, and finally the *reduce* phase that take the results of the *map* phase and reduces them into a final result. Although it is not considered a main phase of MapReduce, the *split* phase is of great importance and should

be taken into account, mainly because its where the input data is decomposed into a well defined number of partitions.

Besides the poor use of cache friendly strategies, another problem related with the MapReduce model has to do with lack of support of some computations that naturally have the tendency of benefit from temporal locality. Examples of this kind of computations are the matrix multiplication or even the stencil computations. The matrix multiplication is a good example of the excessive complexity inherent in the development of an application with MapReduce, where the programming model does not express this computation as well as the regular iterative version. The iterative version of the matrix multiplication only requires some loop instructions to express the computation, where its contra part requires the implementation of special classes and strategies only to adapt this computation to the MapReduce model. Regarding the stencil computations, they can be adapted to the MapReduce model with some effort. However, the incapacity of performing computations in the *map* phase, as explained in Section 1.1.2, makes this model unattractive to express these computations. The main reason for this is in the way that the two main phases are executed, where the *map* phase is mainly performed in parallel, but the *reduce* phase performs its computations in a sequential way. With this, it becomes clear that the lack of computation in the *map* phase can become a constrain when implementing stencil computations with MapReduce.

In order to reach a solution that is based on a in-memory, cache optimized MapReduce, some changes have to be performed in the *split/map* phase. The reason why this phase is chosen to be modified is related to its function, namely the decomposition of the domain that greatly influences the mapping of the subsequent tasks. On the other hand, the lack of support for computations which benefit from temporal locality can be solved by the creation of a new layer that supports computations such as the stencil computations.

The main challenges that arise from a cache optimized MapReduce which gives some support to stencil computations, are:

1. – The implementation of a cache friendly *split* phase

2. – In-node work-flow modification, transferring more computations to the *map* phase

3. – The development of support for stencil computations as an API

The implementation of a cache friendly *split* phase must handle data in a manner that fits the cache dimensions, either in the situation where it receives data or when it produces results. The modification of in-node work-flow its performed by the inclusion of more computations in the *map* phase. In turn, this transference of computations is closely related with the modifications made in the *split* phase and it is used in the development of a stencil API, which abstracts many of the complex strategies used to express this computations in the MapReduce model.

Some of these challenges are addressed by frameworks, such as Tiled-MapReduce [11] that take advantage of the dimensions of the memory/cache hierarchy by partitioning large jobs into smaller ones to get better locality and less contentions, in which the problem of choosing the size of the partition/tile is present. The Metis platform [25] that uses some data structures to handle different workloads of MapReduce or Phoenix [31] that also tries to approach an in-memory MapReduce, being one of the first attempts.

Even though these are in-memory MapReduce approaches for multicore machines, none of these by itself addresses all the challenges. There are some efforts to take advantage of better data locality, but neither one does a cache-conscious decomposition. By that we mean that these approaches do not take into account the depth, capacity and organization of the cache hierarchy. Hence, there is some space for improvement at the data and computation decomposition, taking into account the cache features and at the same time keeping these burdens out of the programmer shoulders.

The same can be said about the support of stencil computations which is currently non-existent. Therefore, development of such support can be relevant not just for the creation of the support itself, but also to test our strategies that try to take advantage of predisposition to temporal locality of an application.

In this thesis we propose an approach that delegates the responsibility of distributing the application across the memory hierarchy to the runtime system in a in-node context, in order to optimize the main phases of MapReduce. And also the implementation of an API which supports the development of stencil computations in a more programmer friendly way.

1.3 Proposed Solution

As said in the previous sections, both the lack of support of strategies that take advantage of the cache memory hardware and also that benefit from concepts such as temporal locality are relevant problems in MapReduce frameworks, such as Hadoop [2]. This framework has been one of the most used implementations of this programming model, either for applications designed for clusters system, where its use is more predominant, or for single nodes.

With that said, in this thesis it is proposed a solution that helps to integrate block-tiling strategies in the programmer's Hadoop MapReduce application, where the dimensions of the blocks produced by this decomposition are influenced by some cache hierarchy characteristics. This integration is mainly made by the extension of abstract classes that are offered to the programmer so that he/she can express the decomposition logic of its application's data. These abstract classes are related to the *split* phase, where besides performing the normal problem's domain decomposition that is based in the number of processing units, it also performs a second decomposition at the in-node level. This second decomposition controls the size of the tasks by taking into account the cache characteristics to decompose the received data blocks into smaller tasks. These modifications

also affect the *map* phase, bringing more computations to this phase which means more parallelized computations. With more computations being performed in the *map* phase, the *reduce* phase that is completely sequential gets less work, which can lead to better performances.

Another aspect of the proposed solution is related to the validation of the proposed decomposition that, as said in the previous sections, can benefit some applications that have some temporal locality. Given that, to validate our decomposition solution and to verify how applications that benefit from temporal locality are influenced by it, this document presents the integration of a Hadoop MapReduce API for the development of stencil computations in this programming model. This API is composed by some classes that try to abstract some concepts of the implementation of stencil computations in MapReduce, in order to make this task less complex for the programmer. With this API are also integrated some optimizations that aim to leverage the temporal locality principle in this kind of computations. The creation of a stencil API widens the type of applications supported by Hadoop Mapreduce, and provides us applications in which the increase of computations in the *map* phase is easier to perceived.

To sum up, this thesis tries to show that it is possible to obtain performance gains in a programming model with two main phases, such as MapReduce, that decomposes an application and maps it onto the memory hierarchy in an cache friendly approach, with little or no special effort for the programmer. Validating these concepts by the development of stencil applications with an equally programmer-friendly API.

1.4 Contributions

The contributions of this dissertation are the following:

- An integration of block-tiling decomposition strategies in Hadoop in an in-node context.
- Use of the cache memory hierarchy features to define the block/tile size, with little or no programmer intervention
- Assessment of the usefulness of our approach by applying it to the context of stencil computations. For that purpose, we designed and implemented an API for stencil computations to Hadoop MapReduce.
- A comparison between the performance of the original Hadoop and the performance of the one implemented by us.

1.5 Document Structure

The remainder of this document is organized as follows:

Chapter 2 – Presents the state of the art with regard to MapReduce frameworks, more specifically to the Hadoop MapReduce and other frameworks linked to the in-memory and multicore context. Besides that, are presented some strategies related to cache optimization;

Chapter 3 – Introduces our decomposition solution integrated in Hadoop MapReduce, including the theoretical solution, programming model and an example;

Chapter 4 – Specification of the MapReduce solution for stencil applications, showing our Hadoop API, the programming model and a implementation example;

Chapter 5 – Presentation and discussion of the result of the performance evaluation;

Chapter 6 – Closes this thesis by exposing our conclusions about the work accomplished and with some suggestions of possible future work in this topic.

Chapter 2

State of the Art

In this chapter we explore the state-of-the-art of MapReduce-based computing. Starting with the description of the programming model and execution of this framework, followed by the presentation of Apache Hadoop, the evaluated framework that implements a form of MapReduce. Finally, we discuss some works in the field of in-memory MapReduce.

2.1 MapReduce

2.1.1 Programming Model

The MapReduce model can be viewed as a pipeline of data-processing phases, two of which - *map* and *reduce* - have to be user-defined. Both the data source and sink assume the form of files as depicted in Figure 2.1, a simplified version of the execution pipeline. Additionally to *map* and *reduce*, the MapReduce pipeline comprises other phases, for which the framework provides implementations by default (that may be overridden by the programmer).

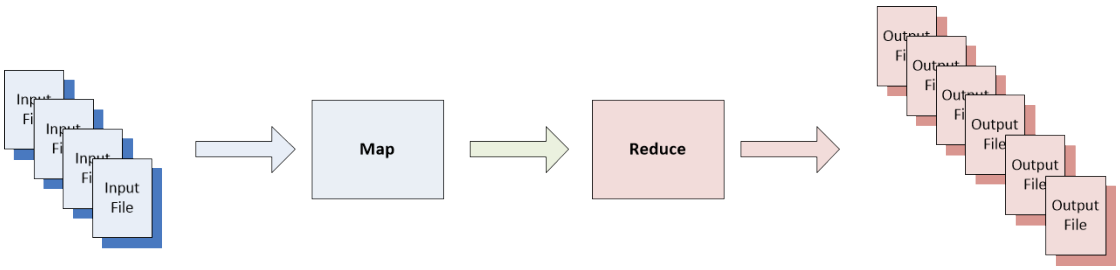

Figure 2.1: Simple MapReduce data flow.

The MapReduce programming model is based on two user implemented functions,

the *map* and *reduce* functions. These two functions are respectively associated with the *map* and *reduce* phase and are the core of the user defined behaviour of the MapReduce model, where the prototypes of these functions are presented in Table 2.1. Both functions manipulate key/value pairs of user-specified data types. In particular, the *map* function receives as input one of such pairs and returns a set of pairs, entitled *intermediate* pairs. The results generated by the application of the *map* function are grouped together by a common intermediate key, generating pairs (key, list of values) that are passed to the next phase. In turn, the purpose of the reduction function is, as the name implies, to apply a reduction upon the received list of pairs (key, list of values), so that it produces a single value for each pair received (key, list of values).

$$map < IK, IV, OK, OV > (IKin\_key, IVin\_value) \longrightarrow List < OK, OV >$$
$$reduce < OK, OV > (OKintermediate\_key, List < OV > out\_values) \longrightarrow List < OV >)$$

Table 2.1: Map and Reduce function prototypes.

Both the input data and the final results are held in separate files, each one of the latter related to a *reduce* task. One way of controlling how many result files are created is to set the number of *map* and *reduce* tasks, respectively M and R.

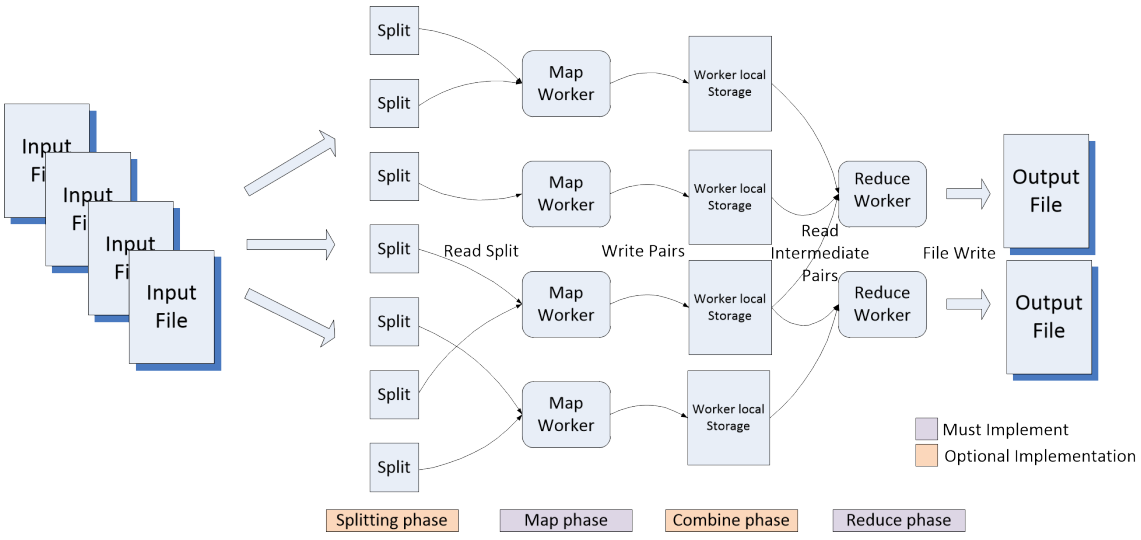

Figure 2.2: MapReduce pipeline.

Figure 2.2 presents a more comprehensive version of the MapReduce pipeline, featuring all the user-controllable phases. The first phase that can be overridden is the *split* phase. The default logic of the *split* phase is to divide the input data, which are files in a specified directory the file system, into multiple partitions. Also this phase stores each of the produced partitions on separate files that are subsequently fed to the *map* phase. To control the behaviour of this phase, the programmer can specify a custom partitioning function, which is the center of the splitting action and the true responsible for how the

split is accomplished. One related settable option is the number of data chunks that this function produces, meaning that the programmer can specify the number of input data chunks that are at the disposal of the *map* tasks.

Another phase whose behaviour may be redefined by the programmer is the *combine* phase. This intermediate phase occurs between the *map* and *reduce* phases and has the objective of merging (or combining) the intermediate results that flow from the *map* tasks to the *reduce* tasks. To specify the behaviour of this phase, the programmer must define a *combiner* function that allows a partial merging of repetitive keys that come from the *map* phase. An advantage provided by this phase is that it prevents sending the same key/value pair multiple times to the *reduce* phase. Often this function shares the same code with the *reduce* function, but they are used in different contexts. The *reduce* function output is written to a final output file, while the combiner function writes the output to an intermediate file that is sent to a *reduce* task.

2.1.2 Generic Execution Model

As previously said, the focus is in the in-node execution model of MapReduce and in optimizations related to memory management. However, in order to better contextualize our work, we provide an overall description of the entire MapReduce execution model.

The model in Figure 2.2 may be implemented in several different ways, depending on the application's purpose and the target hardware. In this section we focus on the original model, defined in [16] - the base upon which the others have built. Therefore, as mentioned above, this section is dedicated to the description of the general execution model of MapReduce, describing briefly its phases.

The execution model starts by applying the split phase to the input data. The result is stored in M chunks of data for consumption by the subsequent *map* phase. Before the *map* phase can begin, the system has to decide which workers of the cluster will receive *map* or *reduce* tasks, wherein all the workers in the cluster are responsible for a chore in the execution model. In regard to this task assignment process, the original paper [16] does not mention how the roles are distributed in order to take advantage of the multi-core resources that are available. This can be done in two different ways. A task can be divided into sub-tasks in the node taking in account the available resources or the assignment can distribute multiple tasks for each node. As soon the task assignment is over, the *map* workers start reading the input data that has been appointed to them, this marks the beginning of the *map* phase. Finished the data reading process, each map worker proceeds to parse the data forming key/value pairs, which are passed to the user implemented *map* function. The intermediate results produced by the *map* function, which are key/value pairs, are usually stored in memory and periodically persisted in a file system. The file system can be either local to a given node or distributed by multiple nodes. When these results are written, the worker informs the system about their location, so that these ones can be assigned to the *reduce* workers. Before the intermediate results are persisted in the

file system, the workers apply the combine phase. This reduces the amount of repeated key/value pairs that has to be transferred across the network.

As the *map* tasks complete, the *reduce* workers launch remote procedure calls to the location of the intermediate data to retrieve it, beginning the *reduce* phase. Once the data is read, the *reduce* worker sorts locally the retrieved data by their key to get the values with the same key grouped together. The reason for this sorting is that the values with a certain key have to be sent to the same *reduce* task. Next, the worker iterates the intermediate unique keys to obtain the values for each key. As it iterates these unique keys, the worker passes the key and its values to the user defined *reduce* function, that starts producing the final results. In the original implementation instead of passing the entire list of values, what is actually passed is an iterator for these. The reason for using an iterator instead is because of the possibility that these values can be too big to fit in memory. After the data is processed in this phase, the results are written in one of the R output files. Typically, these results are written in a output file corresponding to the processed partition by a *reduce* task.

Finally, when all the workers have finished their work, the system returns the execution to the user program. The result of the MapReduce computation is a collection of R files, one per *reduce* task, that can be used by another computation or another distributed application.

2.1.3 Apache Hadoop

Apache Hadoop [2] is a distributed data processing framework developed for the purpose of running applications on clusters of low-cost machines. This framework offers reliable, fault-tolerant and high throughput data flow to its client applications in a transparent way. Hadoop offers one of the more popular implementations of the MapReduce programming model [5]. This component is responsible for parallel and distributed data processing of large data-sets using the cluster of resources to execute the two main phases of this programming model, *map* and *reduce*. Besides the MapReduce implementation, Hadoop has more components such as its own distributed file system (HDFS [8]) that provides the storage for the application data in the cluster nodes. It also provides YARN [36], a job scheduler and resource manager for the cluster. In this section is presented an overview of this framework, discussing its architecture and some of its more relevant components to this context.

2.1.3.1 Overview of the System

Hadoop can be viewed as a three tier framework as depicted in Figure 2.3, with the lowest tier being the HDFS as the borderline with the cluster hardware. On top of HDFS is YARN the resource manager, and on top of YARN there are data processing components, one of them being MapReduce.

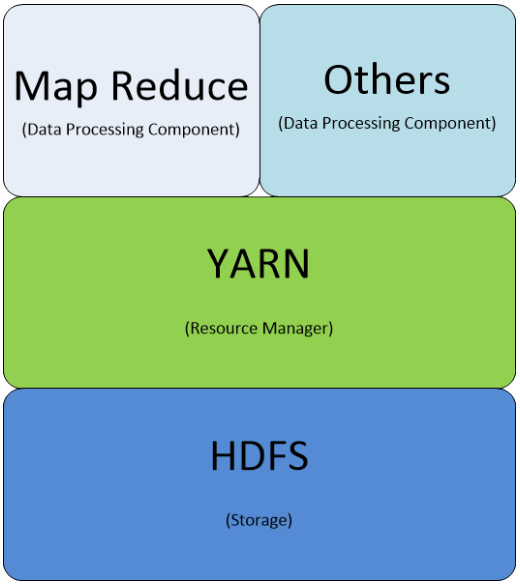

Figure 2.3: Hadoop tier architecture model.

HDFS [8] is a distributed file system based on Google File System [22] that was developed to support Hadoop. As a distributed file system HDFS aims to run on hundreds or more nodes, but it is distinguished by the fact that it does it on top of low-cost hardware. Fault tolerance is a important goal of this system, mainly because when targeting clusters of large dimensions the probability of machine failure increases, requiring so the ability to support the problems that this failures provoke such as data loss. This file system has some characteristics that are relevant to mention, namely the accessing and processing of the data comprised in such files. For starters HDFS supports large files, typically from gigabytes to terabytes in size being ideal to applications with large data sets. A feature that fits MapReduce applications is the handling of files that are accessed in a write-once-read-many model, simplifying the data coherency process. Another very important feature of this file system is the way computations are carried out. Instead of moving data from node to node, HDFS moves computations, as it is more efficient in a system with large datasets to execute the computations nearer to the data than to move it to a specific node. With this approach the system can minimize network congestion and increase the throughput of the processing. For example, in a MapReduce program the *map* tasks are assigned to nodes that store the data relative to that tasks. However, yet in the MapReduce case, there are some transferences of data, mainly when the *reduce* task fetch the key/values pairs that have to process from another node.

As stated earlier, YARN [36] is Hadoop's resource manager. YARN was developed to decouple the programming model of data processing components from the resource management of the system. The latter has the responsibility of managing and monitoring the workloads and data access of data processing components such as MapReduce. This task was previously a responsibility attributed to the data processing components.

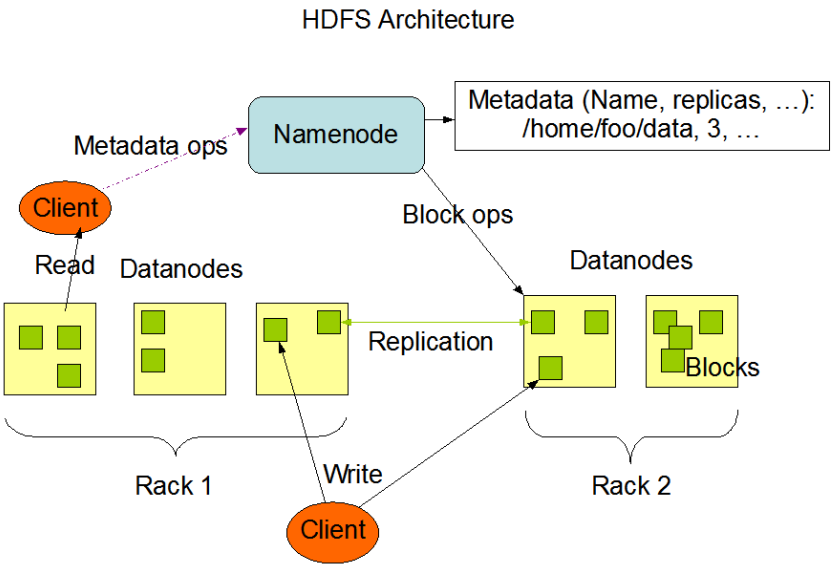

Figure 2.4: HDFS architecture from [8].

The MapReduce implementation provided by Hadoop is targeted to clusters environments. The execution model is very close to the original [16]. It is worth to mention that this is the only communication step in MapReduce, neither the *map* tasks communicate with each other, nor the *reduce* tasks. When the worker receives the intermediate pairs, it sorts them to feed them to the *reduce* task. Finally, the *reduce* phase starts and the intermediate pairs are passed to the user defined *reduce* function that processes this pairs and write them in the output in its local HDFS.

2.1.3.2 Splitting

In this section we describe how the splitting phase is done in the Hadoop framework, diving in the programming interfaces and relating them to the different steps of this phase. The splitting phase is reviewed in more detail mainly because it is one of the points in which our work is focused, as it is the case of the in-node execution model. Being the major objective adapting this two in order to optimize the memory management advantages that a cache support may bring to this programming model. This phase is of immense importance not just for the load balancing of the work attributed to *map* workers, but for the data processing efficiency in general.

In the Hadoop framework the HDFS divides the input data files into multiple data blocks that are stored in the work nodes. Each of these nodes has stored locally different data blocks to be processed. When all the nodes have the data blocks in theirs file system, they start the MapReduce job by the splitting phase. This phase is done in each node individually and the input data blocks stored locally are divided in multiple chunks that are fed to the *map* workers as tasks.

To fulfil this splitting task there are three major classes in the Hadoop's MapReduce

implementation, the InputFormat class, the InputSplit class and the RecordReader class. These classes base their work on data read from input files that generally reside in the HDFS.

The InputFormat class is responsible for how these input files are split. This class can be one of three types, TextInputFormat, KeyValueInputFormat or SequenceFile InputFormat. These three classes are differentiated by the way that they handle the split of the input files and how they define their types of key/value pairs. Hence, the InputFormat class selects the input files from a user specified directory and according to the split type and files size, defines the splits that break this input files into chunks that are fed to tasks. These splits are merely logical and the resulting information from these is stored in another class, the InputSplit class. Thereby, the input files are divided into one or more InputSplit instances, according to their size and on the type of splitting specified by the InputFormat. So the InputSplit class describes a *map* task in a MapReduce job as described in [37]. After the splitting as occurred, the InputFormat defines the tasks that compose the map phase and these are assigned to the workers based on the location of the input files chunks, without taking into account the resources of the node. Consequently, a worker may have many tasks assigned to it, processing them in parallel as much as it can. A worker accesses the data associated to a given task via the RecordReader class, which converts the original data described by an InputSplit into key/value pairs.

2.1.3.3 Node Execution Model

A node stores locally chunks of the input files to be processed. As mentioned in the previous section each node takes some *map* tasks that are directly related to the parts of the input data stored in the node file system. These parts of input data files are used by the InputFormat class to do the splitting process and create the data splits. After this splitting process is done in the node, as depicted in Figure 2.5, calls an instance of RecordReader for each split and starts reading the data and producing the key/value pairs that are fed to the *map* tasks. The number of *map* tasks in a node can be defined by altering the mapred.tasktracker.map.tasks.maximum global parameter, mainly to control the level of the node parallelism of the system.

The *map* tasks in the node process the pairs produced by each RecordReader and produce intermediate key/value pairs that are passed to a Combiner, which can be a instance of the Reducer interface, to combine the intermediate results produced by a *map* task. When the intermediate reduce is done, the Combiner passes the resulting key/values pairs to a Partitioner. The job of the Partitioner is to exchange the produced pairs with other nodes in the system so that these pairs can be processed by the correct *reduce* tasks, as can be seen in Figure 2.5. When this exchange of pairs is over the node starts shuffling the pairs to be fed to its *reduce* task, so that this task can be executed as sequentially as possible. Finally the *reduce* tasks starts to produce the final key/value pairs and writes them back in the node local file system.

