Nuno Miguel de Brito Delgado

BSc in Computer Science

A System's Approach to Cache Hierarchy-Aware Decomposition of Data-Parallel 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 António Maria Lobo César Alarcão Ravara

Arguentes: Prof. Doutor João Pedro Barreto

Vogais: Prof. Doutor Hervé Miguel Cordeiro Paulino

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Maio, 2014

A System's Approach to Cache Hierarchy-Aware Decomposition of Data-Parallel Computations

Dedico este trabalho à memória da minha avó Maria Celeste, que tomava conta de mim quando eu era pequeno. Dedico também este trabalho à memória do meu avô Alfredo, que fazia imensas engenhocas e assim me inspirou para ser engenheiro.

Agradecimentos

Em primeiro lugar, quero agradecer ao Prof. Hervé Paulino por toda a atenção disponibilizada e apoio prestado ao longo do período de desenvolvimento desta dissertação, não só a nível profissional como também pessoal.

Quero também agradecer ao Departamento de Informática da Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa, pelas excelentes condições de trabalho que me foram proporcionadas para o desenvolvimento deste trabalho, bem como por todas as oportunidades de envolvimento nas activades de investigação e apoio pedagógico do departamento, nas quais desenvolvi competências que me foram imensamente úteis ao longo do desenvolvimento desta dissertação.

Por último mas não menos importante, gostava de agradecer à minha família e a todos os meus amigos (alguns também colegas) por todo o apoio e contributo que me prestaram ao longo deste período.

Abstract

The architecture of nowadays' processors is very complex, comprising several computational cores and an intricate hierarchy of cache memories. The latter, in particular, differ considerably between the many processors currently available in the market, resulting in a wide variety of configurations. Application development is typically oblivious of this complexity and diversity, taking only into consideration the number of available execution cores. This oblivion prevents such applications from fully harnessing the computing power available in these architectures.

This problem has been recognized by the community, which has proposed languages and models to express and tune applications according to the underlying machine's hierarchy. These, however, lack the desired abstraction level, forcing the programmer to have deep knowledge of computer architecture and parallel programming, in order to ensure performance portability across a wide range of architectures.

Realizing these limitations, the goal of this thesis is to delegate these hierarchy-aware optimizations to the runtime system. Accordingly, the programmer's responsibilities are confined to the definition of procedures for decomposing an application's domain, into an arbitrary number of partitions. With this, the programmer has only to reason about the application's data representation and manipulation.

We prototyped our proposal on top of a Java parallel programming framework, and evaluated it from a performance perspective, against cache neglectful domain decompositions. The results demonstrate that our optimizations deliver significant speedups against decomposition strategies based solely on the number of execution cores, without requiring the programmer to reason about the machine's hardware. These facts allow us to conclude that it is possible to obtain performance gains by transferring hierarchy-aware optimizations concerns to the runtime system.

Keywords: Data-Parallelism, Hierarchical Parallelism, Domain Decomposition, Runtime Systems

Resumo

Ao longo dos últimos anos, o aumento do poder computacional dos CPUs tem sido alcançado através do aumento do número de cores e não através do aumento da frequência de relógio. Esta tendência levou à ascensão dos modelos multicore a modelo arquitectural predominante nos computadores de hoje em dia. Estes CPUs têm o potencial de aumentar a velocidade de programas que possam tirar partido de computação paralela. Além dos seus múltiplos cores, estes apresentam hierarquias de cache complexas, com diferentes configurações e afinidades aos cores existentes, abrindo assim as portas para optimizações cientes destas hierarquias.

Simultaneamente, as frameworks de programação estão incrementalmente a passar de modelos sequenciais para modelos paralelos, de modo a explorar na totalidade o potencial adormecido destas arquitecturas. A incorporação de paralelismo é feita através exposição explícita deste ao programador, ou através de transformações implícitas de código que introduzem paralelismo automaticamente.