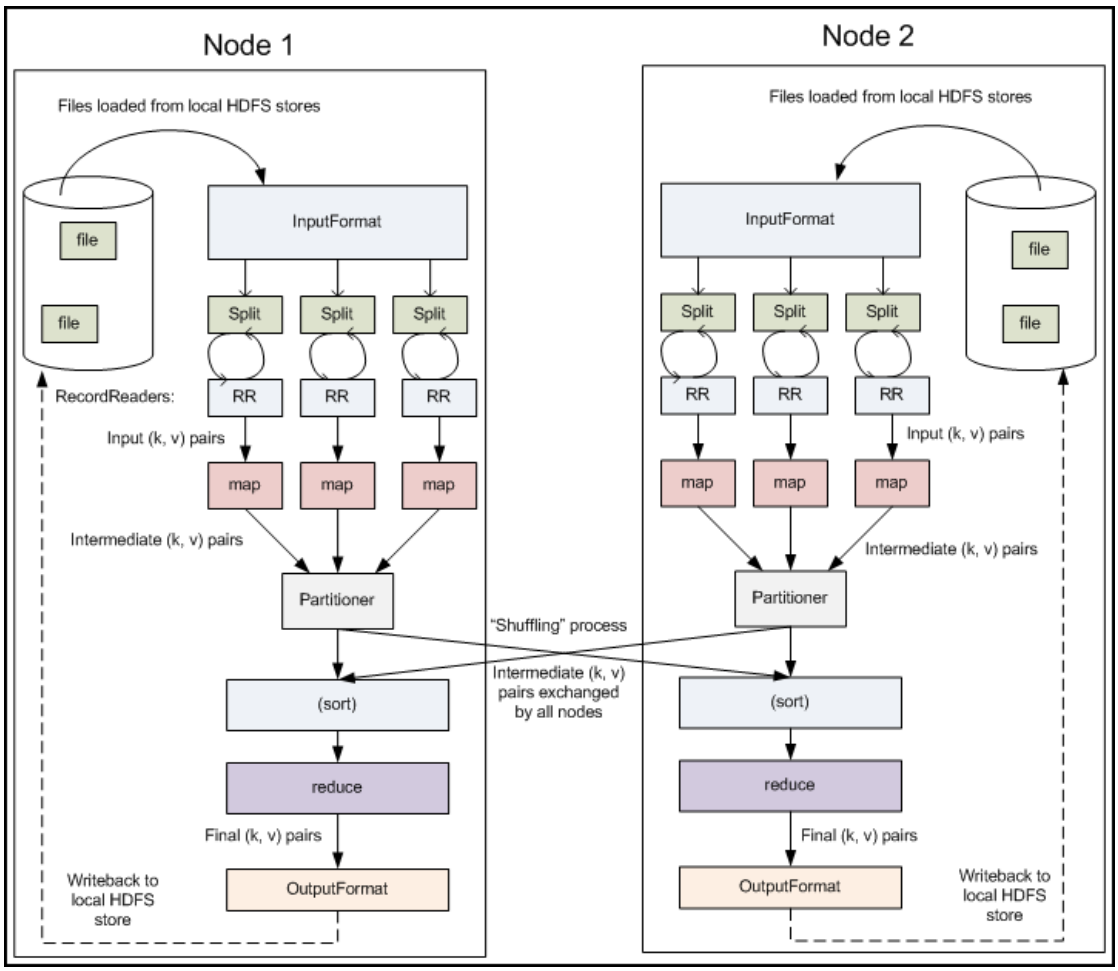

Figure 2.5: Hadoop in-node Execution Model from [37].

As can be seen in Figure 2.5, a node is responsible by doing all the MapReduce flow on the data stored locally. For this, it uses its parallel computation capacity to execute as much splits and consequently as much *map* tasks as possible.

2.1.4 In-Memory and Multicore MapReduce

As stated previously in this document our main focus is in the in-node execution of MapReduce and for that it is crucial to understand and review the concepts behind some frameworks that perform memory related node-level optimization. These are commonly referred to as In-Memory MapReduce, such as [34] and [33], or are pure multicore-directed implementations of the model, such as [11] and [25]. In either case, these frameworks share a common objective: to optimize the execution of MapReduce model in a single machine taking advantage of its resources, such as the multiple core power and the storage capacity of memory. To achieve this goal several distinct strategies have been proposed that have some basic common ground, namely distribution ruled by the number of cores and usage of shared memory mainly for intermediate results. For the rest of this

section we will discuss the main challenges addressed by these In-Memory and multicore directed MapReduce implementations and how they are tackled.

2.1.4.1 Data Structures

A point that has been subject to great scrutiny and therefore subject to some improvements has been the data structures that store the key/value pairs produced by the *map* tasks. These data structures can be shared by multiple workers or be private to a single one. In the first case of shared data structures some problems can arise, probably the most evident are the classic problems from parallel/concurrent programming, data-races and deadlocks. Hence to avoid this type of problems it is common to use individual data structures for each worker. With these, a worker can handle its data without the complexity of locks and synchronization through messages with other workers.

A framework that uses the multicore MapReduce and per worker data structures for intermediate key/value pairs is Phoenix [31]. Phoenix has a simple runtime, it attributes a worker for each available core, assigns the tasks to these workers after the splitting of the input data, sorts the intermediate pairs produced by the *map* workers before passing them to the *reduce* workers, applies the *reduce* function and then merges the *reduce* results sorted by key. This framework uses per worker buffers, to accommodate the intermediate pairs, that are initially sized with a default value that it is updated as needed. Each of these buffers are described in [25] as cells in a matrix where a row of these are associated with a split and the columns refer to the pairs of entries with the same key hash function result. This column strategy has an advantage for the *reduce* tasks, with it a *reduce* worker has only to read columns of the buffers of a *map* worker, simplifying the communication of the pairs from *map* to *reduce* task. Each one of these buffers are hash tables sorted by key that have as values a list of values of each pair with a given key. This solution has some computational costs. The cost of hashing a key is always $0(1)$, but the cost to calculate the correct column of the hash table can be $0(\log k)$, being k the number of distinct keys in a entry if the table already has a entry for it, otherwise the cost becomes $0(k)$.

The costs of these data structures can influence the performance of a MapReduce program, so it is important to choose carefully the trade-off of each data structure. Choosing a data structure can be largely influenced by the properties of a given problem, such as workloads. For instance in the solution provided by Metis [25] they conclude that there is no single data structure that always get good performances regardless of the problem's workload. Three different data structures are described. A similar approach to the one implemented in Phoenix is an hash table per *map* worker that is used to store the produced pairs. All the hash tables have the same size and the same hash function to facilitate the partition of the work tasks for the *reduce* workers. For this hash table the entries are composed by a hash of pairs of the form key/value as key, and a list of these ones as values. This data structure can have a low computational costs of $0(1)$ for inserting and

looking up entries, being that these costs are only possible if the hash table has enough entries so that the probability of collisions is low, which makes it suitable for workloads with many repeated keys. Another data structure mentioned is a append-only buffer of key/value pairs that is sorted before the *reduce* workers start reading the contents of this buffer. The most expensive operation over this type of data structure is the sorting, whose cost is proportional to the number of pairs produced. The weight of the operation is most noticeable when the *map* worker produces too many repeated keys, being that the data structure loses the advantage of producing key sorted results. Hence the append-only buffer could be a good data structure for cases that have workloads with few repeated keys. The third option is a tree that uses the keys as index. With a computational cost for the lookup of $O(\log k)$(k being the number of keys), this data structure is suitable for workloads with repeated keys and with an unpredictable number of keys.

The Metis platform [25] implements a conjunction of the hash table and tree data structures, with the goal of performing good for a wide range of workloads. To that end, it combines this hybrid implementation with a prediction phase, that through some heuristics based on the number of keys of a portion of the input can reach good performances for a wide range of workloads. The produced data structure is an hash table with a b+tree in each entry, to achieve the best of both data structures in different situations. When the number of keys is predictable and there are enough entries to avoid collisions, the costs of $O(1)$ of the hash table will be applied. If the number of predicted keys is not large enough or the keys do not hash uniformly, which implies large b+trees in hash table entries, then the costs of a tree are applied $O(\log k)$ (k being the number of keys).

2.1.4.2 Workload Optimizations

From the previous subsection, can be concluded that the workload highly conditions and influences the performance of a MapReduce program in a single-node. The workload determines both the data structures to be used, which can perform better or worse depending of its computational costs, and the ability of expressing efficiently a given problem in a shared-memory MapReduce model. In [34] the authors characterize a workload according to three characteristics that relate *map* tasks to intermediate keys:

1. – the *map* task to intermediate key distribution,
2. – the number of values emitted per key,
3. – the amount of per task computation.

As previously mentioned, the key distribution affects the intermediate storage of the key/value pairs produced by *map* tasks. This is easy to see in some distributions where a hash table can get better costs than a tree and vice-versa. The number of values per key affects directly the implementation of the combine function that processes the intermediate pairs for a reduction of the number of pairs passed to the *reduce* tasks. Depending on

the characteristics of a problem the combine function may have simpler or more complex implementations that can cost more or less memory. With this consumption of memory allocation the advantage of having a combine function that reduces the communication of pairs may be concerned. The third characteristic mentioned to define a workload refers to the overhead that a library of a framework may impose, in that it forces the user to use bad, or at least not ideal strategies for a problem.

The same work [34] proposed Phoenix++, a rewrite of the Phoenix framework, which enhances some aspects related to the described characteristics of a workload. The proposed solution grows from the concepts of container and combiners. Containers are a way of adapting the framework to various types of workloads and combiners allow the program to handle workloads with high number of values per key. These two improvements allow to address the problem related to the cost of a fixed data structure and the problem of memory allocation. A container is a group-by functionality that it is present between *map* and *reduce* phases and groups emitted key/values pairs by key, much as a data structure, that are passed to the combiners. Phoenix++ supplies 3 container implementations to meet the different workloads. These implementations include a variable width hash table for each *map* that it is present in the hash container, ideal for workload with previously unknown number of key. A fixed-size array per worker with the requirement of the keys being well known integers is present in the array container. And lastly a non-blocking array shared by all workers, to workloads where tasks only output a single, unique key. In this framework a combiner is not just a function, but an object that is invoked every time a key/value pair is emitted by a *map* task. If this invocation is not done at every emit, a problem related to the combine invocation at the end of a *map* task may arise, mainly because of the great memory allocation that is caused by the produced key/value pairs that have to be stored. At the end of the *map* task when the combiner is executed the pairs may no longer be in cache, meaning that it has to access to a higher memory tier which brings more costs. This framework already has available two implemented combiners, the buffer combiner and the associative combiner. The buffer combiner is a approximation to the standard in the MapReduce, it buffers all the produced pairs until the end of the map task. The associative combiner as the name indicates aggregates all the emitted values into a single one, meaning that when a new value is received it is combined with the previously received an therefore already aggregated.

2.1.4.3 Memory Management

Another challenge to address in the efficient execution of MapReduce computations on a single computational node is the memory management. Bad memory management in this programming model normally comes from large data-parallel applications that tend to use a lot of tasks, which can lead to pressures that not only affect memory itself but also CPU utilization and consequently can lead to poor performances. A situation that contributes to this problem and it is present in some frameworks that use In-Memory

multicore MapReduce, such as Phoenix [31], is the persistence of intermediate data in memory not just until the tasks are finished, but until all the processing is finished. This problem is more evident in applications that are more memory-intensive and, to deal with this, some solutions propose the sharing of resources by the tasks, to ease the allocation of memory or even to take advantage of cache/memory locality. One solution that tries to take advantage of cache/memory locality is the Tiled-MapReduce [11]. This solution extends the basic MapReduce with a tiling strategy whose main premise is to reduce the consumption of resources by partitioning big jobs into smaller ones that are processed sequentially. Hence, these small jobs only require fractions of the resources that a bigger job would require and being sequential executed the allocation of memory for intermediate data structures is done only once, allowing the smaller jobs to share the data structures. Besides that, due to the small size of the jobs, the input is obviously smaller and this brings the opportunity to explore the principle of locality inherent to memory hierarchy.

Another measure described in [11] is the reduction of the idle time of the multiple cores. This is described as a problem that comes from the original implementation of MapReduce, in that a *reduce* phase must wait for a *map* phase to end, and just then the *reduce* phase may start doing useful work. To treat this problem Tiled-MapReduce overlaps a *reduce* phase from a small job with a *map* phase of the next small job to be executed. This not only reduces the idle time of the working cores but also improves the performance of the "bigger" job. To get these solutions the data-flow of MapReduce in Tiled-MapReduce is a little different from the original. This data-flow is based on multiple iterations that represent the small jobs and are compose by consecutive *map* and *combine* phases, the latter behaving like a intermediate *reduce*. Is in these iterations that lies the advantage of this solution, because it allocates memory for a common intermediate buffer such as the described in Phoenix [31] at the first job. At the same time it allocates memory for a more specific, but equally shared iteration buffer that harnesses the intermediate results of the iterations that are used by the next *reduce* phase. After these smaller jobs are done, it takes place a "global" *reduce* phase and then the merge of the results.

A similar problem related to the management of memory is described in [39], where the allocation of a lot of Java Virtual Machine (JVM) per machine, one per task, in the Hadoop MapReduce [2] results in bad memory utilization. This happens mainly because of the duplication of data structures across these JVMs. Another problem that is described and it is related with the use of these JVM, is the way key/value pairs are processed by the *map* function. This function is handle by a JVM, as it exists one per task, and just processes one pair at a time, which means that to start processing a new pair the task must have already processed the previous pair, implying a sequential pair processing flow. These problems affect both the memory allocation and program performances, causing unnecessary memory pressures and do not take advantage of the potential of the multicores. To deal with these problems [39] presents HJ-Hadoop, a extension of Hadoop [2] that implements the `Map` and `Reduce` classes using Habanero Java [39], a

Java implementation for multicore parallelism. This framework aims to the reduction of the number of JVMs created per node, in order to better parallelize the *map* tasks and buffering key/value pairs that have to be processed by the *map* function. It has to be said that the focus of this solution is in a intra-JVM context. Hence,the major difference between Hadoop and HJ-Hadoop in the number of tasks, is because the latter subdivides the tasks into asynchronous tasks that are run in parallel on a intra-JVM context. The presented solution starts by allocation of a new buffer per asynchronous task to better take advantage of those resources. When the allocation is over, it is started a buffering of a variable number of key/value pairs to be processed by the *map* function. The actual number of pairs to be buffered is dynamically set by a main thread that tests some samples to identify the optimal number of pairs. The buffering is done by an asynchronous task that fetches from a task queue the pairs to the buffer while other tasks do the processing of these pairs. When the task has completed its work, it frees the buffer.

There is yet another problem in memory use that it is related with the size of the intermediate data structures. This data structures can become too large for the available main memory or just require too much memory, which implies more costly communications with the hard disk. MapReduce has some situations where this can happen, namely in *map* phase at the production and storing of intermediate pairs in the intermediate data structures, at the start of a *reduce* phase when fetching the pairs or even after this phase at the merge phase. A solution to relive the pressure from the main memory and to control the accesses to the hard disk is described in [19]. In this solution the the accesses to disk are controlled by a mechanism that verifies if the main memory has enough space to handle the new tasks. For this mechanism to work a buffer-like data structure has been added to MapReduce, mainly between the major phases. This data structure is presented as spill buffer that when some limits of the main memory occupation are reached it stores the last column of the intermediate data structure (generally represented in MapReduce by an matrix where each row is relative to a *map* worker and the column to a *reduce* worker). The spill buffer stores the key/value pairs in which they are fetched. Thus, this data structure can sometimes be seen as a buffer for disk writing operations, namely because the buffer is filled with pairs until it is full (the size of the buffer is the number of columns of the matrix), in the *map* phase. When the buffer is full, the thread responsible for the buffer starts writing the pairs on the hard disk. In turn, this buffer may also serve the purpose of a loading buffer, for example, to control the space that the to-be-processed pairs occupy in the main memory, in the *reduce* phase. In such case, the buffer is loaded with the necessary pairs.

2.1.4.4 Others

To close this section we present a new implementation of Hadoop MapReduce [2]. This implementation does not solve a well define problem, but promises better performances

than Hadoop in In-Memory environments. The engine is called M3R [33] and implements the Hadoop MapReduce APIs aiming to provide better in-memory support. For this, M3R launches multiple JVMs, and each of these handles multiple *map* and *reduce* workers, which share the heap-state between jobs. This not only improves the memory management of the cluster but it has the potential to parallelize multiple jobs. However it has the limitation of the size of memory that a cluster can provide, which limits the scalability of the system. Another limitation of the system is the absence of a fault-tolerance system, which means that the failure of a node causes the engine to fail. The authors discuss this problem and conclude that M3R is not suited to jobs that take long hours to run, but are more indicated to commodity clusters.

2.1.5 Discussion

The most straightforward approach to take advantage of the parallel processing hardware In-Memory MapReduce is to do a decomposition of the problem's input domain, distributing the result by the processing hardware. However, this approach does not take advantage of the growing hardware dedicated to caching that, if taken into account, can improve the performance of the applications, by making use of the principle of locality.

In the current state of the art some of the presented MapReduce frameworks show some concern about locality. For example, Phoenix++ [34] tries to take advantage of locality by invoking the combiner after every emitted value of a *map* task.

There are, however, some frameworks that not only make some effort to benefit from locality, but also use strategies that could be used in order to better take advantage of the cache hardware. Tiled-MapReduce [11] employs a pipeline of map and reduce tasks that make use of the same memory spaces and reduces idle time of the processing units, promoting locality. Another feature of Tiled-MapReduce is its usage of tiling strategies for the domain split, but this tiling is not cache friendly. Phoenix [31] adjusts the size of the input and output data of a *map* tasks, so that the data can fit in L1 cache, which reveals a concern about the utilization of the cache. However, Phoenix does not take into account the cache organization.

Some solutions presented by the previous frameworks can be further applied to our problem. For example, the tiling strategies can be integrated and explored in the context of our problem. On the contrary, the solution to improve locality used by Phoenix++ is limited to the combiner, which does not enter in our work plan. The same can be said about the strategy of dimensioning the output of *map* tasks of Phoenix. However, the part of that strategy that is related to the *map* tasks input dimensioning presents a similar approach that we idealize.

2.2 Cache-Optimizations

Being the thesis focussed in cache-directed optimizations of MapReduce, it is of a major importance to explore which strategies have been used to perform such kind of optimizations in other types of computations. This overview is important not only to contextualize our work but also to understand what strategies can serve as inspiration. This section starts by giving a general overview on compiler optimizations that can be used to improve the usage of the memory-hierarchy specially in terms of cache usage. Next, in Section 2.2.2, we present some cache-oblivious algorithms that take advantage of caches without knowing the characteristics of these. Before moving into this section, it should be mentioned the principal types of cache, which are the direct mapped cache, n-way set associative cache and fully associative cache. In a direct mapped cache each main memory block is mapped to only one cache line and it does not have a replacement policy. The n-way set associative cache differs from a direct mapped cache in that a main memory block can be mapped to n cache lines and replacement policy depends of the n possible cache locations. The last type is the fully associative cache, which allows each main memory block to be mapped to any cache line.

2.2.1 Compiler Optimizations for Sequential Code

Compiler Optimizations can influence the performance and the usage of the memory-hierarchy by improving the locality, either data locality or instruction locality. Leaving these optimizations to the compiler, frees the programmer of the burden of writing programs with special characteristics to achieve good locality, which can be time consuming and error-prone. These optimizations can be done by applying code transformations based on the analysis of data dependences. It should be noted that there are some optimizations for parallelization of loops with cache awareness. However, due to the sequential execution flow of MapReduce, we chose to just talk about sequential optimizations.

2.2.1.1 Loop Transformations

Loop transformation presented in [29], [10] and [24] is the most common type of cache-directed program transformation. These transformations may be applied to any loop statement, provided that no dependencies are violated, and may significantly improve cache locality and performance of a program, mainly because of the large usage of these statements and specially due to its link to data structure traversal. The most well known loop transformations that leverage locality are loop permutation, loop fusion, loop distribution, loop reversal, loop blocking/tiling and loop skewing. We briefly describe each one of them to get a sense of how these transformations work and how they can be applied.

Loop permutation can improve cache locality by modifying how a program traverses the data in cache. For example, assuming a scenario where a program traverses a matrix first by column and then by row, but the matrix is stored in row-major order. This way

of traversing the matrix will cause many misses in the cache, which prevent the reuse of the data and consequentially prevent cache locality. If no dependences between loops are violated, loop permutation can be applied in this situation by swapping the order of the traversal loops, where it starts by traversing the rows and then goes through the columns. This transformation is used in situations such as the latter, where the traversal of data structures are altered to adapt to the memory layout, reducing the number of cache misses. However this transformation has other purposes [29], for example, inner loops paralellization and "persisting" some data into registers.

Loop blocking/tiling increases the depth of a loop nest, which means adding more loops to the nest. As described in [24], this strategy enables more data reuse in cache to improve data locality. Matrix multiplication is a problem that can be improved in terms of data locality by applying loop blocking.

Another strategy is loop fusion. A strategy that combines two adjacent loops, which use the same iteration space traversal. This transformation reduces the overhead of multiple loops with common iteration space. Also, in terms of cache optimization, it improves locality when these loops share accesses to the data structure, such as an array. In the case of multiple loops accessing the same array(s) in the same order, the number of cache misses are reduced by taking advantage of the fact that a single loading of the data may serve several loops. This may not happen when the loops are separated, resulting in multiple loads of the same data from main memory in the second loop.

The reverse of the fusion transformation is loop distribution, which disperses the statements of a single loop into multiple loops. This strategy by itself can help to synchronize accesses with the memory data layout. However, loop distribution can also be a mean to enable loop permutation.

Yet another strategy that enables optimizations is loop reversal that reverses the traversal order in which a loop iterates, if the dependences are not violated. For example, this transformation can allow the utilization of loop permutation. Loop skewing rearranges the accesses to a multidimensional array of a nested loop where the inner loop depends on the its predecessors in order to move the dependencies to the outermost loops.

Loop transformations can be combined, either to enable utilization of others, or to better adapt the data problem to the way the cache has to be populated, in order to achieve better locality. Compound [10] is an algorithm to reduce the number of cache lines accesses that combines these loop transformations. This algorithm uses loop permutation, fusion, distribution and reversal together with a cost model to reduce the number of cache lines accessed, or improving the utilization of the data in cache.

2.2.1.2 Data Placement

Some optimizations to the memory hierarchy can be done via data placement. Data placement is a compiler optimization that assigns addresses to data objects, such as global variables, stack variables, heap and constants defining the location in memory of these.

It is a NP-complete problem that consists on the allocation of data onto certain locations, such as the cache memories, in order improve locality and to eliminate interference cache misses [28]. In this section we present some solutions that use data placement strategies to improve the locality in cache, which then can translate into better performance for the application. These do not thrive for optimal solutions but have demonstrated that good results are possible if the cache features are take into account.

An example of a data placement optimization is Cache-Conscious Data Placement (CCDP) [9], which has the objective of reducing the frequency of the data cache misses. CCDP execution comprises three phases in order to optimize data placement, the profiler phase, the data placement optimizer phase and the run-time support for allocation of heap objects phase. The first part of the CCDP, the profiler, consists in the characterization of how a given program uses its data. The characterization of a program, treats each heap, stack, global or constant variable as an object, and produces two profiles, the Name and the Temporal Relationship Graph, that are composed by data structures of these objects to determine an estimation of the number of cache conflicts that would arise if two of these objects were overlapped in the same cache line. When the characterization is over, the produced profiles are fed to a data placement optimizer in the compiler, which reorders the global data segment and the stack to their new starting locations. If heap optimization is performed at this phase, then are created some custom allocation routines for posterior placement of heap objects. In the last phase, the routines created in the previous phase are used to allocate the data at the preferred locations indicated by the data placement algorithm. The data placement algorithm used in CCDP uses the profiles data structures together with the dimensions of the target cache to achieve its objective of reducing cache conflicts and increase the cache utilization.

Another approach of the data placement concept is Cache-Conscious Structure Layout (CCSL) [12]. CCSL provides a mechanism to improve the cache locality and the performance of pointer-manipulating programs by exploring data placement techniques such as clustering and coloring. These techniques are focused in increasing the temporal and spatial locality of pointer structures, hence, improving cache utilization and reducing cache-conflicts. The clustering technique tries to group data structure elements that are likely to be used at the same time into the same cache block, in order to improve the locality, either temporal or spatial, and at the same time provides a form of prefetching. Coloring handles the problem of caches with finite associativity, which is the limit of the number of concurrently accessed data elements that can be mapped onto the same cache block. To cope with this, its done the coloring of the partition's cache space into regions of frequently accessed elements and unfrequently accessed elements, so that the data accesses of the last do not conflict with others, reducing the number of cache misses. CCSL uses these two techniques to implement two strategies of data placement, the cache-conscious data reorganization and cache-conscious heap allocation. Cache-conscious data reorganization, as the name indicates reorganizes the layout of the data structures in memory, mainly because the layout produced by the first data structure allocation may not

be the most appropriate for the program's data access patterns, reducing the probability of cache misses. The cache-conscious heap allocation complements the previous and, as also the name indicates, performs cache-conscious data placement at the allocation of the elements. CCSL [12] reports performance benefits.

The work in [13] addresses the issue of choosing the size of a tile when aiming to improve cache miss rates. This algorithm named Tile Size Selection chooses the size for tile, in a give problem, by taking into account the data placement, cache size and cache line size in a directed-mapped cache. As stated in Section 2.2.1.1 the utilization of tiling enables reused data to be moved closer together, thus eliminating capacity misses that occur when the workload it is too big for the cache. The present solution also covers misses resulting from replacement of data elements by another data elements of the same data set, being called self-interference misses. To choose the tile size the proposed solution first computes a set of tile sizes that do not cause self interference or other type of misses, then it chooses the tile by the cross interferences and the size of the workload. A restriction that it is imposed for the selection of a tile size based on the cache size, line size and array column dimension, is that the column dimension must be multiple of the cache line size. This solution shows that the usage of a compiler optimization strategy such as tile size selection algorithm can provide to a program good performances on directed-mapped caches.

2.2.2 Cache-Oblivious Algorithms

Another approach to improving the mapping of a program onto a memory hierarchy is cache-oblivious algorithms. The oblivious qualifier relates to the fact that these algorithms use asymptotically optimal workloads and move data between multiple cache levels in a asymptotically optimal manner without the need to be tuned to the features of a given hardware configuration, such as cache size and cache-line length.

The objective of these algorithms is to minimize the utilization of the hardware configurations while improving or at least getting low cache miss rates. These algorithms were first mentioned in [21], where to analyse them and to assess its complexities was introduced an ideal-cache model. This model is composed by a two-level memory hierarchy comprising a full associative cache and a large main memory. In [21] are described algorithms for matrix multiplication and transposition, and even sorting algorithms that are asymptotically optimal with cache-oblivious approaches, using recursive divide and conquer strategies. However, probably the most important proof is the fact that if an cache-oblivious algorithm yields good performances in a full associative cache setting, it is likely that it will achieve such kind of of performances in other cache models, such as two-level models, multilevel ideal caches, hierarchy memory model and serial uniform memory hierarchy (SUMH).

The design of these algorithms led Erik D. Demaine [17] to the exploration of data structures that are based in the cache-oblivious concept. The work shows the possibilities

of using cache-oblivious design to create static data structures, such as static search trees, and dynamic data structures, such as b-trees and linked lists, with good performances in data locality.

2.2.3 Memory Hierarchy Aware Programming Models

Along with the cache-conscious approaches, there is some work in the field of memory hierarchy aware models. To better handle and represent communication between memory hierarchy elements this type of models need to represent the memory hierarchy in an abstract manner.

At the base of memory hierarchy-aware programming models is the Parallel Memory Hierarchy (PMH) model [1]. This model abstracts a parallel architecture into a tree-like memory hierarchy, modelling the communication in this hierarchy and its costs.

Sequoia [18] was the first system to instantiate and extend the PMH model, followed by other, such as Hierarchical Tiled Arrays [20] and Hierarchical Place Trees [38]. In this section will briefly describe Sequoia and Hierarchically Tiled Arrays for being those more closely to what we intend to do in this thesis.

2.2.3.1 Sequoia

Sequoia [18] is a programming language designed to ease the development of memory hierarchy aware programs and to address the problem of communication between tiers of the memory hierarchy. Moreover, it aims to produce programs that can be portable and hence independent of the hierarchy of one machine.

Sequoia abstracts the concept of hierarchical memory by mapping it to a tree-like model, so it can differentiate the various memory models of a hierarchy. However, it is the job of the programmer to define such model with the language's mechanisms. The result does not only describes the data communications between tiers but also contains the location where the data is stored. Consequently, it yields gains in portability and even performance. Portability gains can be seen mainly in the separation between problem solving logic, or algorithmic expressions, and machine-specific optimizations.

More efficient executions can be achieved by partitioning computations into smaller ones, mainly because of the increased probability of using concepts such as locality and also by increasing the utilizations of the available hardware. Some techniques for computation optimization are explored by Sequoia, such as the tiling for cache locality or problem decomposition in a more broader context for minimizing network communications in clusters. The computations themselves are represented via the task abstraction, which describes the communications and workloads of the computation. In order to enable parallelism these tasks are isolated in their local address space without any form of communication with other tasks besides the calling of sub-tasks and returning to parent, similarly to MapReduce tasks. Also, as in MapReduce it should be noted that the only form of data communication in Sequoia is expressed by passing arguments to a tasks.

Sequoia has the possibility to define many implementations of a task in order to specify the better implementation for a given context, for example, a implementation of a task can be useful in a L2 cache, but not suitable for a L1 cache in a computation. To achieve this separation of implementations, Sequoia defines a task in one of two variants, inner and leaf. The inner variant of a task implementation has the job of calling sub-tasks that can be `inner` or `leaf` tasks and handles the communication of the data blocks to these sub-tasks. It is also worth mention that `inner` tasks cannot access the data blocks directly, this is a measure to enable code portability preventing tiers of the hierarchy that do not have access to processing units to perform computation on the data nodes. The `leaf` variant is responsible for doing the computations on the data blocks that are local to that tier.

With regard to the control of the program structure and execution in a memory hierarchy-aware approach, the programmer must specify an independent file that describes which tasks have to be executed in determinate memory level and which implementations should be used. Additionally, the programmer may have to specify the parameters values, called `tunable`, to adapt the task execution to its execution context, in order to control the computations. An example of a tunable parameter is the number of elements of an array that must reside in the target cache level. Another way to control the execution of a program is through the constructs offered by the language for task implementation, such as `mapper`, that maps tasks in a parallel iteration, creating sub-tasks that are executed in parallel, and `mapreduce`, which maps tasks onto blocks and performs a reduction operation to these tasks to reduce the number of subtasks produced.

Sequoia places some burden in the datasets' communication up and down the hierarchy levels. This burden is caused by the divide and conquer strategy employed that splits the data set into smaller chunks until these can fit the lower, smaller and faster levels of the hierarchy. Besides this possible communication burden, Sequoia uses a static approach to hierarchically partition the data that as stated in [35] can be insufficient for some cases, where the best approach is to do the partitioning dynamically. Recently, to handle these situations, an approach based in the Legion programming model [35] proposes a solution that incorporates static and dynamic partitioning and offers a more relaxed view in the partitions, where they are called regions and can overlap each other and are even possibles alias.

2.2.3.2 Hierarchically Tiled Arrays

Tiled Arrays are a class of data structures that boosted the creation of a memory hierarchy aware programming model called Hierarchically Tilled Arrays [20]. This model permits the development of single-threaded programs that use the operations of tiled arrays to perform the computation and communication between processing elements in parallel.

Tiled arrays are arrays partitioned into tiles of the same dimensions. In turn a hierarchically tiled array (HTA) is a tiled array where each tile can either be a normal array

or another HTA. These HTAs can be created in two forms, one form simply partitions an array using well defined delimiters for its dimensions, for example, indicating in which columns and rows the tiles start and end. The second form partitions the array by defining the exact dimensions of the tiles. With the partitioning of the arrays into tiles, HTAs can help in the improvement of locality and even facilitate the usage of parallel approaches. This improvements can be achieved by distributing the outermost tiles by the processors for parallelism and the innermost for locality.

In this programming model a program runs on a main thread that is connected to a distributed memory served by multiple processors, called servers, which are associated with the top-level tiles. It should be noted that the subsequent tile sizes must be explicitly defined by the programmer, in other words, the program has to compute the tile size along the hierarchy. There are some distinct execution situations that can occur in this model. One, where a computation involves some distributed tiles of a HTA is handled by broadcasting the computation to the related servers in order to do this computation in parallel. Another situation may arise when a computation only make reference to tiles that are stored in a single server, then the server does the computation locally. Yet another situation can be described when a server needs tiles stored in other servers, in this situation the server must first request the tiles and then execute the computation. A pattern that can be extracted from this situations is that the computations related with tiles of a HTA control the communications and the parallelism of a program.

2.2.4 Cache-Conscious Decomposition of Data-parallel Computations

The work proposed in [26] aims at automatizing the cache-conscious decomposition of data-parallel, where it removes from the programmer the burden of programming cache-oblivious algorithms or explicitly programming the memory hierarchy. This solution determines the optimal partition size of a domain, and controls the whole execution of an application in an automated manner relieving the programmer of the hardware related burden With this, the programmer can focus just on the decomposition itself and on the application logic.

The main difference between the approach presented in [26] and the programming models presented in the previous section, without considering the execution automation, is that the first only requires some information about the data layout to calculate the optimal partition size for a better cache fit. To get this information this model requires that the programmer instantiates the interface `Distribution` shown in Listing 2.1.

```
1 publicinterface Distribution<T> {
2 /**
3 * Partitions the input domain into nParts partitions .
4 * @param nParts the number of partitions to be produced
5 * @return the partitions
6 */
7 T[] partition (int nParts);
```

```
8  /**
9  * Returns the average size of a partition of T (in number of elements )
10 * @return size of P
11 */
12 float getAveragePartitionSize(int nParts);
13 /**
14 * Returns the average size of line of a partition of T (in number of elements )
15 * @return size
16 */
17 float getAverageLineSize(int nParts);
18 /**
19 * Returns the size of an element of T (in bytes )
20 * @return size
21 */
22 int getElementSize();
23 }
```

Listing 2.1: The `Distribution` interface from [26].

The instantiation of the `Distribution` interface, helps the model to estimate the optimal partition size, but this estimation is done with an iterative algorithm based on the target cache level (TCL). The main idea of this work is to iteratively try new partitions sizes, so that the optimal size is achieved. To accomplish this, the methods in the `Distirbution` interface help to validate this partitions sizes. In order to validate a partition size this one has to comply with some conditions regarding the TCL dimensions. The main condition says that the sum of quotients between the size of the domain(s) and the number of partitions has to be equal or less than the TCL size. At each iteration of the algorithm uses the method `getAverageLineSize`, which validates whether a dataset may be decomposed into the number of partitions passed as argument and, if its valid, the method returns the average line size of the partitions. With this validation the method is used to stop the partition size search when the value of the number of partitions is not valid.

In [26] are discussed some static scheduling strategies that are based on the idea of an initial distribution of the tasks by the workers in a way that these workers receive a sufficiently large group of tasks to balance the difference between the number of tasks and the number of processing cores. The strategies used are Contiguous Clustering and Sibling Round-Robin Clustering (SRR). The first explores locality by assigning contiguous tasks that have contiguous data to the same task. The SRR is based on the concept of round-robin, as can seen by its name, and how the processing cores use the memory hierarchy or how these cores share the Last Level Cache (LLC). In this scheduling strategy there are two levels of task assignment, the cluster-assignment that assigns clusters of tasks to groups of workers which share the LLC, and the tasks-assignment that distributes the tasks within the groups by the workers in a round-robin way, as depicted in Figure 2.6 b).

These strategies are used taking into account the trade-off between the execution

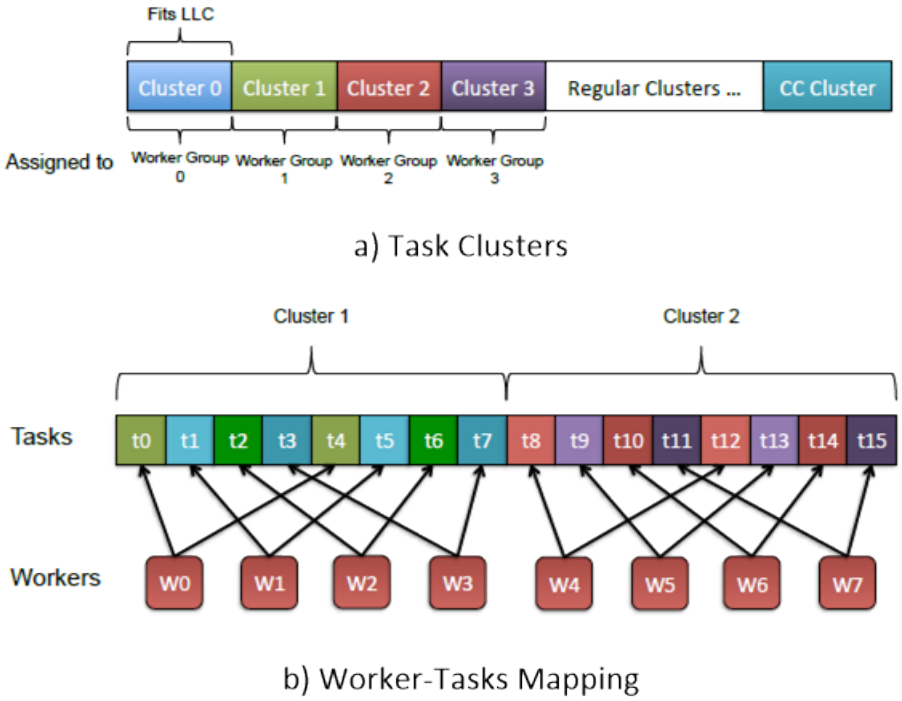

Figure 2.6: Sibling Round-Robin Clustering from [26].

overhead of these approaches and the usage of the memory hierarchy, more specifically the cache hierarchy.

2.2.5 Discussion

As can be perceived from the previous sections, there is a lot of work that can lead to the statement that the proper use of cache memory can greatly improve the computations of a given application, mainly through data locality. However, none of the studied and documented approaches in this section can be applied to the MapReduce model. In fact, as far as we know, there is no solution in this context that addresses these four big topics:

- Compiler Optimizations;
- Cache-Oblivious algorithms;
- Cache-aware programming models;
- Automated decomposition;

Not all these topics are applicable to the solution presented in this document, such as the compiler optimizations and cache-oblivious algorithms. The compiler optimizations are limited to solutions that are performed before the execution of a program, which does not fit in the context of our work. The cache-oblivious algorithms try to improve computations that take advantage of the cache without knowing anything about its features, unlike our objective that is based on these to try to achieve better performances.

Although the studied solutions cannot be directly applied to our context, there are several concepts that can serve as inspiration. An example of useful concepts that can be found is the block-tiling strategies, as seen in the Tile Size Selection Using Cache Organization and Data Layout [13]. This solution takes into account the features of the memory hierarchy, specially of the cache memories and also use some tiling strategies, both can be applied to our decomposition solution so that it takes into account these features.

Our work is the logical follow-up of the one previously conducted in [26]. This work was developed under the concepts of Sequoia and Hierarchically Tiled Arrays, but it also keeps the memory hierarchy management transparent to the programmer. However, this work aims only embarrassingly parallel computations and was implemented in the context of an experimental platform [27] [32], which is not in the context of a framework with the generality and complexity of Hadoop [2].

We intend to build upon the strategies developed in [26] for the *split* and *map* phase optimizations. Therefore, the strategies that include partition's size and the cache memory hierarchy informations will be refined to the reality of MapReduce computations, which do not exist in the current state of the art. For instance, the gathering of the information of the memory hierarchy and the utilization of this information to the decomposition in a MapReduce application.

Chapter 3

Cache-Friendly Tiling for MapReduce Tasks

The main objective of this work is to improve the performance of MapReduce computations that benefit from the concept of temporal locality. This concept predicts that if we maintain in memory a data set that is accessed multiple times in a short span of time, the performance of the application that uses that set can improve just by avoiding moving data between different memory levels. In order to improve the performance of applications that benefit from this concept, we focus on improving the mapping of MapReduce computations in a machine's cache memory hierarchy. This mapping has to be portable, meaning that the performance of MapReduce applications that benefit from temporal locality should be portable across distinct architecture, mainly distinct cache hierarchies, with little or none intervention from the programmer. Therefore, the final form of our objective can be described as an in-node approach of a cache-conscious Hadoop MapReduce extension.

To this extent, in the present chapter we first present the concepts of our approach and how they fit in the MapReduce execution model. Followed by an explanation of how these strategies were applied, presenting the actual changes to the Hadoop MapReduce phases and all the components added. Finally, we analyse the impact of our solution in the programming model, showing what the programmer has to do differently to use our solution.

3.1 Approach

To develop our temporal locality approach of MapReduce tasks we had to analyse which phases of this model were better candidates for the enhancement of the mapping of an application's data into the memory hierarchy. Given our objective, we wanted to modify the way the input data is partitioned and distributed, so that we could harness the full

potential of the underlying hardware. A modification like that has to be done so that the application data can be partitioned in a way that enables the exploitation of the concept of temporal locality. Besides leveraging the cache memory hardware, a computation can obtain better performances by taking advantage of its machine's processing resources. Given that, if this partitioning is performed also by taking into account the number of cores available in a machine, then some conditions can be created for the parallelization of computations. Howeverm, this parallelization is only possible if the application does not have a lot of data coupling, so that the tasks resulting from the created partitions could be processed in parallel and then later aggregated in a final result. Thus leading to some more application parallelization and consequentially to some computation time improvement.

From the presentation of the MapReduce model in Section 2.1.1, the natural candidate phases to handle the decomposition and mapping of an application's data are the *split* and *map* phases. Both phases have some data distribution control. Although the *map* phase could be modified to integrate the desired decomposition before the *map* function is applied, the conceptual idea of this phase was not to include data decomposition of operations. Therefore the *split* phase presents itself as the best option to implement some modifications to the application's data decomposition, which can leverage the usage of temporal locality and better avail the hardware.

In order to improve the temporal locality of the computations performed by the *mappers*, the *split* phase must produce key/value pairs that are best suited to that end. The normal key/value pair contains as its value field units such as a line or even a single value. However, these units are not always the best for computations that benefit from temporal locality. As seen in Chapter 2, the usage of block-tiling strategies can benefit the computations that benefit from the concept of temporal locality. With this in mind, Figure 3.1 shows the two different decomposition approaches: one that uses a line as its value field, the line approach; and a second that uses a block as its value field, the block approach.

The *line* approach is one of the most used approaches, where the pairs produced by the *split* phase are composed by a given key and a line of input data. This approach is mainly used for text input, wherein the value can be a sentence from a document or a line of numbers to be analysed. As an example, consider the stencil computation presented in Section 1.1.2. With the *line* approach a *mapper* that receives a line of data is limited to compute a partial value, mainly because the line does not contain enough information to complete the computation. With only partial values in the *map* phase, the application will have to do extra data transfers which implies more performance costs. By transferring these partial results to the *reduce* phase to perform the full computations, it is likely that the application's performance will suffer some degradation. The main reason for that is related with the fact that the *reduce* phase is performed completely sequentially. This situation can happen if the logic of the application is not taken into account or, in other words, the value of a key/value pair is not the best fit for a given computation.

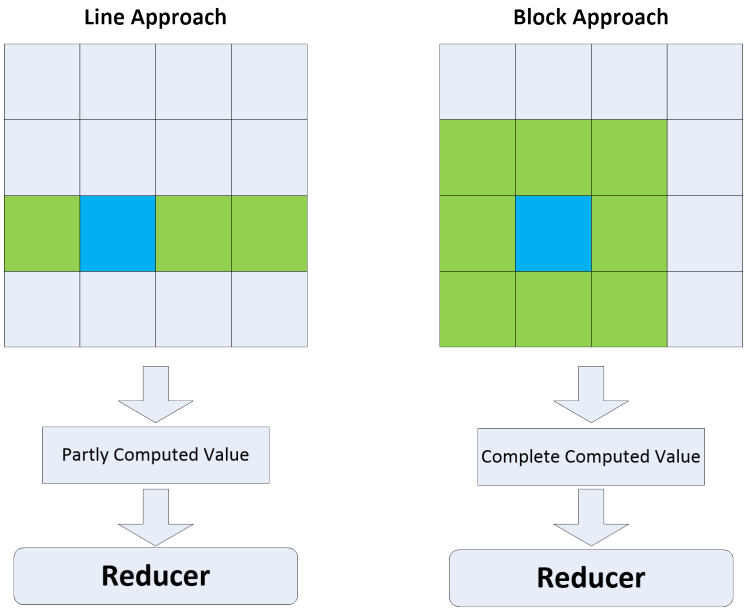

Figure 3.1: Difference of using a *line* approach and a *block* approach, on the beginning of the *map* phase of a stencil application.

Applications that benefit from temporal locality, such as stencil computations, can gain from block-tiling strategies. Given that, the alternative block approach is based on the block-tiling strategies and only differs from current approaches in the content of the value that composes the key/value pairs. To the best of our knowledge, this approach has not yet been studied in the context of the generic MapReduce model. However, as can be inferred from Figure 3.1, if a data block is received as a value by a *mapper*, it is more probable that this block includes the needed neighbourhood for the element's computation. If this neighbourhood data can be maintained in memory, then there can be some improvement in the avail of temporal locality at the *map* phase. Therefore, by getting more neighbourhoods completed in the *map* phase, there can be more final results in this phase. Getting these results at the *map* phase not only can reduce the costs of data transferring, but also reduce the amount of computation done in the *reduce* phase. The reduction of the computation in the latter phase will contribute to improve the application's overall performance. Adding to this the usage of cache-conscious sizes and we can further reduce the number of data transfers between cache memory levels.

The previous example showed the benefits of using block-tiling strategies to produce key/value pairs that leverage temporal locality, but in order to map the application's data onto the cache memory hierarchy these pairs have to be cache-friendly. This means that we have to introduce some cache memory hierarchy information in the production of key/value pairs. By producing data blocks using information such as the cache size, cache line size and others, we can improve the data transferences between cache levels and adapt the data to the cache geometry.

In sum, some modifications have to be applied to the *split* and *map* phase to cope with

the introduction of data blocks with sizes that fit a target cache level. Being our focus in the in-memory optimizations, it is desirable that our modifications do not affect the cluster level, either in its configuration, execution model or even at the programming model. This is due to the objective of maintaining compatibility with the current framework, so that our solution can be easily integrated. Therefore, our focus was always on the local execution, in-node, of MapReduce which means that its execution model stays unchanged on the cluster level, hence the compatibility with the current version it is not called into question. Nonetheless, is important to mention that our approach supports heterogeneous clusters, since the modifications are made at an in-node context, and thus the optimizations are made according to the memory architecture of each target machine.

3.2 Decomposition Implementation

In this section we will detail how our approach is implemented using the Hadoop MapReduce framework. First we start by giving an overview of the implementation, mainly to identify the components that had to be modified. Subsequently, we go through the details of our implementation.

In order to see how an application can be mapped onto the cache memory hierarchy with our solution and also to understand the modifications that were made at the in-node level of Hadoop MapReduce, we must remember the in-node execution model shown in Figure 2.5. There, we can see that each node/machine receives a partition of the original input dataset, being this partition locally handled by an instance of the `InputFormat` class. The latter decomposes the received partition into several splits, which are subsequently parsed into key/value pairs (by `RecordReaders`) to be consumed by the *mappers*. With this model it can be perceived that to address the application mapping and, hence, its data decomposition into data blocks that fit a given cache level, we have to modify the way the data is read from the split and how the key/value pairs are build. In this context, our modifications to the framework target the two main classes of the *split* phase: the `InputFormat` and `RecordReader` classes.

3.2.1 Implementation Overview

The integration of our solution in the Hadoop MapReduce framework was done by developing some classes as an extension of what exists, instead of modifying the classes currently available in the framework. These classes are related mainly with the *split* phase, where our solution decomposes the splits produced by the Hadoop MapReduce classes into multiple data blocks that are supposed to be fed to the *mapper* of each split. As mentioned in the previous sections of this chapter, the objective of this decomposition is to provide to the *mappers* data blocks that have enough data to produce computations that take advantage of the parallel executions of this phase. At the same time, the size of data used to produce these blocks is dependent of some cache memory hierarchy features.