A introdução de paralelismo por si só, contudo, não garante que o hardware está a ser utilizado no seu máximo. O mapeamento adequado de uma aplicação para a hierarquia de cache subjacente é crucial para explorar ao máximo o poder computacional destas arquitecturas. Os ganhos de desempenho derivam essencialmente da exploração tanto da localidade temporal como da espacial, no acesso aos dados. Contudo, a gestão das memórias cache é completamente transparente na programação de nível utilizador. Esta responsabilidade recai tipicamente sobre a infraestructura de hardware, cuja função é apenas garantir que dados acedidos recentemente estão mais próximos da unidade de computação do que os restantes, dado que provavelmente serão acedidos novamente.

Este trabalho procura contribuir para o estado da arte nesta área, mais concretamente, na programação eficiente de aplicações para processadores multicore. O escôpo do nosso trabalho irá incidir sobretudo em optimizações guiadas pela hierarquia de cache do hardware subjacente.

Palavras-chave: Paralelismo de Dados, Paralelismo Hierárquico, Decomposiçao do Domínio, Sistemas de Execução

Contents

List of Figures

Listings

1

Introduction

1.1 Motivation

Over the last years, the increase of CPU computational power has been sought through horizontal scaling of processing cores rather than the increase of clock frequency. This trend led to the rising of the multicore model as the de facto architectural model for today's computers. These CPUs have the potential to increase the overall speed of programs amenable to parallel computing. In addition to their multiple cores, these feature complex cache hierarchies with different configurations and affinities to the existing cores, opening the doors for hierarchy-aware optimizations.

Simultaneously, programming frameworks are incrementally moving from sequential to parallel ones, to fully exploit the dormant potential of these architectures. The incorporation of parallelism is carried forward either by explicitly exposing this parallelism to the programmer, or by performing implicit code transformations that automatically introduce parallelism.

The introduction of parallelism *per se*, however, does not guarantee that the hardware is being used to its fullest. Adequately mapping an application onto the underlying cache hierarchy is crucial to fully harness the computational power of these architectures. These performance gains derive essentially from the exploitation of both temporal and spatial cache locality in the access to data. However, the cache memory management is completely transparent to user-level programming. This responsibility typically falls upon the hardware infrastructure, whose function is only to guarantee that data accessed recently is closer to the computing unit than the remainder, since it will likely be accessed again.

This work seeks to contribute to the state-of-the-art in this area, more concretely, in

the efficient programming of applications for multicore processors. Our work scope will focus on optimizations driven by the cache hierarchy of the underlying hardware.

1.1.1 Motivational Example

Although we have presented an overview of the challenges and potential performance gains of hierarchical decomposition, an example can better illustrate the rationale behind these gains and how large these can theoretically be.

Let's consider a well-known linear algebra operation, the transposal of a matrix. The mathematical properties of the operation are not the focus in the context of this thesis, so we will rely solely on a Java implementation of the algorithm that performs the operation, presented in Listing 1, as the base for our discussion.

```
1  void transpose(int[][] A, int[][] T) {
2    for(int i=0; i < A.length; i++)
3      for(int j=0; j < A[0].length; j++)
4        T[i][j] = A[j][i];
5  }
```

Listing 1: Matrix Transposal Algorithm

It is worth noting that the arguments and reasoning afterwards presented assume that the lines of bi-dimensional arrays are stored in row-major order in memory. Row-major order describes methods for storing n-dimensional arrays in memory which store rows contiguously in memory, in contrast with column-major order, which stores columns (for bi-dimensional arrays) contiguously in memory rather than rows. To illustrate this concept, consider the following matrix:

$$M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

If M was stored in a row-major order, the elements of M would be laid out contiguously in memory as $\{1, 2, 3, 4, 5, 6\}$. On the other hand, if M was stored in a column-major order, its elements would be laid out as $\{1, 4, 2, 5, 3, 6\}$. Row-major order is prevalent in most programming languages, including Java.

We can observe that when the algorithm iterates matrix `A` across its lines, `T` is iterated across its columns. If `A` and `T` are square matrices with side $N = 4$, we will have the iteration pattern illustrated in Figure 1.1. The value in each block of each matrix represents its order number in the sequence of accesses to the matrix by the algorithm.