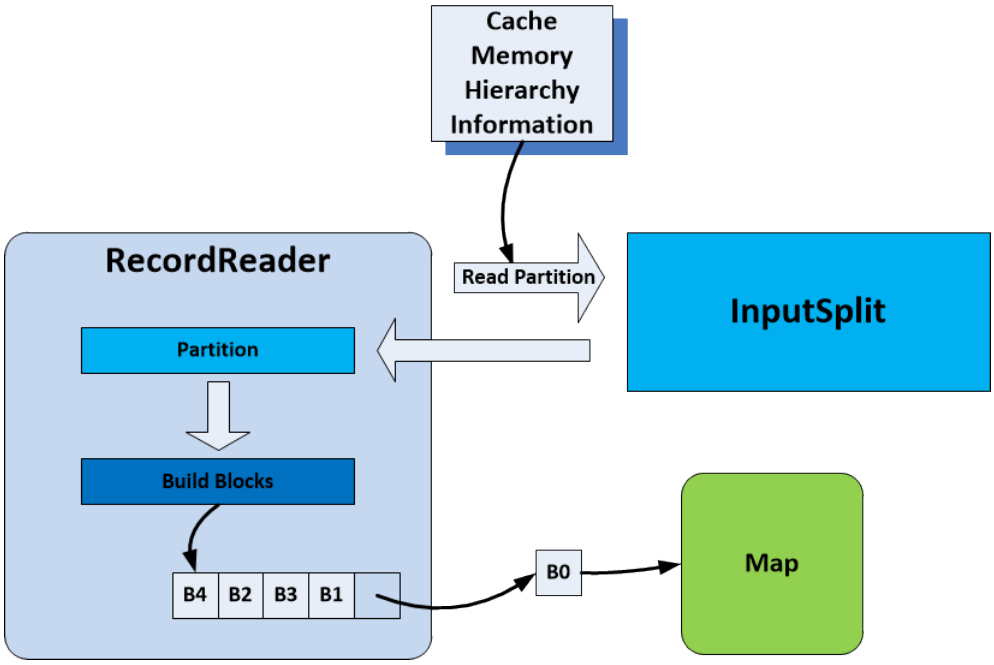

Figure 3.2: *RecordReader* block decomposition with a cache-concious support.

To apply this solution we had to target the `RecordReader` class, which is the central class of the *split* phase of Hadoop MapReduce. The main idea for our class, which performs a cache-friendly block-tiling decomposition, can be divided in two parts: the reading of a partition with a size influenced by some cache memory hierarchy informations, and the integration of an application's decomposition logic.

As can be seen in Figure 3.2, the first part of our decomposition is focused in reading data from a *mapper's* split. This is represented by the *ReadPartion* action in Figure 3.2. The main idea of this action is to read a partition of a split that has a size that is dependent of some cache memory hierarchy features. Given that, our `RecordReader` class reads from a configuration file information of the cache hierarchy, such as a given cache size, cache line sizes, or even the level of cache sharing (how many processor share a cache level). A given algorithm uses this information and produces a value that is the size of the partition read from the split. However, one thing that we have tried to achieve was the possibility of modifying this algorithm, so that the programmer could choose the hierarchy features that were more relevant for its application. With this, the algorithm can be implemented by the programmer. For example, we can use the size of a target cache level to limit the partition size. This approach assures that the blocks produced from a partition of that size will fit in the target cache, which can help in the mapping of the applications data onto the cache hierarchy.

The second part is related with the integration of block-tiling strategies in the extended `RecordReader` by the definition of an application's decomposition logic. This logic will receive the partition's data and use it to produce the data blocks that are fed to

the *mappers*. However, this decomposition logic is subjective to each different application, wherein each application may require different kinds of blocks. With that in mind, the programmer implements a decomposition logic to adapt our `RecordReader` to a given application. The result of the implementation of these methods is represented in Figure 3.2 by the *Build Blocks* action, where it can be seen that these actions take the partition's data and produce blocks that are then passed to the *map* phase.

In the next section we will present in more detail the implementation of the decomposition approach and its cache-friendly support in the abstract `RecordReader` and explore the possibilities that can be drawn from the proposed methods.

3.2.2 Implementation Details

As previously discussed in Section 3.2.1, different applications may require different decomposition logics, hence the developed classes were idealized with two intentions in mind. The first is related with the main objective of this work, optimize the usage of the locality principles. The second intention was tied to the importance of developing these classes to be as extensible and easy to modify as possible, without losing the main decomposition principle. Given that, we developed classes for the *split* phase that could be used by every kind of application where the use of data blocks makes sense. Next we describe the classes that we have extended to integrate our solution into Hadoop MapReduce by first explaining the details of the block decomposition and then discussing the support for cache-concious approaches.

3.2.2.1 Block-Tiling

The solution is composed by the implementation of the two main classes of the *split* phase, the `InputFormat` and the `RecordReader`. In this section we go through the details of each class implementation.

We start by the `InputFormat` class, where we have extended the class `SequenceFileInputFormat`. The choice of using this class as base for the extended `InputFormat` is related to its requirement of a type of binary input file, which is composed by key and value objects. This type of files are called `SequenceFiles` and are offered by the Hadoop MapReduce, as a file that is easier to manipulate by `RecordReader`. With this, our `InputFormat` class is called `BlockSequenceFileInputFormat` and it only differs from a regular `SequenceFileInputFormat` in the implementation of the `createRecordReader` method, where its implementation returns an instance of `BlockSequenceFileRecordReader` (our `RecordReader` implementation).

The `BlockSequenceFileRecordReader` class targets the production of key/value pairs whose value field is a data blocks. As the other `RecordReader` classes, the `BlockSequenceFileRecordReader` gets a data partition from its `InputSplit`, which is passed to a programmer's decomposition algorithm that produces the key/value pairs. Depending on the programmer's decomposition, the data partition can produce more than one pair, but

Method	Description
HierarchyLevel getHierarchyLevel()	Returns an HierarchyLevel, that represents a level of the cache hierarchy
long getPartitionSize()	Returns the size of split's partitions, taking into account the cache level given by the user
Tuple<K,V> buildBlock(List<Tuple<K,V>> pairsList)	Abstract method that builds a pair composed by a *K*ey and a **V**alue, where the value is a block of input data
Object copyKey(Object o)	Auxiliary abstract method to prevent errors in the modification of key objects
Object copyValue(Object o)	Auxiliary abstract method to prevent errors in the modification of value objects
boolean hasMoreBlocks()	Boolean abstract method that indicates if there is any more blocks in the chosen data structure to be consumed
Tuple<K,V> getNextBlock()	Abstract method to get the next block from the chosen data structure
int size()	Abstract method to get size of the chosen data structure

Table 3.1: `BlockSequenceFileRecordReader` class methods (not included methods inherited from the base `RecordReader`).

only one pair is handed to the *mapper* at a time. Accordingly, the remainder pairs have to be stored somewhere, so we introduced a set of methods that interact with a storing entity that keeps these surplus pairs, which can be a simple data structure or a even a data base. Following the same idea of the `RecordReader` extensibility, the choice of how and where the surplus pairs must be stored is delegated to the programmer. The motivation for this design decision arises from the fact that different decomposition algorithms (or even applications) have different orders in which the pairs are outputted to the *mapper*. Using a default data structure independently of the decomposition logic, could hamper the task of expressing this logic. Given that, we created a set of abstract methods that the programmer must implement when extending the `BlockSequenceFileRecordReader` class. These methods are presented in Table 3.1, and are divided into two groups: the methods to express the application's decomposition logic of a given application that produces data blocks; and the methods that are used as the communication interface between the `BlockSequenceFileRecordReader` and the programmer's data structure. The first set of methods is composed by the `copyKey`, `copyValue` and the `buildBlock`. The methods related to the data structure are the `hasMoreBlocks`, `getNextBlock` and the `size`.

In order to introduce the decomposition logic methods and to get a more precise view of what happens under the hood of the `BlockSequenceFileRecordReader`, we present Figure 3.3. This figure shows what happens when a *mapper* asks for a new key/value pair from its `RecordReader`. The latter starts by verifying if the data structure that stores

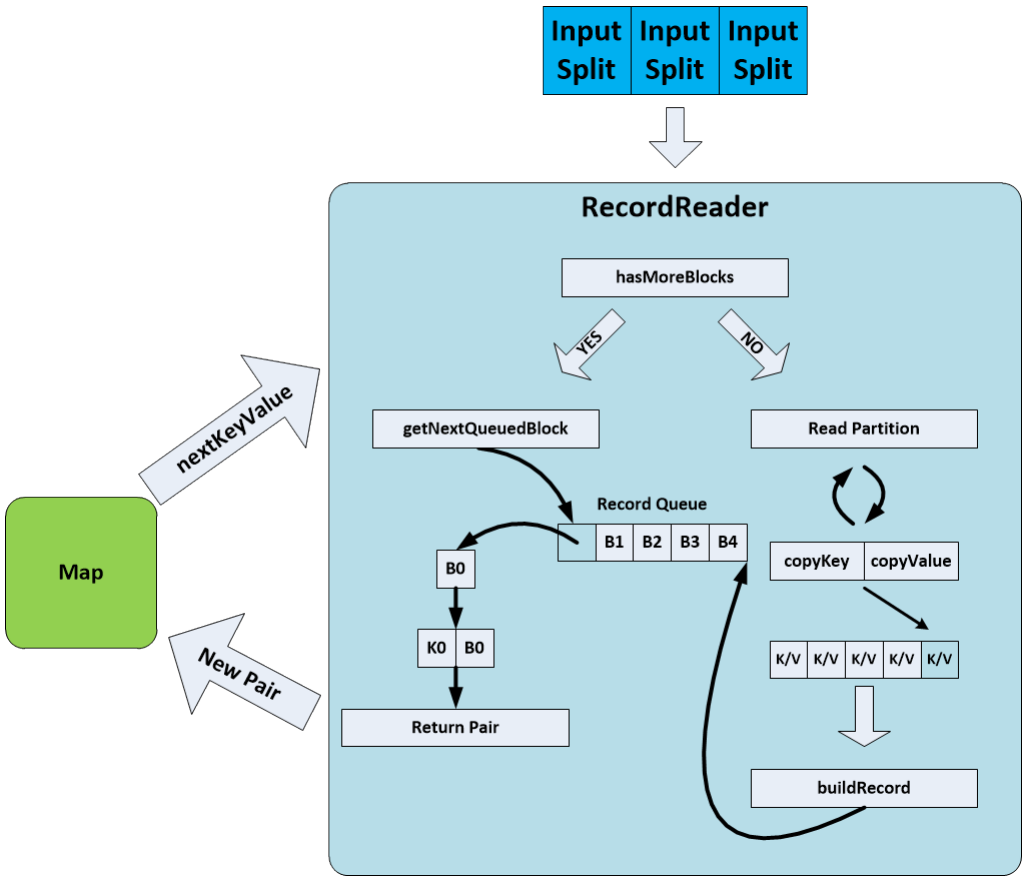

Figure 3.3: Solution `RecordReader` execution model.

the surplus pairs has some pairs available, by invoking the method `hasMoreBlocks`. This method is therefore responsible for verifying if the data structure is empty. Depending on the value returned by the `hasMoreBlocks`, there can be two scenarios: where the data structure is empty (most probable at the first call); and a the second case which implies that are at least one pair ready to be sent to the *mapper*.

The first case can be interpreted as there is no more pairs to hand to the *mapper*, or it was the first request. The first request for a key/value pair, as the majority of the requests, starts with `RecordReader` reading a `InputSplit` partition. This partition is not read at once, instead the `RecordReader` reads it pair by pair until the pre-determined amount of bytes is read. This amount is determined with the help of some information of the cache memory hierarchy, but that it is left for discussion in Section 3.2.2.2. As each key/value pair is read from the split, the `copyKey` and `copyValue` produce a copy of the pairs and stores the copies in a internal list. The act of copying the keys and values is necessary to avoid errors, such as the ones related with adding references of the key and value objects to the internal list. This problem can happen mainly because of the split read mechanism offered by the Hadoop API. Given that, the `RecordReader` is agnostic of how the data is handled, it is of the programmer's responsibility to specify how to copy the key and value objects.

After the partition's size reaches the limit, the list of key/value pairs is passed as a parameter to the invocation of the `buildBlock` method. Is in this method that the programmer must express the application's decomposition logic by implementing the algorithm that produces the data blocks and consequently the key/value pairs. As this method can produce more than one pair for a given partition, it is also the `buildBlock` responsibility to handle the surplus pairs. This behaviour can be seen in Figure 3.3, where `buildBlock` uses the input key/value pairs and produces new data blocks that are stored. In reality, what is stored are key/value pairs and not the blocks, the figure presents them as blocks to emphasize the usage of blocks in the value field. However, even with the storing of the surplus key/values pairs, the programmer's implementation has to return a pair. This pair can be one produced at the time, or one stored at the data structure. Hence, the programmer also has the power to choose in which order the key/value pairs are handed to the *mappers*.

The second case is closely related with the end of the `InputSplit`. When the `RecordReader` finishes reading the respective `InputSplit` there may still exist some pairs to be processed by the *mapper*. Hence, after the `InputSplit` is fully read the control of the order in which the pairs are sent is passed to the `getNextBlock` method. The control changes to this method because it is the responsible for returning the next block from the data structure. Therefore, this method is responsible by returning the stored pairs produced by the `RecordReader`, doing it in a order specified by the programmer's implementation of this method.

For both cases the *mapper* call to its `RecordReader` end with the return of a key/value pair that contains a data block as its value field.

3.2.2.2 Domain Splitting

The decomposition of the application's data onto the cache memory hierarchy its done mainly by the determination of the size of the partition read from the `InputSplit`. This is done in method `getPartitionSize`, also present in Table 3.1. The implementation of this method joins the information retrieved from a configuration file and an algorithm that determines the desired partition size. The configuration file provides a specification of the cache memory hierarchy an its features, such as the cache size or line size. With this information some strategies can be employed to produce partitions that consequently give rise to cache-friendly data blocks. Perhaps the simplest example is to use a target cache level size as the size limit for a partition. This approach ensures that all the blocks produced from these partitions will fit on the target cache, which is the default implementation in the `BlockSequenceFileRecordReader`.

Nonetheless, there are more complex strategies that cannot be expressed just by the size of the partition. For example, a vertical decomposition approach that basically partitions the input data throughout the cache hierarchy, as explained in [26]. This more

complex approach can be expressed in the `buildBlock` method as the programmer's decomposition logic by starting with a partition with the size of the last level cache (LLC), and using the cache memory hierarchy information available by the `getHierarchyLevel` method, to decompose the partition through the cache hierarchy.

3.2.2.3 Other Details

One of the work premisses was to explore the locality and decomposition concepts in an in-memory environment. However, as mentioned in Chapter 2, the Hadoop framework was developed mainly for a cluster environment and uses disk-oriented approaches to store intermediate results. Besides the in-node context, each task is executed on its own JVM (Java Virtual Machine), which means that the tasks cannot share a process-level heap. These two features of the Hadoop framework represent a obstacle to an in-memory approach as is ours. To solve these problems we used the GridGain In-Memory Accelerator 6.6.4 extension. This extension was developed to optimize the Hadoop MapReduce jobs by replacing its HDFS for an in-memory HDFS, which means that the interactions with the hard drive are replaced with interaction with the remaining of the memory hierarchy. Furthermore, the GridGain extension enables a JVM to have more than one task, or even use a single JVM for all the tasks.

3.3 Programming Model

In this section we describe what the programmers have to implement, so that they can use our cache-conscious block tiling support and adapt it to their applications. Hence, we describe in more detail the implementation of a class that extends `BlockSequenceFileRecordReader` abstract class, by explaining what should be used and which extra requirements are needed so that they can be integrated into an Hadoop MapReduce application.

3.3.1 What has to be Implemented?

A programmer that wants to use our approach has to extend the `BlockSequenceFileRecordReader` class. In order to do it and to express the application's decomposition logic the programmer must implement the methods presented in Table 3.1.

To use `BlockSequenceFileRecordReader` as the base `RecordReader` for an Hadoop MapReduce application, the programmer has to implement the methods related to the reading of the data from the `InputSplit`. These methods are the `copyKey` and `copyValue` methods. The implementation of these methods is strictly required, because our approach does not assume any type either for the key or value fields in a pair. Given that, the programmer must implement these two methods so that from an `Object` general type it can produce a copy of the received object. They were made many experiences that included generic versions of these method, but the resulting performances were very bad.

In order to express the decomposition algorithm of the programmer's application, the programmer must implement the `buildBlock` method. This method is the responsible for receiving the data and produce the pairs, therefore this method generally is the most complex to implement in our approach. Since this method is also responsible for managing the data structure that stores the surplus blocks, it is recommended to separate the logic of build the cache concious blocks and consequently the pairs, from the data structure logic.

The data structure for the surplus pairs is of the programmer's responsibility, both its choice and management. The `BlockSequenceFileRecordReader` only interacts with this data structure by three methods that the programmer must implement. The first method is called `hasMoreBlocks`, and can be implemented has a query to the data structure so that our class knows whether there are pairs remaining to send to the *mapper*. The second method is the `getNextBlock` and returns the next pair to be processed. This method does not need to follow the data structure order of retrieving its objects, but the programmer can implement the order which better fits the application needs. The same cannot be said about the third method, `size`. This is a more strict method, but easier to implement, that can use the data structure methods to get the information about the number of elements stored in the data structure.

The basic solution we propose only takes into account the size of a chosen cache level to see the size of the data partition read from the split, but other approaches might be desired. To modify this logic of our solution the programmer can modify the `getPartitionSize` method. In this method the programmer can access any variable setted in the Hadoop `Configuration` of its application by using the variable `conf` and with it implement the desired algorithm to find the optimal partition size. It is worth to mention that this implementation is completely optional, so any modification is of the programmer's responsibility.

3.3.2 Configuration Requirements

The implementation and utilization of these classes requires some extra information, so that the cache-conscious decomposition is possible. With this, the programmers must add some more information about the cache hierarchy of a given machine to the `Job Context`, enabling the use of our classes in their applications. These additions are related to the definition of two fields in the `Context` object and their names are *cache.level* and *hierarchy.file*. The first field, *cache.level*, informs our classes which cache level is the target for a given application. This is a optional field that can be used in some implementations to target a specific cache level. The *hierarchy.file* field gives the path of a JSON file. This file is inspired in the ones defined in [26], which contains the cache hierarchy of the machine in use, as can be seen in Listing 3.1. The cache hierarchy files are composed by a well defined structure of JSON nested objects that include the fields:

- child - main object that represents a memory level and contains all its information.

```
{
    "siblings": [[0,1,2,3,4,5,12,13,14,15,16,17],
      [6,7,8,9,10,11,18,19,20,21,22,23]],
    "size": 15728640,
    "cacheLineSize": 64,
    "child": {
      "siblings": [[0,12],[1,13],[2,14],[3,15],[4,16],[5,17],[6,18],
        [7,19],[8,20],[9,21],[10,22],[11,23]],
        "size": 262144,
        "cacheLineSize": 64,
        "child": {
            "siblings": [[0,12],[1,13],[2,14],[3,15],[4,16],[5,17],[6,18],
              [7,19],[8,20],[9,21],[10,22],[11,23]],
            "size": 32768,
            "cacheLineSize": 64,
            "child": null
        }
    }
}
```

Listing 3.1: JSON file with the cache memory hirarchy.

If its value is null then the current memory level is the bottom-most one

- siblings - represents the siblings cores that share a memory level
- size - size of the memory level (in bytes)
- cacheLineSize - size of the cache coherency line (in bytes), used only for memory levels that represent a cache level

Listing 3.1 shows the representation of a cache memory hierarchy composed by three levels. On the more general level, the *L3* level, we have two caches of 15 megabytes with cache line of 64 bytes, wherein each cache is shared by half the cores. The *L2* cache level has 12 cache each shared by two cores and has a capacity of 256 kilobytes and a cache line of 64 bytes. The level closer to the processing cores, the *L1* cache, is also shared by two cores at a time and has a capacity of 32 kilobytes with a 64 bytes cache line.

Both of these fields are very important in our solution either in the decomposition solution, presented in this chapter, or in the stencil API presented in Chapter 4, but in different contexts. In this solution they have to be specified by the programmer. However, the stencil API integrates these fields in the configuration of the applications, via a configuration file such as the one presented in Listing 4.4.

3.3.3 Implementation Example

To give a better idea of how a decomposition algorithm can be implemented using our `Recordreader` as a base class, we present one implementation that is used to decompose input data represented as a matrix. With this, we have implemented and extended

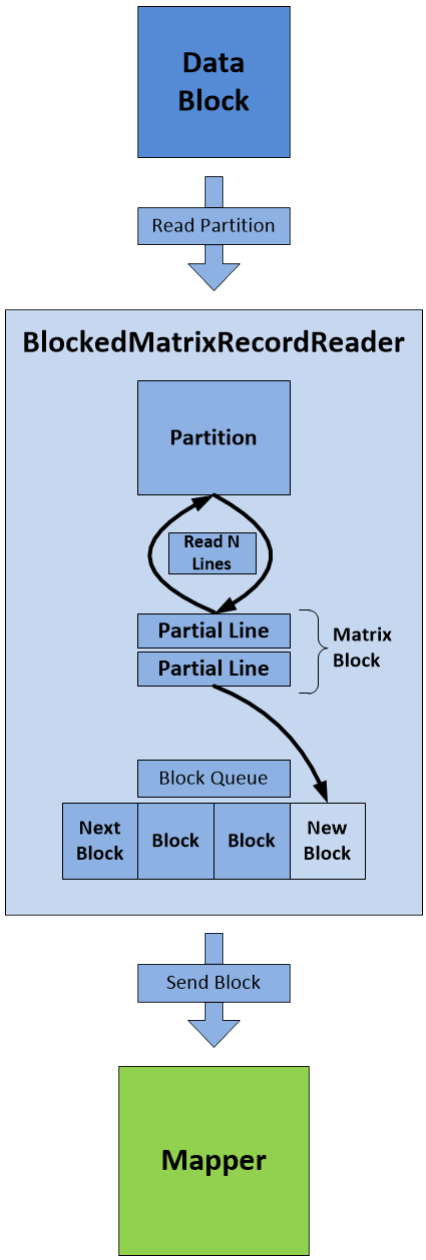

Figure 3.4: `BlockedMatrixRecordReader` block decomposition.

the `BlockSequenceFileRecordReader` that handles matrices that is called `BlockedMatrixRecordReader`. The objective of this class is to handle matrix-like inputs and produce key/value pairs, where the value field represents a block of matrix's elements. This class can be used in applications/computaions that have as its input a single matrix, for example stencil applications or single matrix operations. In the implementation of `BlockedMatrixRecordReader`, only the required abstract methods (and some auxiliary) were implemented, which means that the `getPartionSize` remained remained untouched. Given that, to better understand how we have implemented the extended class, we go through the execution presented in Figure 3.4.

The `BlockedMatrixRecordReader` class was developed for inputs that represent matrices of *float* values, mainly to get a more concrete example. However, the developed class can be adjusted in order to support any other type of elements without modifying its main concept. The input is represented by a `SequenceFile` that is composed by two types objects, one for the key and the other for the value. The key objects are of the type `LongWritable` and represent a number of the matrix line. For the values, these objects are `BytesWritable` which store the content of a line in bytes. As can be seen in Figure 3.5, the actual content of the `BytesWritable` objects are `n floats` for this concrete example.

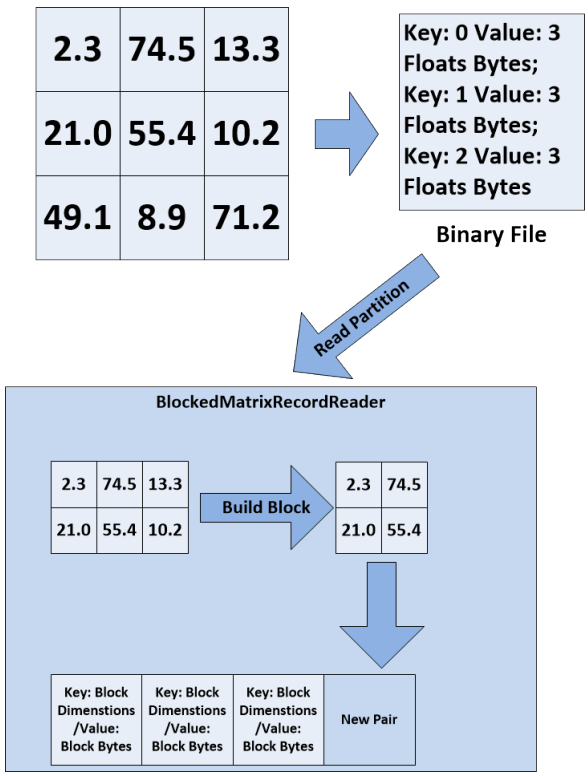

Figure 3.5: Block Matrix `RecordReader` pair production example.

The normal execution of the `BlockedMatrixRecordReader` class starts by reading a partition from the respective `InputSplit`, as shown in Figure 3.4. The process of getting this partition is performed by reading key/value objects until a limit is reached. This limit

represents a quantity of bytes that is determined by the `getPartitionSize` method. For the `BlockedMatrixRecordReader` class, it is used the default implementation that gets the size of a given target cache level. A mechanism that is not represented in Figure 3.4 is the process copying the objects read from the split to the pairs internal list that is passed to the decomposition algorithm. This process is applied to every pair read from the split using the `copyKey` and `copyValue` methods. In this case, the `copyKey` method receives an `Object` object and copies a `LongWritable` from it, then this copy is returned as a `Object` object to the `BlockSequenceFileRecordReader` base class. The `copyValue` method works in the same way, but instead of getting a `LongWritable`, it gets a `BytesWritable` object.

After getting the list of pairs that amount to at most the target cache level size in bytes, the decomposition algorithm is triggered through the invocation of the `buildBlock` method. The decomposition algorithm for `BlockedMatrixRecordReader` is based on two previously calculated values, the number of rows and columns for a desired block/tile of the input matrix. These dimensions are used together with the internal list of pairs to produce the key/value pairs that are handed to the *mappers*. As shown in Figure 3.4, we have to read n rows of our matrix which really means reading `n` pairs from the list, because of the content of value field is indeed a line. In Section 3.3 we advised the programmer to separate and decouple each part of the decomposition algorithm, and also to separate most of this logic from the data structure management. The decision of maintaining a single method was taken due to the unpredictability of possible implementations. In other words, we could not assume any structure either for the decomposition logic or for the data structure and its management, as it could limit the range of possible implementations. For our approach, we have separated the decomposition into two main methods presented in Table 3.2. As a result, the process of reading n lines is implemented by the `getNLines` method.

To produce the actual blocks the implemented solution passes the `n` lines to the `createBlocks` method, which decomposes the set of lines according to the number of columns for the regular tile/block of the input matrix. When a tile/block with the specified dimensions is produced, the `createBlocks` method builds a key/value pair, whose key field contains the block dimensions and whose value field is the data tile/block itself. The resulting blocks may have different dimensions (as is exemplified in Figure 3.5). Hence, the *mappers* that will process these blocks have to know theirs dimensions. To tackle this problem we use the key field of the handed key/value pairs to store the dimensions of the produced blocks, thus avoiding possible errors in the application level.

In the end of the decomposition logic, the produced key/value pairs are added to the data structure that stores the surpluses pairs. This data structure chosen for this implementation is a *FIFO* queue, mainly because it seemed the most natural behaviour for the applications that would use our class. The name used for data structure management method `getNextBlock`.

Something that is very important when extending and using the `BlockSequenceFileRecordReader` is the type of the objects used for the key/value pairs, which must be

Method	Description
List<Tuple<LongWritable,BytesWritable>> getNLines(long n,Queue<Tuple<K,V>> pairsQueue)	This method reads n lines of a matrix. Each element of keysQueue represent a line in a matrix and the its values are stored in the same position in the valuesQueue.
void createBlocks(List<Tuple<LongWritable, BytesWritable>> blocksLines)	CreateBlocks it is the method responsible for the actual creation of the blocks. This is done by receiving n lines of matrix and according with the given regular block dimensions these blocks are created.

Table 3.2: `BlockedMatrixRecordReader` class methods (not included methods inherited from the base `BlockSequenceFileRecordReader` class).

in accordance with the types received by *mappers*. This situation must be carefully done, mainly to prevent application malfunctions. For our example, we chose both for the key and the values the `BytesWritable` type, mainly because this type is easily read and its does not add more dependences. This is not a problem created by our solution, but a inherited problem from the version of the framework used at the time of development.

3.4 Final Remarks

In this section we have presented our solution for the decomposition of the application's data using block-tiling strategies, with some support for the inclusion cache friendly strategies. This approach was implemented in the Hadoop MapReduce framework which proved to be well fitted for our experimentation, even with the need to add the GridGain In-Memory Accelerator extension to take Hadoop to an in-memory context. This framework revealed itself to be easy extensible, wherein the development of new classes where always based on base classes made available by the framework. Moreover, the framework's popularity and extended community helped in numerous situations, which could have delayed the work.

The next chapter will present our study case to test and validate our decomposition solution. As our objective targets applications that have a natural tendency to benefit from the concept of temporal locality, we choose the case of stencil computations to experiment our solution. In addition to the reasons already mentioned, the choice of using stencil computations derives from its great importance in many scientific applications and also because of the absence of support to this kind of computations in the Hadoop context.

Chapter 4

A Programming Model for Stencil MapReduce Computations

Stencil computations are quite important in the field of scientific computing, which by itself makes the construction of an API dedicated to this type of computations useful. Moreover, stencil computations also posses characteristics that can be used to validate the cache-conscious block-tiling decomposition solution we have proposed in the previous chapter. These computations are generally embarrassingly parallel which is suitable for MapReduce, and also they benefit from temporal locality, where a computation of a element's stencil has to visit its neighbours in a short span of time. Expressing stencil computations using MapReduce in order to make the most of its computing in the *map* phase can be a non trivial task, due to the management of the neighbourhoods. Moreover, these tasks do not get any easier when we try to map these neighbourhoods into the memory hierarchy of a given machine. In this context, the development of a stencil directed API that makes use of our locality enhanced MapReduce framework is a relevant and useful contribution. Hence, as a way to validate our solution and to test our concepts we developed an API that expresses stencil computations in the Hadoop MapReduce model.

4.1 Stencil Computations

In general, stencil computations can be described as a piece of code which represents a given pattern to be applied to an array of elements. In other words, a stencil is a computation based on a pattern that is applied to all the elements of a multi-dimensional vector, such as a matrix (including images), and that generally depends of the neighbourhood of each element. We begin this section be presenting some relevant work in the parallelization and optimization of this kind of computations. Then we discuss the way that the

stencil computations can be mapped to the context of a MapReduce application.

4.1.1 Optimization of Stencil Computations

The optimization of stencil computations has been a topic of some research that tries to find better mechanisms or strategies to improve the performance of this kind of computations, so that the hardware is better availed whether in terms of memory, or in terms of processing units. Herewith, this type of research has these similarities with our work, as can be seen in studies such as [14], or [30].

In the study presented in [14], shows that the low performance of this kind of computations can be related to the disparity of speed between the main memory and the processing units. With this, the study evaluates the usage of strategies such as block-tiling and time-skewing as an attempt to exploit some cache locality, trying to exploit both temporal locality and spatial locality. However, some cache-oblivious algorithms are used and compared with cache-aware algorithms. This is done mainly by comparing the stencil performance with both approaches and to verify if the cache-oblivious algorithms can be a feasible strategy. In this work it is done a extensive evaluation on the type of stencil used, testing the impacts of the mentioned strategies to single iteration stencils and multiple iterations stencils, wherein the first type does not possess the characteristics to employ a time-skewing strategy.

A more recent study presented in [30], shows a similar approach to the above mentioned where the block tiling and time-skewing strategies are used to study the performance improvement of stencil computations. However, in this study the main focus goes to the combination of strategies that not only improve locality, but also try to improve parallelization. Whence some strategies such as wavefront parallelization or pipeline parallelization are employed for multi-iteration stencils.

Both the aforementioned works resort to strategies such as block-tiling or time-skewing, but in the context of this work only block-tiling is useful. The time-skewing strategies are only useful for iterative stencils and the focus of this work is in the optimization of a single iteration, mainly because it is this behaviour that allows us to validate the developed solution. However, as reported in both papers the block-tiling strategy can achieve some good results, improving the stencil computations performance and therefore the usage of this strategy can be useful for a better avail of the memory hierarchy.

4.1.2 Stencil Applications with MapReduce

Since the stencil computations operate on neighbourhoods, they benefit substantially from temporal locality. Thus they become good candidates to take advantage of our block-tiling approach that produces key/value pairs containing data blocks, improving its memory usage and consequentially its performance. Yet, if we want stencil computations as the type of applications that can test and validate our solution, then it is necessary to find a way to express this kind of computations in the MapReduce model.

From the characteristics of the stencil computations, we can extract some actions, such as the construction of an element's neighbourhood and the application of an operation to those elements. These actions could be expressed in a way where the neighbourhood construction is done in the *map* phase and the computations of the values could be made in the *reduce* phase, as explained in Section 1.1.2. However, this approach can deteriorate the application's performance just by the amount of data transference between the *map* and *reduce* phase. Besides, this approach would not explore much the concept of temporal locality.

Another way of expressing stencil computations is to decompose the stencil operations into an element's value and to its neighbourhood values partial computations. In other words, we could have the computation divided into the main element part and the neighbourhood part, being the latter an operation applied to each neighbour value. If these operations are applied at the iteration of the elements, then they could be expressed in the *map* phase. With this, for each data block received, our application would try to compute the stencil of its elements based on the partial computations it could accomplish. However, the attainment of complete stencil computations is limited by the elements that each data block contains. It may happen that for a given element the current data block does not contain all the elements needed for that stencil computation. Hence, the *map* phase tries to do as much computations as it can, and sends the partial and complete computations to the *reducers* in charge of a given element computation. Still in this second approach, the *reducers* are still responsible for the final computation of an element's value. Being that each *reducer* has multiple elements assigned to it, but only computes each element value at a time. Given that, a *reducer* receives a set of partial results related to a given main element and then aggregates them into a final result, later moving on to the next main element computation.

These strategies can be applied either for the *line* approach or the *block* approach, as mentioned in Section 3.1. However, if the *mapper* receives a data block, it can perform more computations, which on one hand will increase parallelism and on the other will allow us to use the cache-conscious block-tiling decomposition to leverage the cache hardware and so improving the temporal locality of the *map* phase. To effectively benefit from temporal locality, the data read from the input must persist in memory across the execution of multiple map functions. Only so it will be revisited multiple times.

The purpose of this part of our work is to validate the developed decomposition solution using a single iteration of stencil computations, mainly because it is this best suited context to do it. However, stencil computations are iterative by nature. Even though this is not the focus of our work, the iterative behaviour of a stencil can be emulated by the chaining of multiple MapReduce jobs, where the input of a job is the output of the previous one. This can be done with a single loop that has a stopping condition, dependent of the specific stencil, and also with some intermediate data structures if necessary. Yet, has said before this is not the context of our work, so we will not delve into it further.

4.2 Stencil API

A stencil computation in MapReduce can be concisely expressed with a full or partial computation of an element's stencil in the *map* phase, leaving the aggregation of the possible partial computations and the output of the final results to the *reduce* phase. The *map* phase has the responsibility of receiving the data blocks and compute the stencil of as many elements as possible. However, the block received by the *map* phase only contains enough data to compute the stencil of the elements that have all their neighbourhoods in the block. When this situation occurs then there are two possibilities: 1 - the necessary elements for the full computations of stencil's element was already processed and, hence, they are cached in a intermediate data structure; 2 - the elements need for a given element stencil are not available, and in this case the stencil computation is left to the *reduce* phase.

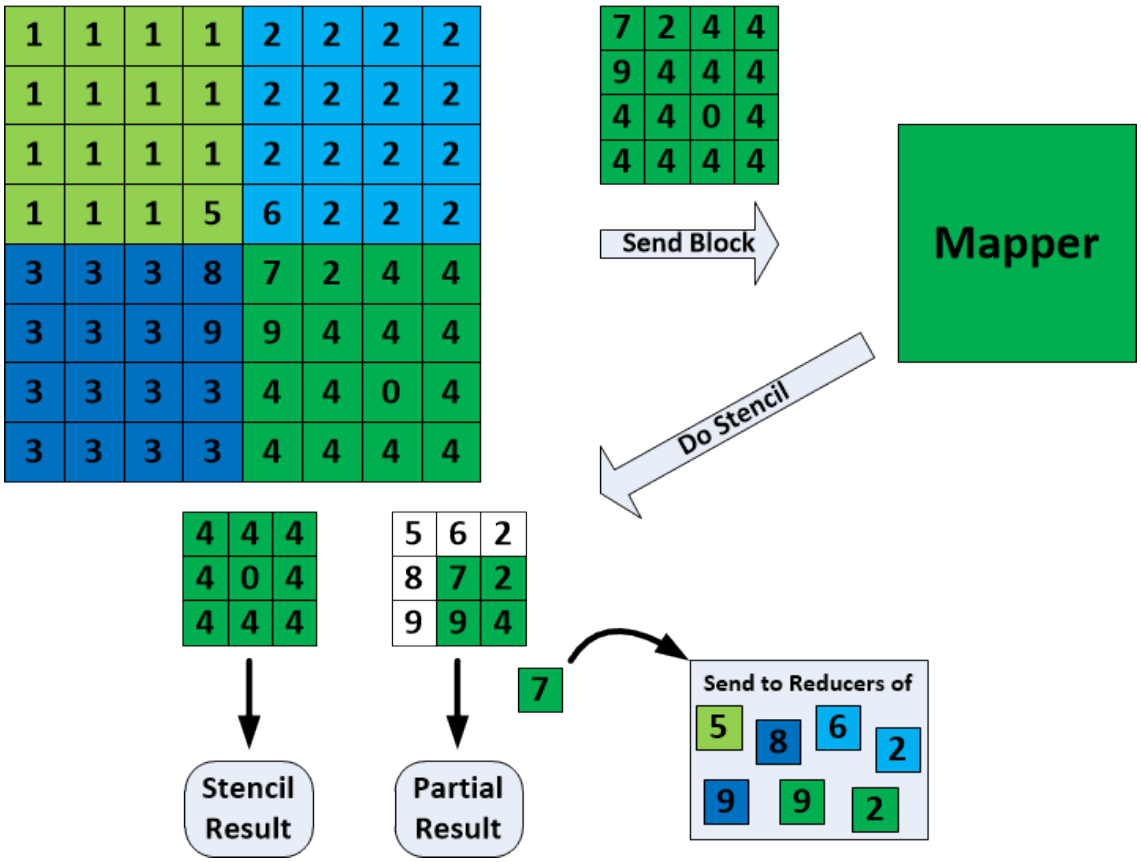

Figure 4.1: Example of stencil computation applied to a matrix. The neighbourhood is composed with elements that are within a element reach.

In both previous cases the the element for which the stencil is being computed has to be sent to the *reduce* phase, mainly because it is needed for the completion of the stencil computations of the elements that are out of the block currently being processed and which have this element in their neighbourhood, as shows in Figure 4.1. In the first possibility, the portion of the neighbourhood that is out of the current block has already

been read and cached in an intermediate data structure. This means that the stencil of these elements were left for the *reduce* phase, since they needed the block that is only now being processed.

The second possibility happens when the current block is read before the neighbour and, hence, this block has already been cached in the intermediate data structure. This may be true for many situations, but is not guaranteed because several mappers can be running in parallel. The fact that the *mappers* are executed in parallel influences the behaviour related with the intermediate data structure. This data structure is shared by all the *mappers* and, as a block arrives to a *mapper*, the elements in that block are stored so that more computations can be achieved in the *map* phase. With this, it is necessary to do some concurrency control to the data structure access, which means that the parallelism of the computations is decreased.

The *reduce* phase is less complicated or, in other words, more thin than the *map* phase. This phase has to distinguish the partial computations from the full computations, so the two scenarios can happen. The first derives from an incomplete or partial computation of an element's stencil. In this situation the *reducer* has to iterate over all the partial computations or elements, compute the final result and do its output. In the second situation we have a simpler scenario, where the *reducer* has to identify the final result and do its output. Yet, with the optimizations and the cache-like data structure, the second scenario is the most usual. Hence, the overhead of iterating over partial results happens mainly for the elements of the first processed data blocks, therefore being a minority.

Being the concepts applied in *map* phase intrinsic to the majority of stencil computations, we can refer to them as the *stencil logic*. Hence, to abstract this *stencil logic*, in order to reach the majority of stencil computations, the programmer has to provide to our API the way a stencil computation applies a stencil to an element and to its neighbours, and which elements of a given input are subjected to this stencil. To achieve this, we developed some abstract classes that help to translate a stencil computation into a MapReduce computation.

4.2.1 API Classes

In Figure 4.2 it can be seen the layer model of our solution. The first layer is related to the original Hadoop, where our solution is based on. The second layer represents the classes added to Hadoop to handle our cache-concious decomposition solution, presented in Chapter 3. The last three layers are related with classes developed for our stencil API. The first layer shows the base classes, which represent the base for all MapReduce stencil computations. In the second layer we have the classes needed for a stencil application that receives a matrix as its input. Finally the last layer, the application layer, which is where the programmer's application classes are contained. To use our API the programmer has to extend three major classes related to the *map* phase, to the stencil computation logic and to the stencil driver application. In the next sections we present the API classes, by

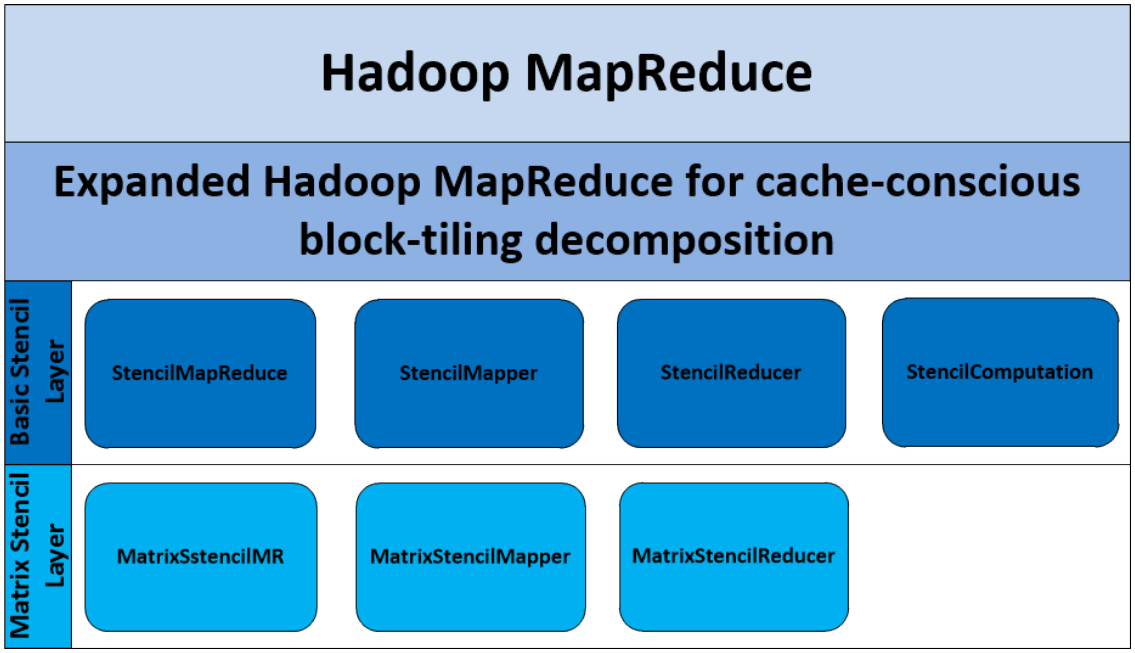

Figure 4.2: API classes layer model. In the first column are the driver application classes; On the second column are the classes related to the *map* phase; Third column has the classes related to the *reduce* phase; Fourth column is related to the stencil computation classes.

going through the columns presented in Figure 4.2.

4.2.1.1 Map Phase Classes

The API classes related to the *map* phase are depicted in the second column of Figure 4.2. The base class for the *map* phase is the `StencilMapper` abstract class that extends the Hadoop `Mapper` class and overrides its main methods: `setup`, `cleanup` and `map`. Its main objective is to offer a way to read the data blocks and a basic set up for the *map* phase. As shown in Table 4.1, this abstract class asks the programmer to implement some methods such as the `readElement` which tells the class how to read the value of an element from a `ByteBuffer`, the `stencilMapLogic` that tells how the data block has to be iterated and the `compute` method which basically applies the stencil computation logic to each element.

Being a generic class for the map phase, `StencilMapper` does not embed information about how the input data is organized or, in other words, how the elements are arranged and how to apply its computation correctly. The corresponding class in the Matrix support layer, `MatrixStencilMapper`, is responsible for setting up two processes: setting up the environment of the *map* phase and also includes all generic behaviours required to apply *stencil logic* to a matrix.

For the configuration of the *map* phase, the `MatrixStencilMapper` uses the `setup` method (inherited from the Hadoop's `Mapper` class) to: 1 - initialize the cache-like data

Method Signature	Description
void map(BytesWritable blockDims, BytesWritable value, Context context)	Method inherited from Hadoop's `Mapper` class
T readElement(ByteBuffer inputBuffer)	Abstract method that should define the way that an element is read from a `ByteBuffer`, where T is the generic type that represents the type of values handle by the stencil
void stencilMapLogic(Context context)	Abstract method that should implement the stencil logic, therefore the sending of pairs to the reducers must also be included in the implementation of this method
void compute(C coordinate, boolean maintainValue)	Abstract method that applies the computations implemented by user classes to the input elements and writes the complete or partial results to the reducers, The type use to represent the coordinates of element are represented by a class C
int getFirstLineIdx()	Returns the index of the line where the block starts
int getLastLineIdx()	Returns the index of the line where the block ends
int getFirstColumnIdx()	Returns the index of the column where the block starts
int getLastColumnIdx()	Returns the index of the column where the block ends

Table 4.1: `StencilMapper` class methods.

structure with a value returned by a programmer implemented method called `getMatrixInitialValue`; 2 - perform a instantiation of the class that offers the logic of the stencil computation, which is presented later in this section.

The core of the *stencil logic* has to be expressed in the `compute` method, presented in Table 4.2. This method is inherited from the `StencilMapper` and, depending on the extension of the latter class, can have different implementations. However, independently of the logic implemented by a class that extends `StencilMapper` (as the `MatrixStencilMapper`) the programmer has to implement methods such as: the `getMatrixInitialValue` that tells the class with which value initializes the elements of the data structure; and the `isInitialValue` which helps to verify if a given value is the default. These methods have to be implemented by the programmer's class that extends `MatrixStencilMapper`. However, the most important dependence of this class is related to the programmer's extension of the class `StencilComputation`.

Method Signature	Description
M getMatrixInitialValue()	Abstract method that has to give the default value for the empty matrix
boolean isInitialValue(T value)	Abstract method that must verify if a given value is equal to the default value for a empty matrix, where T is the type of the value handled by the stencil
T readElement(ByteBuffer inputBuffer)	Method inherited from `StencilMapper` class, that uses the `StencilComputation` method with the same name to read from a `ByteBuffer`, where T is the type of the value handled by the stencil
void compute(Tuple<Integer, Integer> coordinates, boolean maintainValue)	Abstract method inherited from `StencilMapper` class

Table 4.2: `MatrixStencilMapper` class methods.

4.2.1.2 Stencil Computation Classes

The `StencilComputation` abstract class, whose interface is presented in Table 4.3, offers to the *map* class the *stencil logic*. As mentioned in Section 1.1.2, this logic can be expressed in two components: the main element component and the neighbourhood component. However, the way to aggregate these components it is not always the same. Hence, to express the *stencil logic* we have the main element component, the neighbour element component and the aggregation component, which specifies how the first two components are aggregated to get the final result. With this, the programmer must extend the `StencilComputation` in order to express its *stencil logic*, mainly by implementing the `mainElementComputation`, `neighbourComputation` and `aggregateComputations` method, which are related with three perspectives mentioned earlier in this section. Besides offering the stencil computation logic to the `compute` method, the `StencilComputation` class offers the auxiliary methods such as the `readElement` and `writeElement` which translate elements used in the `compute` method from and to byte form.

As can be perceived in Table 4.3, there are two remaining methods, the `serConfiguration` method and the `newInstance` method. The first is related with the setting of the application's `Configuration` object that may contain some programmer specified field useful for the stencil computation. The `newInstance` method provides the means for the framework to create new instances of the class by using the factory pattern.