Since the granularity of data fetching from main memory into the cache is a cache line, which is a sequence of contiguous bytes, accessing contiguous memory positions sequentially exploits spatial cache locality and provides a better performance than non-sequential memory accesses. Therefore we can observe that, in the context of the matrix transposal algorithm, while the iteration of `A` promotes spatial cache locality, the iteration of `T` does not. When a element of `T` is accessed, the contiguous elements in the same cache

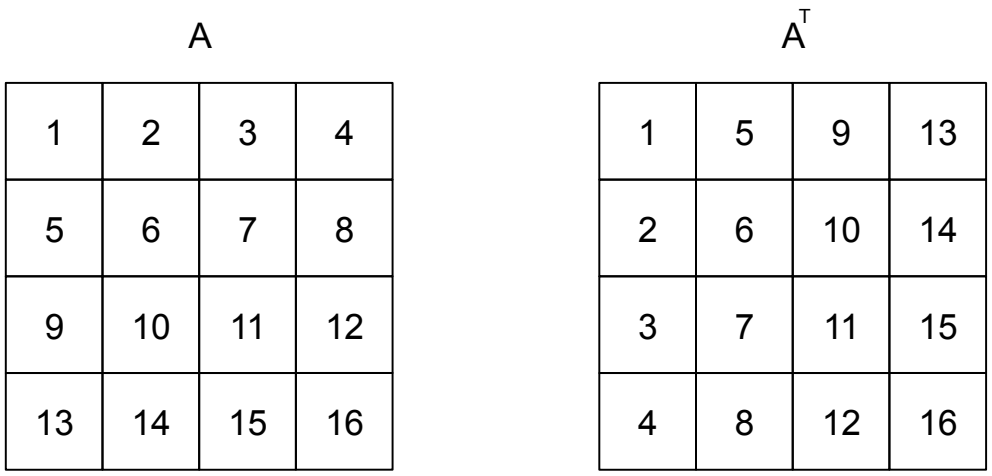

Figure 1.1: Matrix Transposal Iteration Pattern

line are also brought into the cache, but since `T` is iterated across its columns, if `T` is large enough, these elements will eventually be removed from the cache to make space for the elements accessed afterwards by the algorithm. This results in a "waste" of cache hits that could have resulted from sequential accesses across the lines of `T`.

Having each computation well mapped onto the underlying memory hierarchy can provide applications with significant performance gains. If one divides computations into tasks that execute with a reduced input workload fitting the smallest cache level, one can expect to fully exploit both temporal and spatial cache locality during the execution of each task.

Consider that all blocks of the matrices `A` and `T` have the same size in bytes. Now imagine a hypothetical architecture with a cache line size of 2 blocks, and a cache size of 8 blocks. If we divide the original transposal operation into 4 transposal operations that operate over different quadrants of the matrices, and a single CPU executes these operations starting from the top left quadrant of `A` and ending with the lower right quadrant, we will have the iteration pattern depicted in Figure 1.2.

This hierarchy-aware iteration pattern prevents the cache hit "waste" of the original iteration. When an element of `T` is accessed and the elements on the same cache line brought into the cache, these will still reside on the cache once the iteration of `T` moves to the next columns, where these elements reside. Considering the example of Figure 1.2, once the element 1 of `T` is accessed, 3 will also be fetched into the cache since it belongs to the same cache line. Even though the iteration of `T` will then access 2, it will afterwards access 3 and it will be a cache hit since `T` was already brought into the memory; this pattern will occur for every pair of quadrants of `A` and `T`. This results from the fact that the number of elements iterated by each transposal operation fits the cache, hence no elements will have to be removed from the cache during each operation, and therefore, spatial locality will be exploited during the execution.