Method Signature	Description
void setConfiguration(Configuration configuration)	Method responsible for setting the `Configuration` object for the computation class
StencilComputation<C, N, T> newInstance()	Abstract method that has to create a instance of the class that extends the StencilComputation
T mainElementComputation(C mainElementCoord,T mainElementValue)	Abstract method that is responsible for the partial calculations made with the main element of a stencil for this same element
T neighbourComputation(C mainElementCoord,T mainElementValue, C neighborCoord, T neighborValue,int numberOfNeighbors)	Abstract method that implements the partial calculation applied to a neighbour for the stencil calculation of the main element
T aggregateComputations(Tuple<Integer, Integer> mainCoords, T finalResult, Tuple<Integer, Integer> neighborCoords, T partialComputation)	Abstract method that implements the operation that allows the "sum up" of the partial calculation
T readElement(ByteBuffer inputBuffer)	Abstract method that should define the way that an element is read from a `ByteBuffer`
byte[] writeElement(T element)	Abstract method that should define the way that an element is turned into a byte array

Table 4.3: `StencilComputation` class methods. In these method the C generic class stands for the class that identifies the position of an element in relation to the others. The T generic class represents the class type used for the elements, And finally, the N generic class is related to the intermediate values stored in the internal cache like data structure.

4.2.1.3 Reduce Phase Classes

With regard to the *reduce* phase, our API offers the classes presented on the third column in Figure 4.2. The presented hierarchy starts with the base abstract class `StencilReducer` that basically extends the Hadoop `Reducer` class and defines the types that should be used for the input key/value pairs and the types for the output pairs in the `reduce` method, as can be seen in Table 4.4. As for the classes of the *map* phase, we have implemented an abstract class called `MatrixStencilReducer`. This class employs a strategy that distinguishes the received pairs as complete stencil computations or only partial computations. Table 4.5 presents the method that can be modified. The *reduce* method evaluates the

Method Signature	Description
void reduce(StencilPair.Key key, Iterable<StencilPair.Value> values, Reducer<StencilPair.Key, StencilPair.Value, Text, Text>.Context context)	Abstract method that predefines the types of the keys and values the *reduce phase* receives. This method overrides the `reduce` method from the Hadoop's `Reducer` class

Table 4.4: `StencilReducer` class method.

Method Signature	Description
void reduce(StencilPair.Key key, Iterable<StencilPair.Value> values, Context context)	Method inherited from Hadoop's `Reducer` class

Table 4.5: `MatrixStencilReducer` class method.

type of the received value and if it is received a complete computation it goes right to the output part of the method. Otherwise, it iterates over the partial computations to aggregate them and get the final result. In the `MatrixStencilReducer` implementation only the `setup` method (inherited from the Hadoop's `Reducer` class) has some relevant responsibilities. As its homonym from the *map* phase, the `setup` method has to instantiate and configure an object of the `StencilComputation`.

4.2.1.4 Driver Application Classes

Apart from the MapReduce main phases, it is necessary to have a driver application that configures and launches the stencil computation MapReduce job. Therefore, our API has the abstract classes presented on the first column in Figure 4.2 to help with this task.

The base abstract class `StencilMapReduce`, has the job of defining the classes and the file system paths that are to be used by the application job. The classes defined in the `StencilMapReduce` class are mainly composed by the ones we developed for our decomposition solution, for example the `BlockSequenceFileInputFormat` class from Chapter 3. Besides, as can be seen in Table 4.6, this class offers a method called `launch` that uses some parameters to help configuring the job and, consequently, launch it.

Proceeding with the same reasoning used for the *map* and *reduce* phase, the support for matrix input applications it is offered by the abstract class `MatrixStencilMR`. The constructor presented in Table 4.7 receives some information about the input matrix, a configuration file (that we will talk about later), and about the main MapReduce classes to be used in the job. Using its parent constructor, the `MatrixStencilMR` constructor configures the job by setting some global variables and getting some information about the environment in which the application will run such as the number of processing cores available. The configuration done in this class includes the calculation of the normal

Method Signature	Description
<Mapper extends StencilMapper<C, T>, Reducer extends StencilReducer<T>> StencilMapReduce(Class<Mapper> mapperClass, Class<Reducer> reducerClass)	Class constructor which sets the fields related to the application *map* and *reduce* classes, where C stands for the class type that identifies the position of an element in relation to the others and T for the type of values handled by the stencil
Configuration launch(String id, String inputPath, String outputPath)	Method responsible for setting up the application job and launching it according to the received parameters

Table 4.6: `StencilMapReduce` class methods.

Method Signature	Description
<S extends StencilComputation<Tuple<Integer, Integer>, M[][], T>, Mapper extends MatrixStencilMapper<M, T, S>> MatrixStencilMR(long numberOfLines, long numberOfColumns, long inputSize, Configuration othersConfigs, Class<Mapper> mapperClass, Class<S> computationClass, String pathConfig)	Class constructor that uses its parent class constructor and the received parameter to configure the application's job. The M generic class stands for the class type used for the values store in the internal cache-like data structure. The T is related to the type used for the stencil values, both for the input and output. The S generic type provides to the framework the class type of the class that extends the `StencilComputation` class

Table 4.7: `MatrixStencilMR` class methods.

dimensions of a data block for our application, taking always into account the characteristics of both the architecture and the input.

4.2.2 Programming Model

In this section we present our API programming model, explaining what must be implemented and extended with an example of the stencil computation that we presented in Section 1.1.2, which has its classes represented in Figure 4.3. To close this section we go through the configurations that can and have to be done.

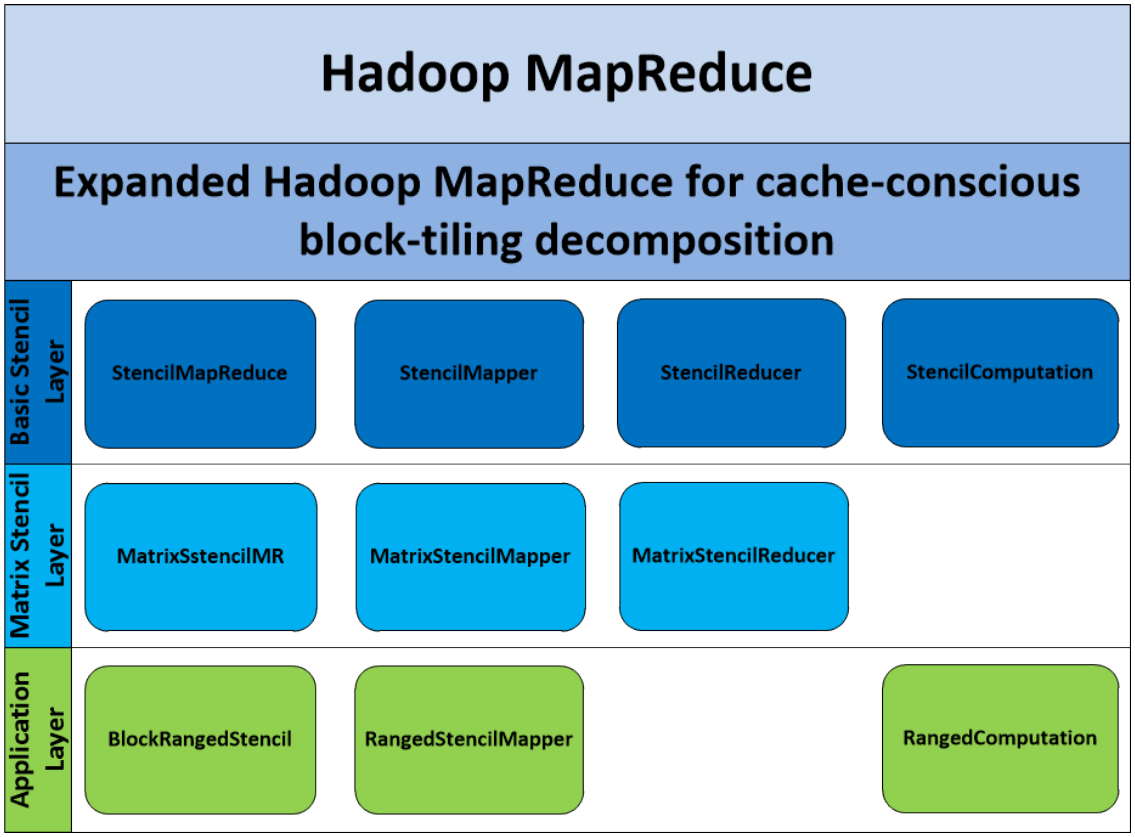

Figure 4.3: API classes layer model with application classes in green

4.2.2.1 Implementation Example

The present section serves to show what it takes for a programmer to develop a stencil application whose input is matrix based. To start with, as already said, the programmer must extend three types of classes, these being the `MatrixStencilMapper` for the *map* phase, the `StencilComputation` to express the stencil computation and also the `MatrixStencilMR` to configure and launch the stencil application.

In order to better understand the programming model of our API, we show the implementation of the mentioned classes for the example presented in Section 1.1.2. The example application uses a stencil that applies a computation to all the elements of a matrix. For each element, the stencil is calculated based on a percentage of the neighbours values and on a percentage of its on value. Remembering the equation from Section 1.1.2 we have:

$$M[i,j] = M[i,j] \times percentage + \sum_{ni=-r}^{r} \sum_{nj=-r}^{r} M[i+ni, j+nj] \times \frac{1 - percentage}{(2r+1)^2 - 1}$$

For our example we assume that the input is composed by a matrix of float number values that are stored in binary `SequenceFiles`, where each pair is composed by a number representing a matrix's line number and a sequence of bytes that represents n float

```
1  public class RangedStencilMapper extends MatrixStencilMapper<Float,
2   Float, RangedComputation> {
3
4    @Override
5    protected void stencilMapLogic(Context context)
6        throws IOException, InterruptedException {
7
8      for(int i = getFirstLineIdx(); i <= getFirstLastIdx(); i++)
9        for(int j = getFirstColumnIdx; j <= getLastColumnIdx; j++)
10         compute(new Tuple<Integer, Integer>(i,j), false);
11   }
12
13   @Override
14   protected Float getMatrixInitialValue() {
15     return Float.NaN;
16   }
17
18   @Override
19   protected boolean isInitialValue(Float value){
20     return Float.isNaN(value);
21   }
22 }
```

Listing 4.1: RangedStencilMapper class.

numbers of that line. With this set up we will start first by analyse the class related with the *map* phase, then we see an example of StencilComputation extended class and lastly we present the driver class for the application.

The programmer must implement three methods to extend our MatrixStencilMapper class, as the example in Listing 4.1 shows. Two of those methods are trivial to implement and are related to the default value with which our caching data structure of the *map* phase is initialized. The first of those two methods is getMatrixInitialValue, which basically indicates to the *mappers* the value to be used in the data structure initialization. The second method is isInitialValue method that simply compares a given value with our default value. This method is very important for the *mapper* to verify if a given element is cached. In Listing 4.1, we can see that we are using floats for the input elements, because of this we use the NaN float value to initialize the data structure and use its comparator method to implement our isInitialValue method. The choice of using the NaN value is due to the fact that we do not want to limit the range of numbers used as input.

The most important method in the class that extends MatrixStencilMapper is stencilMapLogic. As mentioned in previous sections this method indicates to the *mapper* to which elements will be applied the stencil, or in other words, which elements in the received data block are to be applied the method compute offered by the MatrixStencilMapper class. Depending on the *stencil logic* some specific elements can be ignored, as can be seen in Listing A.1. However, our stencil computation is applied to all input elements, therefore we only have to iterate over the indexes of the elements present in the data block. For this, as can be seen in Listing 4.1, we use the block limits methods,

presented in Table 4.1, to indicate the set of elements used. It is worth mentioning that the `boolean` value in the `compute` method indicates if the element must maintain its original value. This addition to the `compute` method can be very handy in some stencil computations such as the SOR computation, which its implementation using our API can be seen in Section A.1. With the implementation of these methods the programmer expresses one part of the stencil computation logic, but there is some more details in this class that deserve our attention. In the class signature the programmer must indicate the type of the input elements and the type of the elements stored in our caching data structure. This measure is necessary so that our abstract classes remain as extensible as possible. In our example we indicate both types as `Float` and an extra type that is also required that enunciates the type of the class that extends the `StencilComputation`.

For the programmer to extend our `StencilComputation`, he/she has to decompose its stencil computation into three methods that express the stencil for the main element, for its neighbours and how these partial computations are aggregated into the final result. To better understand this decomposition we look at our implementation in Listing 4.2. There we can see that our implementation of the method `mainElementComputation` represents the partial computation of the simple stencil for the main element, wherein the method provides the element's coordinates in the input matrix and its value. Our implementation is quite simple, as is the stencil, and the method only returns the multiplication between the element's value and its assigned percentage.

The `neighbourComputation`, as the name implies, expresses the neighbour part of the computation, wherein the main element's information, the neighbour's information and the number of total neighbours is passed, so that the programmer can implement this partial computation. In our example, once again our computation is fairly straightforward and it does a simple multiplication of the given neighbour's value and the percentage resultant from the division of the neighbourhoods assigned percentage by the total number of neighbours.

The last of these three methods, the method `aggregateComputations`, must express how to aggregate partials results. This method receives the coordinates of the main element of the computation, a partial computation that generally is the aggregation of the partial results until then, and the coordinates and computation partial result for a given neighbour. In our specific implementation of this method we only have to sum the two values, but there are more complicated cases that will be shown in Section A.1.

These were the main methods of the `StencilComputation` class. However, this class must also offer the implementation of the auxiliary methods `readElement` and `writeElement`. As the names suggest, these methods show to the *mappers* and *reducers* how to translate the elements values from bytes to the chosen type and back again to bytes. With this, the implementation of these methods basically can be handle by some simple `ByteBuffer` operations.

In order for the *mappers* and *reducers* to use the `StencilComputation` methods, the programmer has to implement the `newInstance` method, so that they can instantiate it.

```
1  public class RangedComputation extends StencilComputation<Tuple<
2    Integer, Integer>, Float[][], Float> {
3
4    private final float elementPercentage;
5    private final float neighborsPercentage;
6
7    public RangedComputation(float elementPercentage,
8        float neighborsPercentage) {
9      this.elementPercentage = elementPercentage;
10     this.neighborsPercentage = neighborsPercentage;
11   }
12
13   @Override
14   public StencilComputation<Tuple<Integer, Integer>,
15    Float[][], Float> newInstance() {
16     return new RangedComputation(0.5f, 0.5f);
17   }
18
19   @Override
20   public Float mainElementComputation(
21       Tuple<Integer, Integer> mainElementCoord, Float mainElementValue) {
22
23     return mainElementValue * this.elementPercentage;
24   }
25
26   @Override
27   public Float neighbourComputation(Tuple<Integer, Integer> mainElementCoord,
28       Float mainElementValue, Tuple<Integer, Integer> neighborCoord,
29       Float neighborValue, int numberOfNeighbors) {
30
31     return neighborValue *
32     (this.neighborsPercentage/((float)numberOfNeighbors));
33   }
34
35   @Override
36   public Float aggregateComputations(Tuple<Integer, Integer> mainCoords,
37       Float finalResult,
38       Tuple<Integer, Integer> neighborCoords, Float partialComputation) {
39
40     return finalResult + partialComputation;
41   }
42
43   @Override
44   protected Float readElement(ByteBuffer inputBuffer) {
45     return inputBuffer.getFloat();
46   }
47
48   @Override
49   protected byte[] writeElement(Float element) {
50     ByteBuffer buffer = ByteBuffer.wrap(new byte[Float.SIZE/8]);
51     buffer.putFloat(element);
52     return buffer.array();
53   }
54 }
```

Listing 4.2: RangedComputation class.

```
1  public class BlockRangedStencil extends MatrixStencilMR<Float, Float> {
2
3    public BlockRangedStencil(int numberOfLines, int numberOfColumns,
4     long inputSize, Configuration othersConfigs, String pathConfig)
5      throws IOException {
6      super (numberOfLines, numberOfColumns, inputSize, othersConfigs,
7       RangedStencilMapper.class, RangedComputation.class, pathConfig);
8    }
9
10   public static void main(String args[]) throws IOException, ParseException{
11     if(args.length < 6){
12       System.out.println("[nLines] [nColumns] [input size]
13       [input dir] [output dir] [config]");
14       System.exit(0);
15     }
16     //For Optional Configs
17     Configuration conf = new Configuration();
18
19     BlockRangedStencil brs = new BlockRangedStencil(
20         Integer.parseInt(args[0]), Integer.parseInt(args[1]),
21         Long.parseLong(args[2]), conf, args[5]);
22
23     brs.launch("brs", args[3], args[4]);
24   }
25 }
```

Listing 4.3: `BlockRangedStencil` class.

This approach was reached after some consideration about the best way to eliminate the possible dependences related to specific stencil computations. To implement this method, the programmer must provide some class constructor, even if it is a empty one, so that an instance can be provided to the workers. In the `RangedComputation` implementation, it can be seen that again the programmer must specify the type of the cached and input elements. As exemplified by `Float[][]` for the cache examples, indicating the usage of a two dimensional array of floats, and `Float` for the input elements.

Finally, regarding the driver/main application for our stencil computation the programmer has to extend the abstract class `MatrixStencilMR` and, as in the previous classes, has to provide the data type for the elements cached in our data structure and for the input data elements, as our example shows in Listing 4.3. Additionally, this class must have some components to fully take advantage of our API. First, this class must provide a constructor that invokes its parent class constructor, passing crucial information such as the input matrix dimensions, the total size of the input in bytes, a Hadoop `Configuration` object with some specific stencil application configurations/values, the classes for the *map* and reduce phase plus the class of the stencil computation and also the path to the configuration file. This constructor can do whatever the programmer want, but the call to the parent constructor is mandatory. The `main` method, shown in Listing 4.3, only requires two steps, the configuration step and stencil job launch step. The first is related to the instantiation of our application class, where are passed the programmer chosen parameters to the constructor to create an instance of the application ready to be

```
Configurations

#Configuration file for the Hadoop Stencil API
# hirarchy_file - Path to the file that represents the cache hierarchy in JSON
# cache_level - Target cache level
# unit_size - Element size in bytes in a data block

hirarchy_file: hierarchy.json
cache_level: L1
unit_size: 20
```

Listing 4.4: Configuration text file.

launched. As mentioned earlier, the programmer may want to include some more configurations/informations specific for a given stencil job, which can be used in the extended classes. For that reason we pass to the parent class an Hadoop `Configuration` object. This situation gets clearer with more complex examples as the ones presented in Sections A.1 and A.2. The second step uses the instantiation created in the first step to invoke the `launch` method that, as the name suggests, launches the stencil job with a given name and paths for the input/output directories.

4.2.2.2 Configuration Requirements

There are some details that have to be address and are out of the programming domain, but are crucial for the utilization of our API. For example, in the previous sections is mentioned a configuration file, mainly in the form of its path. This configuration file gives to our abstract classes some information about the the environment in which the application will run and also some information about the input data, as shown in Listing 4.4. In regard to the environment where the application will run, this configuration file enunciates two fields, the hierarchy file path and the target cache level. The hierarchy file path indicates to our API the file which contains the structure of the cache hierarchy of the machine in use and it has to be a JSON file with the structure presented in [26]. The second field is related to the cache level to be used as a reference to our optimization, that is represented by a simple `string` which indicates this target cache level. The last field is related to the element size of an matrix element in a data block. These fields are closely related to the decomposition logic presented in Chapter 3, so in order to reduce the amount of parameters that have to be setted by the programmer's application we developed this configuration file.

4.3 Execution Model

In this section we present the execution of an application developed with our stencil API. We start by describing the configuration and launching of an application. After that, we go through all the steps of an stencil computation, from the moment a *mapper* receives a data block from its `RecordReader`, passing by the processing of this data block until the

mapper produces some results, which are then passed to the respective *reducer* to produce a correct stencil result. It is worth mention that the decomposition of the input is omitted, as it is explain in Chapter 3.

Starting at the stencil driver application, a class that extends our abstract class `MatrixStencilMR` invokes its constructor in the `main` method, passing arguments such as the dimensions of the input matrix, the size of the input file, a Hadoop `Configuration` object and the path to a configuration file. Besides this information, the types of the classes that extend our *map*, *reduce* and stencil computation API classes are passed to the constructor of its parent class, as can be seen in Listing 4.3. The `MatrixStencilMR` class uses its parent class (the `StencilMapReduce`) to define the variables that store the type of classes for the two main phases of MapReduce. After that it uses the `Configuration` object passed, and starts defining some global variables used by the application in each phase of the job. Between these settings, are calculated the dimensions to be used for the data blocks that will be fed to the *mappers*. This calculation takes into account the size of the target cache, the size of an element of the matrix and the dimensions of the input matrix. Returning to the job configuration, it is defined the number of splits that reflects the number of *mappers* always taking into account the number of available processors for a better load balancing. Also, the type of the programmer's class that extends our `StencilComputation` class is defined as a global variable so that the *map* and *reduce* classes can apply the stencil computation. After this configuration the programmer application as to invoke the `launch` method that sets the classes to be used and input/output paths, and finally launches the Hadoop application, as shown in Table 4.6.

Before the *mappers* start to ask for data blocks, the Hadoop Mapper `setup` method is invoked to set up some configurations for all *mappers*, such as the instantiation of the `StencilComputation` extended class, the definition of the auxiliary variables related to the range of the neighbourhoods and matrix dimensions. Also, the initialization of the two dimensional array, that serves as cache, is done with a default value defined by the `StencilComputation`.

When a *mapper* receives a data block from its `RecordReader` the `map` method unpacks the blocks dimensions that are in the `BytesWritable` key. With this dimensions, it iterates the elements packed in the `BytesWritable` value, caching these elements in the global caching two dimensional array. After that, the `stencilMapLogic` method is invoked and the data blocks are iterated again to apply the stencil computation, as exemplified in Section 4.2.2.1.

The computation is applied to each element by the method `compute` implemented in the class `MatrixStencilMapper`. In this method an element is represented by its coordinates on the input matrix and by its value, then with this coordinates are calculated the limits of its neighbourhood and how many neighbours it contains. When these calculations are done, the element's neighbourhood is iterated to attempt to compute the complete stencil value. As it iterates over the neighbours, the *mapper* verifies if these values are in the received block. If this is not the case, the *mapper* sends the main element's

value to the neighbour *reducer*, as described and justified in the beginning of Section 4.2. Besides this verification, the *mapper* verifies if the neighbour's value is cached, if it is then proceeds to aggregate its partial computation to the final result accumulator. Otherwise, the latter step is ignored. Later, when the *mapper* finishes iterating over the element's neighbourhood, it verifies if the complete calculation was accomplished. If that is the case, the *mapper* send the complete computation to element's *reducer*, if not it sends the element initial value.

There are some points in this *map* execution that are fundamentally related to the `MatrixStencilMapper` and `StencilComputation` extensions class of the programmer. The first is related to the verification of the presence of a neighbour's value in the caching data structure, wherein this verification is done using the method `isInitialValue` in the `MatrixStencilMapper` extended class. The second situation has to do with the `StencilComputation` programmer's implementation of the methods `mainElementComputation`, `neighbourComputation` and `aggregateComputations`, which as the naming suggest applies and aggregates the stencil computation for a given element, as shown in Listing 4.2.

The *reducers* also use the `setup` method offered by the Hadoop `Reducers` class to instantiate the `StencilComputation` extended by the programmer. After the `setup` method and when the *reducer* receives the values for the stencil computation of an element, the *reducer* starts to iterate over the received values and starts to discriminate these values. If the *reducer* finds a value marked as complete it outputs it. Otherwise, it keeps adding the received value to a list. When the first iteration is done, the list is iterated with the objective to do and aggregate the partial results into the final result. This two-way iteration strategy has to be done because we do not know if a complete result was obtained in the *map* phase and also to prepare the accumulator variable with the partial result related to the main element, mainly to avoid null-based errors. Again as it happens in the *map* implementation, the *reduce* implementation uses the `StencilComputation` extended class methods to do the partial computations and the aggregation of these values.

Chapter 5

Experimental Results

In this chapter we present the evaluation of our decomposition solution, implemented on top of Hadoop MapReduce. We begin by presenting the methodology used for the evaluation, followed by the explanation of the application used to determine the performance of the decomposition strategy employed. Subsequently, we present the infrastructure used for the evaluation. And finally, we present and discuss the results obtained.