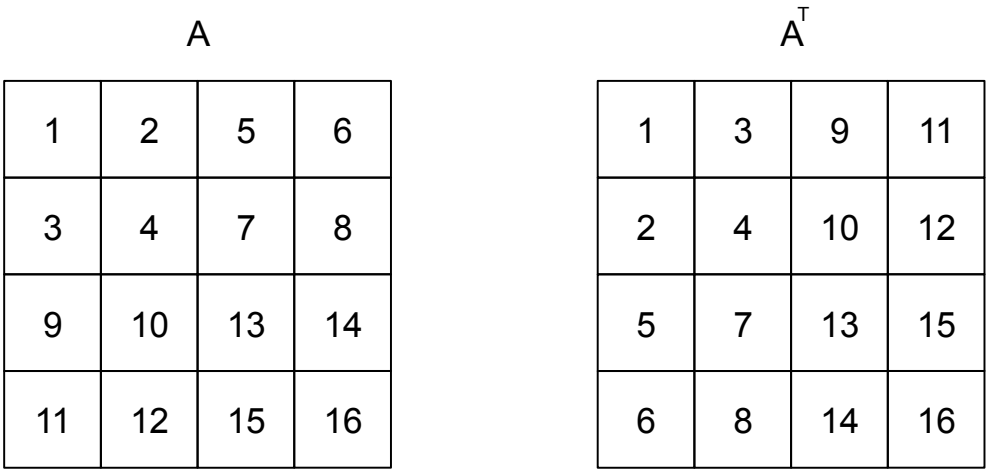

Figure 1.2: Matrix Transposal Iteration Pattern (revised)

1.2 Problem

When performing hierarchy-specific optimizations, there are several aspects that one may take into account and tackle:

- Cache sizes
- Cache layer sharing (dedicated or shared)
- Cache alignment
- Target cache level
- Sibling core couples
- Inclusive or exclusive data storing policy across layers

The greatest challenges in this context are how to perform these optimizations with the minimal intervention from the programmer, who should not be concerned with complex hierarchy details when developing his applications, and how to program applications in a way they can be mapped onto different hierarchy configurations using the same source code.

Several state-of-the-art approaches like Sequoia [FHK+06], Hierarchical Place Trees [YZGS10], Hierarchically Tiled Arrays [BGH+06] and [TBA13] provide machine abstractions for the programmer to abstractly represent the hierarchy of the target machine. Using these abstractions, the programmer can choose where each computation takes place and define the size of its input workload.

Despite providing an increased abstraction of the machine hierarchy, these languages place a heavy burden on the programmer when defining the machine model and the computation mappings and sizes, requiring previous knowledge from the former regarding the memory layers that exist on the machine.

In this thesis we propose a system-based approach to hierarchical parallelism where these optimization parameters are inferred and automatically tuned by the runtime system/compiler instead of being manually set by the programmer.

1.3 Proposed Solution

The popularity of Google's MapReduce parallel programming framework [DG08] led to the generalization of the MapReduce designation, which nowadays is more associated with a programming paradigm, than to Google's original programming model. This programming paradigm is characterized by a workflow of three main stages: `Split` decomposes the given input dataset into a system defined number of partitions, `Map` applies a given computation, in parallel, to each of such partitions, generating a set of results that may then be reduced by the last stage, *Reduce*, that produces the computation's final result.

In this work, we are particularly interested in the `Split` stage. Common domain decompositions split the input dataset into a predetermined number of partitions. Within a single computing node, this number is usually bound to the number of worker threads assigned to the operation (assume N to be the arity of such set). Ergo, the domain is partitioned as evenly as possible between these N workers, spawning that many tasks.

This approach is cache hierarchy-neglectful and hence, in many cases, does not harvest the benefits provided by the (consistently growing amount of) cache hardware available in current computers, from laptops to high-end server nodes. Our proposal overcomes this limitation by applying a vertical, cache-aware, decomposition strategy to the *Split* stage that takes in account cache sizes and CPU cores affinities. As a result, the number of partitions (that we generalize into working sets) will no longer be bound to the number of workers available to perform the computation, but instead be a function of the target machine's cache hierarchy. The *Split* operation is, itself, divided into two stages: the first performs the aforementioned vertical domain decomposition, while the second generates the tasks that will actually be executed by the workers assigned to the computation.

The starting point of this project will be Elina [SMP12], a middleware for parallel programming that supports shared and distributed memory systems in the same framework. Elina was developed with the purpose of being a modular platform, which led to the concept of pluggable adapter and the composition of these into complex adapter hierarchies. The support for both task and data parallelism requires the existence of dedicated adapters that provide concrete implementations of the pool(s) of workers and of domain decomposition. We propose implementing new adapters for Elina that automatically collect the underlying hardware platform specifications and decompose applications in accordance with these. Using this approach, the application's optimization process will be automatically performed.

The statement of this thesis is: it is possible to obtain performance gains in the execution of data-parallel computations by systematically decomposing the computation's domain, in such a way that it leverages the benefits of the cache hardware, without requiring additional attention from the programmer's part.

1.4 Contributions

Our concrete contributions with this dissertation are the following:

- A collection of theoretical concepts and problems involved in hierarchy-based domain decomposition. These concepts are complemented with algorithms and heuristics that attempt to solve these problems;
- A framework that allows programmers to have their applications automatically and efficiently mapped onto a target machine's memory hierarchy, with minimal intervention from the former;
- An evaluation that reveals the speedup of a vertical domain decomposition approach against a horizontal, core-based domain decomposition approach.

1.5 Document Organization

The remainder of this document is organized as follows: Chapter 2 presents state-of-the-art approaches to hierarchical parallelism, namely the models that represent these hierarchies and the mapping of an application onto the formers; Chapter 3 introduces data-size driven domain decomposition, presenting the involved challenges along with algorithms and heuristics that solve or approximate a solution for these; Chapter 4 builds upon these concepts to explain their transposal into the Elina framework, upon which these were implemented; Chapter 5 presents the experimental results of the performed evaluation of the work developed in this dissertation; finally, Chapter 6 closes this dissertation with our conclusions about the developed work, ending with our expectations about the possible future work in this topic.

2

State of the Art

This chapter presents the state-of-the-art on cache-hierarchy aware data parallelism, presenting the languages, models and decomposition mechanisms that were proposed to deal with the problems we are addressing in this thesis. Section 2.1 presents the hierarchical parallelism model that serves as the basis for the representation of systems' hierarchies. Section 2.2 presents existing languages and respective models to express task decomposition. Finally, Section 2.3 introduces hierarchical work distribution, namely the *Hierarchical Mapping Algorithm* (HMA).

2.1 Hierarchical Parallelism

The Parallel Memory Hierarchy (PMH) [ACF93] model is a computational model that builds on a single mechanism to model both the costs of interprocessor communication and memory hierarchy traffic into a tree of memory modules with processors at the leaves. Figure 2.1 depicts some examples of machines modeled using the PMH model.

There are two main types of data movement on a computer: horizontal interprocessor communication and vertical memory hierarchy traffic. PMH models both of these types of data movement within the same framework, with the following traits:

- each child is connected to its parents by a unique channel;
- modules hold data;
- leaf modules are the only that perform computation;
- data in a module is partitioned into *blocks* - the unit of transfer on the channel connecting a module to its parents;

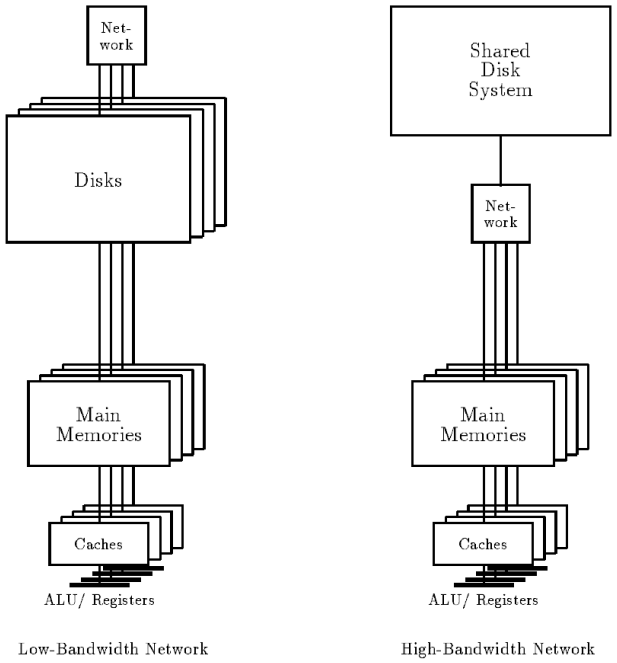

Figure 2.1: Memory Hierarchies examples, taken from [ACF93].

- all of the channels can be active at the same time, but two channels cannot move a given block at the same time.