5.1 Methodology

Our evaluation is based on the assessment of the performance of our solution compared to the original Hadoop MapReduce framework. This aspect was evaluated using a synthetic application, presented in Section 5.2, that simulates the memory access pattern of a stencil application mentioned in in Sections 1.1.2 and 4.2.2. The implementation of such application makes the task of studying the impact of the developed solutions in that kind applications easier. Given that, to have a better control over our study we only have to vary some parameter to test different scenarios, either for the version implemented with our API, or the version developed with the original Hadoop framework. The input of both applications is identical. Both need to have information of the input matrix dimensions and the range of the stencil computation. The input of either application is composed by the matrix dimensions, range of the neighbourhood and the target cache level (TCL). The matrix parameter is always one of the three chosen matrix sizes, being these 1280x1024, 1920x1200 and 2000x2000. With regard to range of neighbourhood, we vary it with the values 1, 2 and 4. And finally the TCL, in which we specify one of the cache levels such as L1, L2 or L3.

In order to evaluate the performance of our decomposition in a speed-up analysis perspective, we measure the time spent in the *map* and *reduce* phases, but also the overall

time spent by the job execution of each application. To calculate the speed-up of our solution we average the time measures of 10 jobs, to obtain not only the overall application speed-up but also the respective speed-ups of the *map* and *phase*.

The evaluation is done with various inputs to better understand the results and to see where are the limits of our solution. Being this a work related to the memory hierarchy, it is worth mentioning that these inputs included a target cache level which specifies in which cache our decomposition will be based. Also, in order to work in a in-node context and to work only with the memory hierarchy, we had to use the GridGain In Memory Accelerator plug-in. This plug-in was integrated in our environment so that we could work in a shared memory environment and also, as mentioned in Section 3.2.1, this plug-in enables us to eliminate the intermediate calls, reads and writes to the disk, inherent to Hadoop MapReduce, which can cause great delays in the computation and are out of the context of this work.

5.2 Application

The MapReduce computation used in our experimental evaluation is essentially the one used as an example in Sections 1.1.2 and 4.2.2. The reason of the choice of using this stencil in our experimental evaluation is related with its behaviour that emulates the general idea of a stencil computation. With this stencil we can not only represent the usual stencil computation memory access, but also control the complexity of its computation by tweaking some parameters. As seen before, this is a stencil computation applied to a neighbourhood defined by a certain range r. In other words, all the elements within a range r from the main element contribute for the resulting value of that element. The computation itself is expressed by the sum of two distinct calculations. The first part is related to the main element own value and its contribution to the final result, wherein this contribution is given by the product of its value and a given percentage. The second part of the computation can be described as the sum of the products of the neighbour's values by a fraction of the remaining percentage. The next formula shows the computation calculation in a more succinct way:

$$M[i,j] = M[i,j] \times percentage + \sum_{ni=-r}^{r} \sum_{nj=-r}^{r} M[i+ni, j+nj] \times \frac{1-percentage}{(2r+1)^2-1}$$

In this formula the M[i,j] represents the main element value and the percentage is the related with the contribution of this value to the final result. The second part of the calculation has the sum of the products of the neighbours, where we divide the remaining of the percentage and calculate the contribution of each neighbour to the final result. It is worth to mention that we decided to use a percentage equal to 50% for the applications implementation.

The implementations of the stencil for the original MapReduce and for our API are naturally different. The main differences are in the input received in the *map* phase and also what actually can be computed in this phase.

The version implemented with the original MapReduce receives as input a complete line of the input matrix, wherein Section 3.1 its called line approach. This line is represented by the usual key/value pair, where the key is represented by the row number in the matrix and the value contains the bytes of n elements. The *map* phase of this approach is limited by its line input, it only calculates the main element part of the computation and also send its own value to its neighbourhood's reducers. With this, the *reduce* phase, has the task of completing the computations of each element with the values of the neighbours and the partial computation obtained from the main element's *mapper*.

On the other hand, the version implemented with our API receives a tile/block of the input matrix. The input of a *mapper* is composed by the block dimensions and the bytes of the actual block. In the key/value pair form, this is translated to a key that is represented by an object that contains the block dimensions and the value which is represented by the bytes of the data block. In the *map* phase of this approach, depending of the stencil range, we can obtain the final result of the majority of the elements. This is possible mainly because of the use o blocks instead of lines, but there are some scenarios where the *mapper* does not have all the necessary elements to finish the computation. For these scenarios we have two measures, consulting the cache (presented in Section 4.2) to try to get the missing elements or, if they are not available, the main element value is sent to its respective *reducer* to finish the computation as in the line approach.

In both cases there is some degree of temporal locality. However, the stencil itself features temporal locality because the input matrix elements are accessed multiple times in short period of time during the stencil execution. However, having the input matrix divided into smaller blocks that fit in cache can help avoiding memory data transfers and so resulting in reducing the computation time in a greater measure than using lines.

5.3 Test Infrastructure

The experimental study was performed on two machines running Apache Hadoop MapReduce 2.6.0 in a pseudo-distributed mode. To simulate a shared memory environment and in order to use a single Java Virtual Machine (JVM) we use the GridGain In-Memory Accelerator 6.6.4, as mentioned in the end of Section 3.2.1. The specifications of these machines follows:

- **System 1 (S1)** - 2 Quad-Core AMD Opteron™ Processor 2376 with three cache levels: a 64KBytes L1 data cache per core, a 64KBytes L1 instruction cache per core, a unified 512KBytes L2 cache per core, and a unified 6MBytes L3 cache per processor; and 8GBytes of RAM memory.

- **System 2 (S2)** - 2 Hexa-Core Intel Xeon™ Processor E5-2620 v2 with three cache levels: a 32KBytes L1 data cache per two cores, a 32KBytes L1 instruction cache per two cores, a unified 256KBytes L2 cache per two cores, and a unified 15MBytes L3 cache per four cores; and 64GBytes of RAM memory.

The choosing of these two systems can be explained by their differences in terms of cache memory architectures, but also by their differences in general at the number of processing units or even at the memory RAM capacity. However, they were chosen mainly because of the good representation of machines that are currently being used in some systems.

System 1 is powered by Debian with Linux kernel 2.6.26-2-amd64, while System 2 is powered by a Debian woth Linux kernel 3.16.0-4-amd64. Both these systems have installed the Java platform OpenJDK 7, but System 1 has the 1.7.0 71 version and System 2 has the 1.7.0 80 version.

The architecture of System 1 is a 4-core machine with 3 cache levels, where each CPU has its own cache in every level of the cache hierarchy. On the other hand, System 2 is a 6-core machine which has the same number of cache levels as System 1, but the L1 and L2 levels are shared by two CPUs and the L3 level is shared by four CPUs.

5.4 Experimental Results

In this section we present our experimental results. First we present the performance results which show the speed-up of both versions of the stencil targeting the different available memory levels of the cache memory hierarchy and using three different matrices, in order to better understand the effect of the input size and target cache level in our decomposition solution. In the following section, we present a breakdown study of the time and weight division of the applications, mainly to see where the applications are spending most of their execution time.

5.4.1 Performance Evaluation

The results presented in this section were obtained by running our test applications on S1 and S2. Following the methodology presented in Section 5.1, the application's configuration varies in three main parameters: the matrix dimensions, the range of the neighbourhood and also the target cache level (TCL). The Figures 5.1 to 5.6 present the speed-ups of our solution (in both S1 and S2) compared with the original Hadoop solution when varying the matrix dimensions, neighbourhood range and the TCL. The scale of ordinate axis of the charts is not the same for the sake of readability. In Tables 5.1 and 5.2 are presented the average of the times that each execution took when varying the mentioned parameters.

The general intuition leads us to believe that the closer a cache level is to the processing unit, the better it is as a target level for the decomposition solution, implying that it

Range	Matrix	L1	L2	L3	Original
1	1280x1024	17,93	17,79	16,92	19,96
	1920x1200	23,24	22,14	20,81	25,31
	2000x2000	28,26	25,21	24,22	31,78
2	1280x1024	22,34	20,40	17,66	27,25
	1920x1200	31,54	28,07	21,91	40,29
	2000x2000	50,71	37,97	28,11	56,72
4	1280x1024	49,17	36,46	21,76	52,93
	1920x1200	63,17	42,92	26,83	86,41
	2000x2000	135,59	140,14	127,42	144,43

Table 5.1: S1 performance results in seconds, with the last column being the original Hadoop version.

Range	Matrix	L1	L2	L3	Original
1	1280x1024	14,36	15,99	15,08	13,63
	1920x1200	15,96	17,36	15,99	15,81
	2000x2000	20,08	19,66	19,34	20,53
2	1280x1024	17,55	16,75	15,03	17,63
	1920x1200	24,10	19,42	17,17	21,38
	2000x2000	37,94	24,32	18,84	31,08
4	1280x1024	41,49	25,77	18,40	29,19
	1920x1200	68,97	35,33	21,79	46,14
	2000x2000	116,21	55,67	25,40	64,94

Table 5.2: S2 performance results in seconds, with the last column being the original Hadoop version.

benefits more from the locality concepts. However, the tendency in our results is to have better speed-ups the higher the level of the target cache is in the memory hierarchy. This can be explained by the relation between the cache size and the input size, meaning that if we divide our matrix according to smaller caches the result is more *map* tasks with less data. The number of smaller *map* tasks will not compensate the set up time of each task, as mentioned in [7]. The smaller the tasks the least computations can be done in the *map* phase, which implies that they have to be done in the sequential *reduce* phase. Besides, there is also the normal MapReduce data that will compete with the application data for space in the cache.

Also it can be perceived in these results that the input size is of great importance too. In either of the Figures from 5.1 to 5.6, it can be seen that the bigger matrices stencil computations get better performances as we go up in the cache hierarchy. With this, can be extracted a relation between the dimension of the input (or the input size) and the TCL, which indicates that computations with big inputs seem to avail from the use of higher cache levels. An exception can be seen in Figure 5.5, where the case of the 2000x2000 matrix with a range equal to 4 appears. This exception is a very particular case that shows the breaking point, or peak in our performance gains, which means that

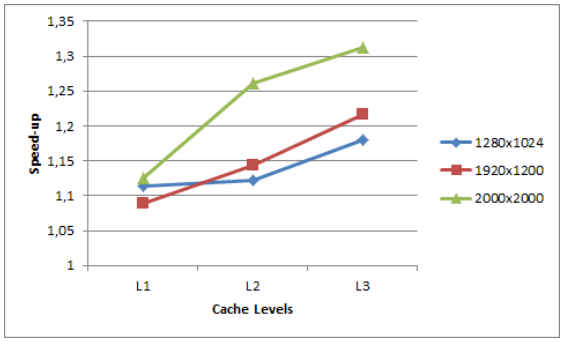

Figure 5.1: S1 speed-ups of the block oriented application (with range = 1).

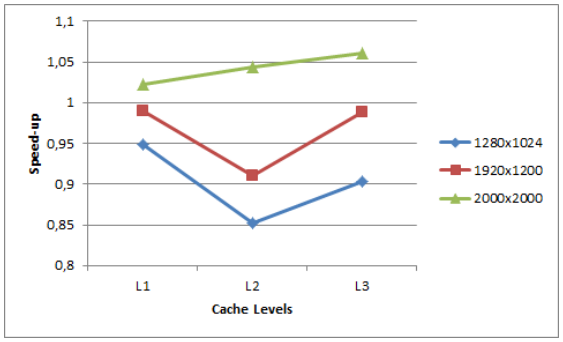

Figure 5.2: S2 speed-ups of the block oriented application (with range = 1).

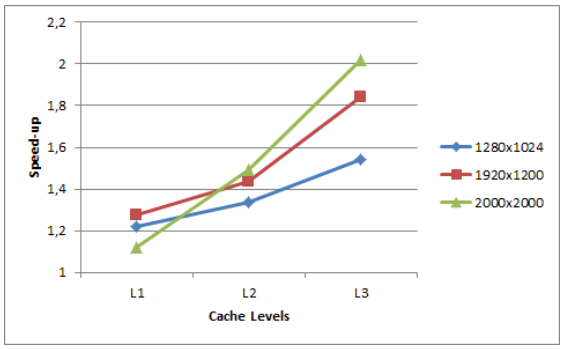

Figure 5.3: S1 speed-ups of the block oriented application (with range = 2).

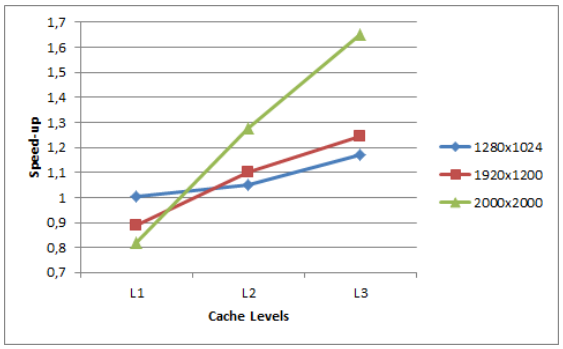

Figure 5.4: S2 speed-ups of the block oriented application (with range = 2).

the computation gets too expensive and the speed-ups drop in comparison with the other lighter configurations. In other words the bottleneck becomes the complexity of the computation itself. The same cannot be verified for the S2 because of its architecture which contains more processing units. Hence, we cannot see the peak or breaking point in S2 at the same configuration because of its greater processing power.

Something else that can be extrapolated from our charts is the influence of the range parameter on the performance of the computation. As can be seen in the figures, our decomposition solution benefits more from stencil computations with bigger neighbourhoods. By itself, this implies that the temporal locality concept is being used by the data blocks that contain more complete neighbourhoods, which means more completed computations in the *map* phase and, hence, less data transferred between the two main phases of MapReduce. This situation can be understood by thinking of the computation done in the *reduce* phase. For each element there is always a *reduce* task, but in the case of a complete element's computation in the *map* phase this *reduce* tasks is less costly, needing only to output the final result. If the opposite situation occurs, then the *reduce* task is more complex and has to iterate over the neighbourhood of the main element and compute the final result.

For the S1 machine, the majority of the configurations reached at least a speed-up of

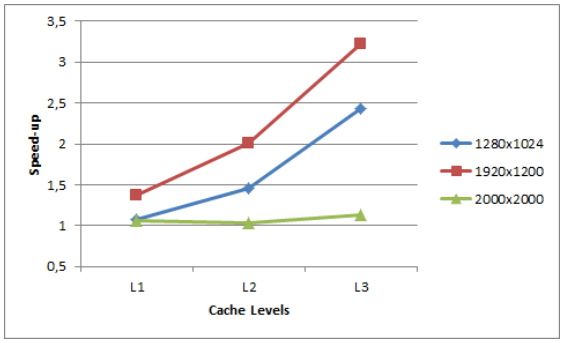

Figure 5.5: S1 speed-ups of the block oriented application (with range = 4).

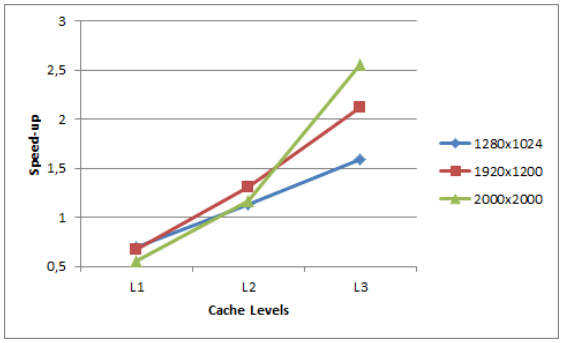

Figure 5.6: S2 speed-ups of the block oriented application (with range = 4).

1.1, or in other words, a improvement of 10%. With the best speed-up to be achieved by the stencil computation of the 1920x1200 matrix, with a range equal to 4 and using L3 cache as its target level. This configuration produced a 3.22 speed-up, as presented in Figure 5.5, which translates to an improvement of 222% of our approach over the approach that uses the regular Hadoop. The worst result comes from the stencil computation with a 2000x2000 matrix, also with a range equal to 4 and using L2 cache as its target level. This configuration only reached a 1.03 speed-up, which means a 3% improvement.

Regarding the S2 machine, we get a completely different scenario. The best speed-up is around 2.5 and its obtained targeting the L3 cache and running our stencil using a 2000x2000 matrix with a range equal to 4. The worst situation is described also by the stencil using a 2000x2000 matrix, but targeting the L1 cache with a range equal to 2. However, this system gets worst results than S1, especially when targeting cache L1. When L1 is the TCL, the API stencil version rarely have some speed-up and when it has does not goes beyond the 1.05 speed-up. This can be the result of the small size of the S2 L1 cache which only has 32KBytes, this meaning that this cache size can be our limit when trying to get some improvements in performance. Another reason that can help understanding the results of S2 is the fact that its cache levels are always shared at least between two different cores. This results in less space for each of the cores, which means that our implemented decomposition strategy did not had into account this detail, making the cores vied for the cache space. Given that, we only needed to add a simple mechanism that would verify if a cache was shared, because this information can be consulted in the file that represents the cache hierarchy of a given machine.

5.4.2 Breakdown

In this section we present the breakdowns of the most relevant cases in our analysis, these being the average of all executions for S1 (which got the better results), the best configuration with the best speed-up and the configuration with the worst speed-up. The breakdowns were done mainly to understand how our solution influenced the execution phases of the Hadoop MapReduce model, and also to see which phases are responsible for

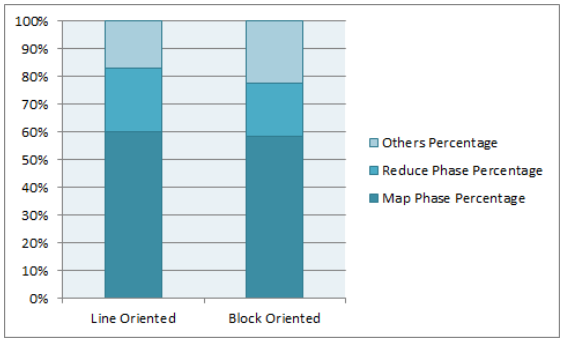

Figure 5.7: S1 average MapReduce phases weight for both approaches.

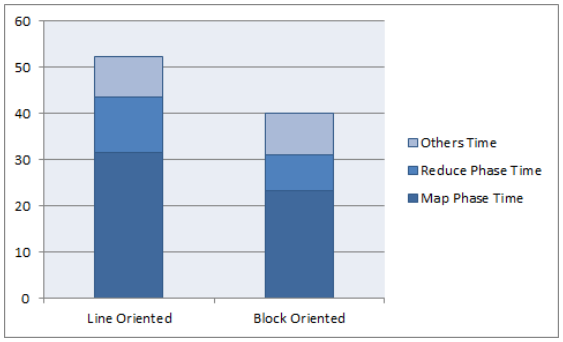

Figure 5.8: S1 average time division for both approaches.

the improvement and deterioration of the application's performance. These breakdowns are composed by two different categories: time division, where it can be seen which phase took the longer; and phase weight, where it can be seen which percentage of the execution each phase takes.

The case represented by the Figures 5.7 and 5.8, show us a view over the average situation for both the line oriented approach (implemented with the current Hadoop) and the block oriented approach (implemented with our API). In Figure 5.7, we can see that in average the percentages of the *map* and "others" phases do not change that much. However, our approach, the block oriented, reduces the percentage in both the *map* and *reduce* phases, but the remaining of the execution increases around 5%. This can be explained by the added complexity of the *split* phase in our approach. In regard to the time division point of view, Figure 5.8 shows that in average our solution takes 40 seconds to finish and the *line* approach takes around 52 seconds, which means a speed-up of 30%. It is also worth to mentioning that although the "others" phases in our solution take more percentage than its counterpart in the line-oriented approach, the difference between them is only around 1.2%.

The second case represents the best execution or the execution with the best speed-up, presenting its phase weight and time division charts in Figures 5.9 and 5.10, respectively. This result was obtained in S1, running our stencil with a range equal to four, targeting the L3 cache and using the 1920x1200 matrix. As can be seen in Figure 5.7, the percentages of the *map* and *reduce* phases decrease and consequentially the other operations percentage increase, as also verified for the average case. Yet, the increase of our solution other operations was of almost 23% relatively to the one of the line oriented approach. Again, this increase can be largely related to the added complexity of our approach to the *split* phase. Figure 5.10 give us the time division of the best run and shows that the line oriented approach took much more time that our approach, wherein the latter, as said in Section 5.4.1, reached a speed-up of around 222%. Although it is clear that proportionally our approach takes more percentage than the line oriented approach in the other operations, both take almost the same time with a difference of only around

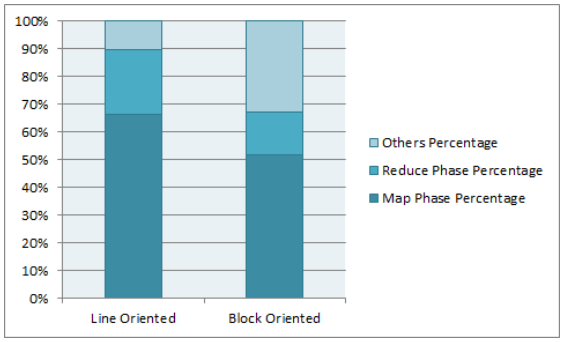

Figure 5.9: Phases weight for the best configuration for both approaches.

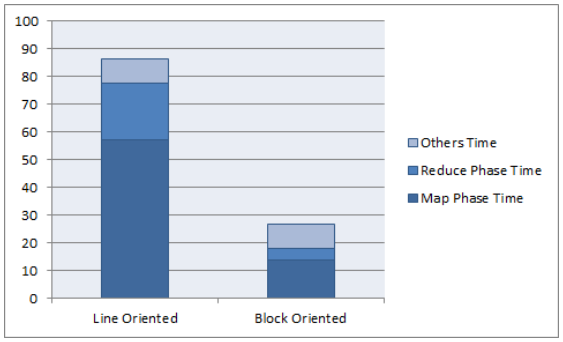

Figure 5.10: Time division for the best configuration for both approaches.

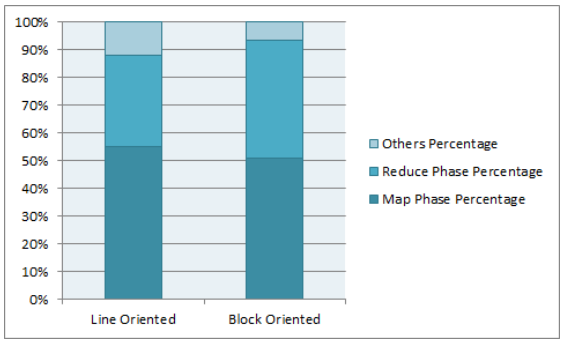

Figure 5.11: Phases weight for the worst configuration for both approaches.

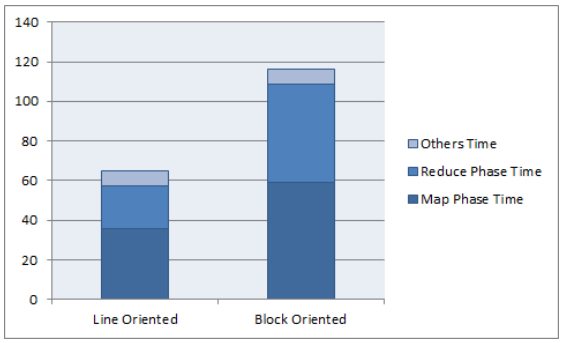

Figure 5.12: Time division for the worst configuration for both approaches.

0.02 seconds. From this case it can conclude that by performing more computations in the *map* phase (which is more cache-friendly) takes less time, Consequentially, this will decrease the amount of work done in the *reduce* phase, therefore we reduce the sequential component.

The third case is presented in Figures 5.11 and 5.12, an show the breakdown of the worst scenario. This case was the result of running our stencil in S2 with a range equal to four, targeting the L1 cache level and using as its input the 2000x2000 matrix. From this weight breakdown chart, we can observe that our solution increased the *reduce* phase percentage by around 9%. This situation can explain the bad result, mainly because the *reduce* phase is performed sequentially, which means that increasing the influence of this phase can damage the application performance. On the other hand, the remaining percentages were decreased, even though not in a relevant way in terms of helping the application's performance. From Figure 5.12, the time division, we can see that the line-oriented approach took almost half the time to perform the stencil. This results can be justified by the L1 caches size in S2, which are only 32KBytes and are also shared by two cores. Given that, it is probable that our solution produced many blocks that were too small, which mean longer *reduce* tasks because they have to do almost all the computations. However, with smaller blocks we get more blocks which leads to more

map tasks. In turn, the larger number of *map* tasks can then explain the increase of the time took by the *map* phase.

5.5 Discussion

The results presented, show that our approach provides better speed-ups than the current Hadoop based approach for the majority of the cases.