Each module *m* in the model is parameterized through four different parameters: *blocksize sm* - number of bytes per block of *m*; *blockcount nm* - number of blocks that fit in *m*; *childcount cm* - number of children of *m*; *transfer time tm* - number of cycles per block transfer between *m* and its parents. When in the presence of a homogeneous system, all modules at a given level of the tree will be parametrized through the same parameters.

Modeling the Computer In order to model a computer, the model requires a tree structure that represents the machine's communication capabilities and memory hierarchy. One has to take in account the fact that this structure defines the data bandwidth between different architectural components (processors, memory, disks, communication controllers, etc.) of the computer.

A consequence of PMH's tree structure is that each channel is the only connection between two subgraphs in the model graph. Also if the channel has a low bandwidth, it is impossible to create a high-bandwidth data path between two components that do not belong to the same subgraph.

The most basic strategy for defining a graph *G* that represents the hierarchy tree is to represent the components as nodes in the graph, and the data paths that connect them as the graph's edges. The weight of each edge is given by the bandwidth of the corresponding data path.

After defining the graph *G*, the corresponding PMH model, *M*, can model can be obtained by applying the following rules:

1. For each processor node in G, create a leaf module in *M*
2. Compute the "threshold bandwidth" *b* as the maximum weight of any edge to a processor node
3. In *G* merge all pairs of nodes connected by edges of weight b or higher, and create a parent module (a new "root") in *M* to represent the subgraph of *G* that was merged with each processor module
4. Repeat the procedure until *G* is a single node

The resulting model *M* will have the sets of modules sorted from the root to the leafs by the maximum bandwidth to a processor node.

Modeling Latency The bandwidth of a channel is not the only factor that defines the time needed to move data from a module to another. The link between components is commonly modeled using both the *bandwidth B* and the *latency L* of the channel connecting these components, a model usually referred to as the *L-B* model.

The algorithm presented in the previous paragraph modeled the system having in account only the bandwidth of communication channels, hence it may be inaccurate when metrics such as latencies or disk access times are relevant to the system. A PMH model is unable to simultaneously represent a model based both on the channels bandwidth, and the channels communication delay.

Modeling Contention In some architectures, it is common that a single communication bandwidth *B1* exists between pairs of processors on a slightly loaded bus or network. The situation is slightly different when several processors are using the channel, reducing the effective bandwidth to a lower value *B2*.

When modeling these architectures, the problem can be solved by giving each module a bandwidth B2 with the parent module.

Modeling Real Computers When comparing a specific model against a "real computer" methodological difficulties may arise. As a workaround to this problem, PMH defines *performance model*, this kind of model is motivated by a real machine and represents an abstraction of the machine's performance characteristics. In [ACF93] two performance models are presented:

- CM-5: 4-ary Fat Tree with 1024 processing nodes
- KSR1: features a two-level hierarchy of rings, a single address space implemented on physically distributed memory, and high-performance RISC processors with large registers and local caches

2.2 Hierarchical Programming Models

These ideas and concepts presented by PMH drove the proposal of hierarchy-awareness on parallel applications, which can lead to significant performance gains, provided that applications optimize data movement and the computation sizes according to the underlying hierarchy.

In order to efficiently define parallel applications with these concerns in mind, it is mandatory to have language-level support. Over the last years, several languages, and respective runtime systems, have been proposed to allow programmers to abstractly define hierarchy-aware applications. The remainder of this section will present the most relevant state-of-the-art approaches.

2.2.1 Sequoia

The Sequoia [FHK+06, HPR+08] programming language was designed with the purpose of facilitating the development of memory hierarchy aware parallel programs. One of the main goals of this work is to provide the means to develop applications that are portable across machines with different memory hierarchies.

2.2.1.1 Sequoia Model

Writing Sequoia programs involves abstractly describing hierarchies of tasks, which are subsequently mapped onto the target machine's memory system. The programmer is required to reason about a parallel machine as a tree of distinct memory modules.

The basic program building block in Sequoia is the `task`, a side-effect-free function with call-by-value parameter passing semantics. Through tasks the programmer is able to express: parallelism, explicit communication, locality, isolation, algorithmic variants and parameterization. These properties allow programs written in Sequoia to be portable across machines, without sacrificing the possibility of tuning the application according to the hardware specifications of each target machine.

Sequoia introduces the `array blocking` and `task mapping` constructs. These are first-class primitives available to describe portable task decomposition. The `mappar` construct is a task mapping construct used to designate parallel iteration, being used during a task's execution to create subtasks that execute in parallel.

Tasks execute isolatedly without any synchronization between cooperating threads. This allows parallel tasks to be executed simultaneously using multiple execution units or sequentially on a single processor. Furthermore, parameter passing during the creation of subtasks is the only inter-task communication mechanism available in Sequoia. The isolation and the lack of other communication mechanisms increase the complexity of expressing cooperative computations. Nevertheless, Sequoia programs feature an increased code portability since tasks are programmed without any assumptions regarding the communication mechanisms available in the underlying platform.

More than one implementation (variant) may exist for a given task. To express such multiplicity; tasks are identified through the syntax `taskname::variant_name`. The rationale behind task variants is closely tied to the definition of recursive algorithms, the base case and the general case can be considered as variants of the same task. In this context, Sequoia identifies two task variants: *inner tasks* and *leaf tasks*. Inner tasks are responsible for spawning subtasks, leaf tasks on the other hand, do not spawn subtasks and operate directly upon workingsets residing within leaf levels of the memory hierarchy.

In the matrix multiplication example provided in the documentation, the base case does not need to be the smallest task unit. The decision to apply either the general or the base case pertains to the machine-specific mapping of the algorithm, which means that the system may stop dividing the problem when it sees fit to do so.

The programmer is required to define tasks in a parametrized form. Parametrization allows the decomposition strategy specified by a task variant to be applied in a variety of contexts, making the task portable across machines and across levels of the memory hierarchy within a single machine.

The programmer is additionally required to provide the compiler with the task mapping specification for the machine where the algorithm will be compiled and executed. The task mapping specification is maintained separately from the Sequoia source, and describes the mapping and tuning of the algorithm for the target machine. This approach places an additional burden over the programmer, requiring the latter to be responsible for mapping a task hierarchy onto the target machine. While an intelligent compiler may be capable of automatically employing this process, Sequoia's design empowers the performance-oriented programmer to manage the main aspects of the mapping phase in order to achieve maximum performance.

Task execution in Sequoia follows a Single Program Multiple Data (SPMD) parallel programming model, where different processes execute the same program over different data. Processes know their process ID and interact through collective operations supported by the runtime system.

Application Example: Matrix Multiplication

The multiplication of matrices `A` and `B`, to produce matrix `C`, can be divided into smaller computations that operate over different subsets of the original matrices. This promotes parallelism in order to achieve a better performance. In Sequoia, this matrix multiplication can be represented as a *matmul* task with two variants:

- **inner**: splits the input matrices into a set of blocks and creates parallel subtasks that compute the partial product of the blocks.
- **leaf**: performs the matrix multiplication algorithm over the input matrices.

An illustrative example of this task division can be seen in Figure 2.2. A possible implementation for these tasks is shown in Listing 2 and Listing 3 respectively.

```
1  void task matmul :: inner (in float A[M][P],in float B[P][N],
2  inout float C[M][N]){
3   // Tunable parameters specify the size of subblocks of A , B , and C.
4   tunable int U;
5   tunable int X;
6   tunable int V;

8   // Partition matrices into sets of blocks using regular 2D chopping .
9   blkset Ablks = rchop (A, U, X);
10  blkset Bblks = rchop (B, X, V);
11  blkset Cblks = rchop (C, U, V);

13  // Compute all blocks of C in parallel .
14  mappar ( int i=0 to M/U, int j=0 to N/V) {
15   mapreduce ( int k=0 to P/X) {
16     // Invoke the matmul task recursively on the subblocks of A , B , and C.
17     matmul ( Ablks [i][k], Bblks [k][j], Cblks [i][j ]);
18   }
19  }
20 }
```

Listing 2: matmul::inner task, taken from [FHK+06]

```
1  void task matmul :: leaf (in float A[M][P], in float B[P][N],
2  inout float C[M][N]) {
3   // Compute matrix product directly
4   for ( int i =0; i<M; i ++)
5    for ( int j =0; j<N; j ++)
6      for ( int k =0; k<P; k ++)
7        C[i][j] += A[i][k] * B[k][j];
8   }
```