Even though we reached good speed-up results for our application with the majority of the configurations, our application is somewhat dependent of the configurations used. In other words, for a given machine the application's configuration can be different in order to achieve the same or better results. This could be verified when comparing the performance results obtained for S1 and S2. It can also be observed that by only changing the cache memory hierarchy file when changing machines, we obtained some reasonable speed-ups. Although S2 did not produce as good results as S1, we can say that our solution (with some adjustments) is fairly machine independent.

The proposed solution can adapt its decomposition to the machine with certain informations, such as the input size and target cache level, but these have to picked by the programmer. The same can be said about the applications configurations, in other words, for different applications/computations the best configurations can be different. Hence, this could be a flaw in our current implementation. To deal with this problem a mechanism could be added to choose the best fit for the target cache level and other parameters, even if this means having more information about the cache hierarchy and application.

Chapter 6

Conclusions

6.1 Final Conclusions

The contributions initially proposed for this dissertation were completely fulfilled. We introduce the concepts for the cache-friendly decomposition of an application's domain and how these could be integrated in the in-node MapReduce model. Some strategies were developed to integrate the support of concepts such as block-tiling decomposition and cache memory hierarchy information in this framework. The devised strategies enable the production of cache-friendly data blocks that can help improving the performance of applications that benefit from the temporal locality concept.

The prototyping of our solution was integrated in a concrete MapReduce framework. The choice of the framework fell on the popular Hadoop MapReduce. The original implementation of Hadoop MapReduce does not support any of our strategies. However, due to the fact that this framework is easily extensible we were able to use the available classes as the base for the development of new classes that integrated our decomposition strategies. Also, the Hadoop framework makes the application parallelization as seamless as possible, which facilitated some of our tasks. Given that, we extended this framework with an API for expressing the application's data decomposition into blocks that are cache-friendly.

In order to validate our solution and subsequent prototype implementation, we applied it to the context of stencil computations. The fact that this type of computations are quite important in the field of scientific computing, benefit from temporal locality and generally are embarrassingly parallel, made them the perfect candidate for our purpose. Given that, we developed a stencil computations API prototype for the Hadoop MapReduce. This API was developed to be as extensible as possible and easy to use. We think that these objectives were also achieved. To support this statement we have counted the

number lines of code of the implemented examples, which include our simple stencil, the SOR stencil and the Jacobi method. The number of lines of code required to express each of these computations were, respectively 101 lines for the simple stencil, 127 lines for SOR and 190 lines of code for the Jacobi method, each divided by the three different classes of our API.

To assess the performance of our solution and prototype, we performed a experimental evaluation based on the comparison between a stencil computation implemented with the original Hadoop and with our API. The results obtained from this evaluation revealed that our solution can achieve speed-ups that up to 222% and with an average around 77%, which for a solution based on optimizations is a reasonable result. Furthermore, these results were obtained for two machines with two completely different memory hierarchies, which shows that our prototype has some performance portability.

The obtained results show that our prototypes, both the decomposition and the stencil computations API, can be considered good contributions to the study of the cache-concious decompositions strategies (based on block-tiling) influence on the application's performance. Also, to the best of our knowledge, there is not any work that explores the discussed concepts in the way that we did.

6.2 Future Work

The developed work served as the first attempt to combine the concepts of block-tiling decomposition and cache memory awareness, and integrate them in the Hadoop MapReduce framework. Hence, there is a lot of work that can succeed and be based on ours, mainly in the improvement and expansion of the prototype. The developed prototype only scratches the surface of applications that benefit from temporal locality. From this, the development of support for other type of computations other than stencils, such as the ones based on matrix operations, could be a useful contribution not only for the added support itself, but also in the study of how different kinds of computations respond to the developed approach.

In terms of the distributed component of the MapReduce model, the study of how the implemented strategies influence the performance cluster of heterogeneous machines is something that could be very useful. Something that arises from this study is the necessity of a mechanism that can adjudicate which are the best configurations for the execution of an application in a given machine, as discussed in Section 5.5.

Bibliography

[1] B. Alpern, L. Carter, and J. Ferrante. "Modeling parallel computers as memory hierarchies". In: *Programming Models for Massively Parallel Computers, 1993. Proceedings*. IEEE. 1993, pp. 116–123.

[2] Apache Inc. *Apache Hadoop*. URL: http://hadoop.apache.org/.

[3] Apache Inc. *Apache Hive*. URL: https://hive.apache.org/.

[4] Apache Inc. *Apache Pig*. URL: http://pig.apache.org/.

[5] Apache Inc. *Apache Spark*. URL: http://spark.apache.org/.

[6] Apache Inc. *Hadoop wiki*. URL: http://wiki.apache.org/hadoop/PoweredBy.

[7] Apache Inc. *Partitioning your job into maps and reduces*. URL: https://wiki.apache.org/hadoop/HowManyMapsAndReduces.

[8] D. Borthakur. "The hadoop distributed file system: Architecture and design". In: *Hadoop Project Website* 11 (2007), p. 21.

[9] B. Calder, C. Krintz, S. John, and T. M. Austin. "Cache-Conscious Data Placement". In: *ASPLOS-VIII Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, USA, October 3-7, 1998*. 1998, pp. 139–149. DOI: 10.1145/291069.291036. URL: http://doi.acm.org/10.1145/291069.291036.

[10] S. Carr, K. S. McKinley, and C. Tseng. "Compiler Optimizations for Improving Data Locality". In: *ASPLOS-VI Proceedings - Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, USA, October 4-7, 1994*. 1994, pp. 252–262. DOI: 10.1145/195473.195557. URL: http://doi.acm.org/10.1145/195473.195557.

[11] R. Chen and H. Chen. "Tiled-MapReduce: Efficient and Flexible MapReduce Processing on Multicore with Tiling". In: *TACO* 10.1 (2013), p. 3. DOI: 10.1145/2445572.2445575. URL: http://doi.acm.org/10.1145/2445572.2445575.

[12] T. M. Chilimbi, M. D. Hill, and J. R. Larus. "Cache-Conscious Structure Layout". In: *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, May 1-4, 1999*. 1999, pp. 1–12. DOI: 10.1145/301618.301633. URL: http://doi.acm.org/10.1145/301618.301633.

[13] S. Coleman and K. S. McKinley. "Tile Size Selection Using Cache Organization and Data Layout". In: *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI), La Jolla, California, USA, June 18-21, 1995*. 1995, pp. 279–290. DOI: 10.1145/207110.207162. URL: http://doi.acm.org/10.1145/207110.207162.

[14] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. A. Yelick. "Optimization and Performance Modeling of Stencil Computations on Modern Microprocessors". In: *SIAM Review* 51.1 (2009), pp. 129–159. DOI: 10.1137/070693199. URL: http://dx.doi.org/10.1137/070693199.

[15] DBM2. *Known applications of MapReduce*. 2008. URL: http://www.dbms2.com/2008/08/26/known-applications-of-mapreduce/.

[16] J. Dean and S. Ghemawat. "MapReduce: simplified data processing on large clusters". In: *Commun. ACM* 51.1 (2008), pp. 107–113. DOI: 10.1145/1327452.1327492. URL: http://doi.acm.org/10.1145/1327452.1327492.

[17] E. D. Demaine. "Cache-oblivious algorithms and data structures". In: *Lecture Notes from the EEF Summer School on Massive Data Sets* 8.4 (2002), pp. 1–249.

[18] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. "Sequoia: programming the memory hierarchy". In: *Proceedings of the ACM/IEEE SC2006 Conference on High Performance Networking and Computing, November 11-17, 2006, Tampa, FL, USA*. 2006, p. 83. DOI: 10.1145/1188455.1188543. URL: http://doi.acm.org/10.1145/1188455.1188543.

[19] T. Ferreira, A. Espinosa, J. C. Moure, and P. Hernández. "An Optimization for MapReduce Frameworks in Multi-core Architectures". In: *Proceedings of the International Conference on Computational Science, ICCS 2013, Barcelona, Spain, 5-7 June, 2013*. 2013, pp. 2587–2590. DOI: 10.1016/j.procs.2013.05.446. URL: http://dx.doi.org/10.1016/j.procs.2013.05.446.

[20] B. B. Fraguela, J. Guo, G. Bikshandi, M. J. Garzaran, G. Almasi, J. Moreira, and D. Padua. "The hierarchically tiled arrays programming approach". In: *Proceedings of the 7th workshop on Workshop on languages, compilers, and run-time support for scalable systems*. ACM. 2004, pp. 1–12.

[21] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. "Cache-Oblivious Algorithms". In: *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*. 1999, pp. 285–298. DOI: 10.1109/SFFCS.1999.814600. URL: http://dx.doi.org/10.1109/SFFCS.1999.814600.

[22] S. Ghemawat, H. Gobioff, and S. Leung. "The Google file system". In: *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*. 2003, pp. 29–43. DOI: 10.1145/945445.945450. URL: http://doi.acm.org/10.1145/945445.945450.

[23] Indiana University. *Twister Iterative MapReduce*. URL: http://www.iterativemapreduce.org/.

[24] M. Kowarschik and C. Weiß. "An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms". In: *Algorithms for Memory Hierarchies, Advanced Lectures [Dagstuhl Research Seminar, March 10-14, 2002]*. 2002, pp. 213–232. DOI: 10.1007/3-540-36574-5_10. URL: http://dx.doi.org/10.1007/3-540-36574-5_10.

[25] Y. Mao, R. Morris, and M. F. Kaashoek. "Optimizing MapReduce for multicore architectures". In: *Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Tech. Rep*. Citeseer. 2010.

[26] H. Paulino and N. Delgado. "Cache-Conscious Run-time Decomposition of Data Parallel Computations". In: *CoRR* abs/1511.05778 (2015). URL: http://arxiv.org/abs/1511.05778.

[27] H. Paulino and E. Marques. "Heterogeneous programming with Single Operation Multiple Data". In: *J. Comput. Syst. Sci.* 81.1 (2015), pp. 16–37.

[28] E. Petrank and D. Rawitz. "The Hardness of Cache Conscious Data Placement". In: *Nord. J. Comput.* 12.3 (2005), pp. 275–307.

[29] A. Platzer. *Lecture Notes on Loop Transformations for Cache Optimization 15-411: Compiler Design*.

[30] S. M. F. Rahman, Q. Yi, and A. Qasem. "Understanding Stencil Code Performance on Multicore Architectures". In: *Proceedings of the 8th ACM International Conference on Computing Frontiers*. CF '11. Ischia, Italy: ACM, 2011, 30:1–30:10. ISBN: 978-1-4503-0698-0. DOI: 10.1145/2016604.2016641. URL: http://doi.acm.org/10.1145/2016604.2016641.

[31] C. Ranger, R. Raghuraman, A. Penmetsa, G. R. Bradski, and C. Kozyrakis. "Evaluating MapReduce for Multi-core and Multiprocessor Systems". In: *13st International Conference on High-Performance Computer Architecture (HPCA-13 2007), 10-14 February 2007, Phoenix, Arizona, USA*. 2007, pp. 13–24. DOI: 10.1109/HPCA.2007.346181. URL: http://dx.doi.org/10.1109/HPCA.2007.346181.

[32] J. Saramago, D. Mourão, and H. Paulino. "Towards an Adaptable Middleware for Parallel Computing in Heterogeneous Environments". In: *2012 IEEE International Conference on Cluster Computing Workshops, CLUSTER Workshops 2012*. IEEE, 2012, pp. 143–151.

[33] A. Shinnar, D. Cunningham, B. Herta, and V. A. Saraswat. "M3R: Increased performance for in-memory Hadoop jobs". In: *PVLDB* 5.12 (2012), pp. 1736–1747. URL: http://vldb.org/pvldb/vol5/p1736_avrahamshinnar_vldb2012.pdf.

[34] J. Talbot, R. M. Yoo, and C. Kozyrakis. "Phoenix++: modular MapReduce for shared-memory systems". In: *Proceedings of the second international workshop on MapReduce and its applications*. ACM. 2011, pp. 9–16.

[35] S. Treichler, M. Bauer, and A. Aiken. "Language support for dynamic, hierarchical data partitioning". In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*. 2013, pp. 495–514. DOI: 10.1145/2509136.2509545. URL: http://doi.acm.org/10.1145/2509136.2509545.

[36] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. "Apache Hadoop YARN: yet another resource negotiator". In: *ACM Symposium on Cloud Computing, SOCC '13, Santa Clara, CA, USA, October 1-3, 2013*. 2013, p. 5. DOI: 10.1145/2523616.2523633. URL: http://doi.acm.org/10.1145/2523616.2523633.

[37] Yahoo Inc. *Apache Hadoop Module 4: MapReduce*. URL: https://developer.yahoo.com/hadoop/tutorial/module4.html.

[38] Y. Yan, J. Zhao, Y. Guo, and V. Sarkar. "Hierarchical Place Trees: A Portable Abstraction for Task Parallelism and Data Movement". In: *Languages and Compilers for Parallel Computing, 22nd International Workshop, LCPC 2009, Newark, DE, USA, October 8-10, 2009, Revised Selected Papers*. 2009, pp. 172–187. DOI: 10.1007/978-3-642-13374-9_12. URL: http://dx.doi.org/10.1007/978-3-642-13374-9_12.

[39] Y. Zhang. "HJ-Hadoop: an optimized mapreduce runtime for multi-core systems". In: *Conference on Systems, Programming, and Applications: Software for Humanity, SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013 - Companion Volume*. 2013, pp. 111–112. DOI: 10.1145/2508075.2514875. URL: http://doi.acm.org/10.1145/2508075.2514875.

Appendix A

Stencil API Applications

A.1 SOR Stencil

```
1  public class SORMapper extends MatrixStencilMapper<Float, Float,
2   SORComputation> {
3
4    @Override
5    protected void stencilMapLogic(Context context)
6        throws IOException, InterruptedException {
7
8      boolean maintain = false;
9      for(int i = getFirstLineIdx(); i <= getFirstLastIdx(); i++)
10       for(int j = getFirstColumnIdx; j <= getLastColumnIdx; j++){
11         maintain = maintainValue(i, j);
12         compute(new Tuple<Integer, Integer>(i,j), maintain);
13       }
14     }
15   }
16
17   private boolean maintainValue(int i, int j){
18     return ((i==0 &&(j>=0 && j<= numberOfColumns-1)) ||
19         (j==0 && (i>=0 && i<= numberOfLines-1)) ||
20         (i==numberOfLines-1 &&(j>=0 && j<= numberOfColumns-1)) ||
21         (j==numberOfColumns-1 && (i>=0 && i<= numberOfLines-1)));
22   }
23
24   @Override
25   protected Float getMatrixInitialValue() {
26     return Float.NaN;
27   }
28
29   @Override
```

```
30    protected boolean isInitialValue(Float value){
31      return Float.isNaN(value);
32    }
33  }
```

Listing A.1: SORMapper class.

```
1   public class SORComputation extends StencilComputation<Tuple<Integer,Integer>,
2    Float[][], Float> {
3
4     private float omega_over_four;
5     private float one_minus_omega;
6
7     public SORComputation(){
8       this(1.5f);
9     }
10
11    public SORComputation(float omega) {
12      this.omega_over_four = omega * 0.25f;
13      this.one_minus_omega = 1.0f - omega;
14    }
15    @Override
16    public StencilComputation<Tuple<Integer, Integer>, Float[][], Float>
17         newInstance() {
18      return new SORComputation(1.5f);
19    }
20
21    @Override
22    public Float mainElementComputation(
23        Tuple<Integer, Integer> mainElementCoord, Float mainElementValue) {
24
25      return one_minus_omega * mainElementValue;
26    }
27
28    @Override
29    public Float neighbourComputation(
30        Tuple<Integer, Integer> mainElementCoord, Float mainElementValue,
31        Tuple<Integer, Integer> neighborCoord, Float neighborValue,
32        int numberOfNeighbors) {
33      int mainElementX = mainElementCoord.key;
34      int mainElementY = mainElementCoord.value;
35      int neighborX = neighborCoord.key;
36      int neighborY = neighborCoord.value;
37
38      if((((neighborX == mainElementX+1) || (neighborX == mainElementX-1))&&
39       (neighborY==mainElementY))||((neighborX == mainElementX)&&
40       ((neighborY == mainElementY+1) || (neighborY == mainElementY-1)))){
41        return omega_over_four * neighborValue;
42      }
43      else
```

```
44        return 0.0f;
45    }
46
47    @Override
48    public Float aggregateComputations(Tuple<Integer, Integer> mainCoords,
49     Float finalResult, Tuple<Integer, Integer> neighborCoords,
50      Float partialComputation) {
51
52      return finalResult+partialComputation;
53    }
54
55    @Override
56    protected Float readElement(ByteBuffer inputBuffer) {
57      return inputBuffer.getFloat();
58    }
59
60    @Override
61    protected byte[] writeElement(Float element) {
62      ByteBuffer buffer = ByteBuffer.wrap(new byte[Float.SIZE/8]);
63      buffer.putFloat(element);
64      return buffer.array();
65    }
66  }
```

Listing A.2: SORComputation class.

```
1   public class SOR extends MatrixStencilMR<Float, Float> {
2
3     public SOR(int numberOfLines, int numberOfColumns, long inputSize,
4      Configuration othersConfigs, String pathConfig)
5       throws IOException {
6       super (numberOfLines, numberOfColumns, inputSize,
7        othersConfigs, SORMapper.class, SORComputation.class, pathConfig);
8     }
9
10    public static void main(String args[]) throws IOException, ParseException {
11
12      if(args.length < 5)
13      {
14        System.out.println("[nLines] [nColumns] [input size] [input dir]
15         [output dir]");
16        System.exit(0);
17      }
18
19      //For Optional Configs
20      Configuration conf = new Configuration();
21
22      SOR sor = new SOR(Integer.parseInt(args[0]), Integer.parseInt(args[1]),
23          Long.parseLong(args[2]), conf, args[5]);
24
```

```
25      sor.launch("sor", args[3], args[4]);
26
27    }
28  }
```

Listing A.3: SOR class.

A.2 Jacobi Method Stencil

```
public class JacobiMapper extends MatrixStencilMapper<Float,
   Float, JacobiComputation> {

  @Override
  protected Float getMatrixInitialValue() {
    return Float.NaN;
  }

  @Override
  protected boolean isInitialValue(Float value){
    return Float.isNaN(value);
  }

  @Override
  protected void stencilMapLogic(
      Context context)
      throws IOException, InterruptedException {

    for(int i = getFirstLineIdx(); i <= getFirstLastIdx(); i++)
      for(int j = getFirstColumnIdx; j <= getLastColumnIdx; j++)
        compute(new Tuple<Integer, Integer>(i,j), false);
  }

}
```

Listing A.4: JacobiMapper class.

```
public class JacobiComputation extends StencilComputation<Tuple<Integer,
 Integer>, Float[][], Float> {
  private ArrayList<Float> resultsList;
  private int numberOfColumns;

  public JacobiComputation() {}
  @Override
  public void setConfiguration(Configuration configuration) {
    super.setConfiguration(configuration);
    ArrayList<String> list =
          new ArrayList<String>(Arrays.asList(
          this.config.getStrings("prev.values")));

    this.resultsList = new ArrayList<Float>();
    for(String value:list)
      this.resultsList.add(Float.parseFloat(value));

    this.numberOfColumns = (int) this.config.getLong("number.columns", 1);
  }
  @Override
  public StencilComputation<Tuple<Integer, Integer>, Float[][], Float>
```

```
22      newInstance() {
23      return new JacobiComputation();
24    }
25    @Override
26    public Float mainElementComputation(
27        Tuple<Integer, Integer> mainElementCoord,  Float mainElementValue) {
28
29      if(mainElementCoord.key == mainElementCoord.value ||
30         mainElementCoord.value == this.numberOfColumns-1)
31        return mainElementValue;
32      return mainElementValue * this.resultsList.get(mainElementCoord.value);
33    }
34    @Override
35    public Float neighbourComputation(Tuple<Integer, Integer> mainElementCoord,
36        Float mainElementValue, Tuple<Integer, Integer> neighborCoord,
37        Float neighborValue, int numberOfNeighbors) {
38
39      return mainElementComputation(neighborCoord, neighborValue);
40    }
41    @Override
42    public Float aggregateComputations(Tuple<Integer, Integer> mainElementCoord,
43        Float finalResult, Tuple<Integer, Integer> neighborCoords,
44         Float partialComputation) {
45
46      if((mainElementCoord.value == this.numberOfColumns-1) &&
47          (neighborCoords.key != neighborCoords.value) &&
48          (neighborCoords.key == mainElementCoord.key))
49        return finalResult - partialComputation;
50
51      return finalResult;
52    }
53    @Override
54    protected Float readElement(ByteBuffer inputBuffer) {
55      return inputBuffer.getFloat();
56    }
57    @Override
58    protected byte[] writeElement(Float element) {
59      ByteBuffer buffer = ByteBuffer.wrap(new byte[Float.SIZE/8]);
60      buffer.putFloat(element);
61
62      return buffer.array();
63    }
64  }
```

Listing A.5: JacobiComputation class.

```
1  public class Jacobi extends MatrixStencilMR<Float, Float> {
2
3    public Jacobi(int numberOfLines, int numberOfColumns, long inputSize,
4       Configuration othersConfigs, String pathConfig) throws IOException {
5      super (numberOfLines, numberOfColumns, inputSize, othersConfigs,
6       JacobiMapper.class, JacobiComputation.class, pathConfig);
7    }
8
9    public static void main(String args[]) throws IOException, ParseException{
10
11     if(args.length < 6)
12     {
13       System.out.println("[nLines] [nColumns] [input size] [input dir]
14        [output dir]");
15       System.exit(0);
16     }
17     //For Optional Configs
18     Configuration conf = new Configuration();
19     float[] diagonal= {10.0f, 11.0f, 10.0f, 8.0f};
20
21     String[] Prev = {"0.0", "0.0", "0.0", "0.0"};
22     conf.setStrings("prev.values", Prev);
23
24     int n = 5;
25
26     Jacobi brs = new Jacobi(
27         Integer.parseInt(args[0]), Integer.parseInt(args[1]),
28         Long.parseLong(args[2]), conf, args[5]);
29
30
31     conf = brs.launch("jacobi", args[3], args[4]);
32
33     String outputFilePath = args[4]+"/part-r-00000";
34
35     for(int k = 1; k <= n; k++)
36     {
37       Prev = readOutput(conf, outputFilePath);
38       if(Prev != null)
39       {
40         System.out.println("OUTPUT READ");
41
42         for(int i = 0; i < Prev.length; i++)
43         {
44           Prev[i] = ""+(1.0f/diagonal[i]* Float.parseFloat(Prev[i]));
45           System.out.println(Prev[i]);
46         }
47
48         if(k == n)
49           break;
```

```
50        conf.setStrings("prev.values", Prev);
51
52        FileSystem fs = FileSystem.get(conf);
53        fs.delete(new Path(args[4]), true);
54
55        brs = new Jacobi(
56            Integer.parseInt(args[0]), Integer.parseInt(args[1]),
57            Long.parseLong(args[2]), conf, args[5]);
58
59        conf = brs.launch("jacobi", args[3], args[4]);
60
61      }
62      else
63        System.out.println("NO FILE " + outputFilePath);
64    }
65  }
66
67  private static String[] readOutput(Configuration conf, String path)
68      throws IOException{
69    Path pt = new Path(path);
70
71    FileSystem fs = FileSystem.get(conf);
72    if(!fs.exists(pt))
73      return null;
74
75    int numberOfLines = (int)conf.getLong("number.lines", 1);
76    int numberOfColumns = (int)conf.getLong("number.columns", 1);
77
78    String regex = "\\[\\d+]\\["+(numberOfColumns-1)+"\\] (-?\\d+.\\d+)";
79    Pattern p = Pattern.compile(regex);
80    Matcher m;
81
82    BufferedReader reader=new BufferedReader(
83      new InputStreamReader(fs.open(pt)));
84    String line = "";
85
86    String[] prev = new String[numberOfLines];
87    int i = 0;
88    while (line != null) {
89      m = p.matcher(line);
90
91      if(m.matches()){
92        prev[i] = m.group(1);
93        i++;
94      }
95      line = reader.readLine();
96    }
97    for(int j = 0; j < prev.length; j++)
98      System.out.println(prev[j]);
99
```

```
100         return prev;
101     }
102 }
```

Listing A.6: Jacobi class. With some iterative attempt.