Listing 3: matmul::leaf task, taken from [FHK+06]

The `matmul::inner` task variant performs a two-dimensional chopping over the input matrices. The programmer can use the *tunable* variables in order to control the amount of data that composes a matrix block, in order to adapt the application for a given memory hierarchy. The produced sets of blocks will be assigned to parallel subtasks in the `mappar` construct, which contains in its body a `mapreduce` construct that is necessary in order to reduce, through a `sum` operation, the results of subtasks upon which a given block of C depends.

The matrix multiplication operation is performed by the `matmul::leaf` variant, which should be choosen for execution when further task divisions would no longer provide better expected results in the the memory hierarchy.

Configuration In addition to programming the tasks required to carry on the computation, the programmer also has to define a configuration for the application, relative to the target memory hierarchy, in order to optimize its execution for the latter. Listing 4 presents a configuration to efficiently execute the matmul task in a cell processor.

The configuration file specifies that the instance executing at the cluster level is a matmul::inner variant, which creates subtasks of the `matmul_node_inst` instance type. Instances of `matmul_node_inst` type execute also execute a `matmul::inner variant`, and so does `matmul_L2_inst`. Only when the task hierarchy reaches `matmul_L1_inst` is the

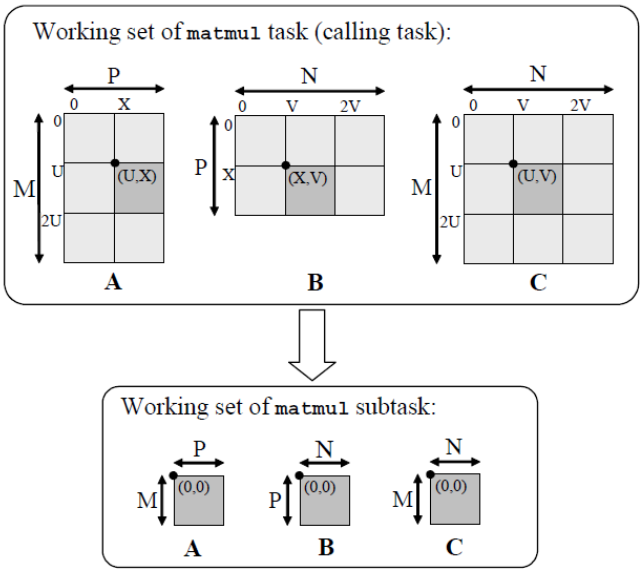

Figure 2.2: Matmul::inner decomposition into subtasks, taken from [FHK+06].

`matmul::leaf` variant executed.

The values assigned to tunable variables `U`, `X` and `V` are defined so that a matmul working set entirely fits the L1 cache level, which leads to a better performance since cache locality is maximally exploited during the execution of each task.

Irregular Parallelism In the previous examples, as well as the reasoning behind the Sequoia principles, parallelism is assumed to be regular. Parallelism is regular if and only if the two following conditions are met:

- **The working set of each subtask is known in advance**: All of the inputs of a task have to be known at the time it is invoked. Also, in order to guarantee that the invoked task has enough space to finish its computation, it is necessary to know at least an upper bound for the size of the task's result;
- **The number of subproblems in a task is known beforehand**: The number of parallel subtasks is fixed upon entry on the `mappar` construct. Control can only return to the parent task once all the subtasks finish executing.

Irregular parallelism may arise in two common situations:

- A task requires only a small portion of a large data set, therefore due to performance reasons the task should be given only the aforementioned portion.
- The output is so large compared to the input that the problem size a task can handle is limited by the size of the output (the whole output has to fit the memory level where the task is executing). Tasks could be able to off-load partial results to the parent and then proceed with their execution.

```
instance {
  name = matmul_cluster_inst
  variant = matmul :: inner
  runs_at = cluster_level
  calls = matmul_node_inst
  tunable U=1024 , X=1024 , V =1024
}
instance {
  name = matmul_node_inst
  variant = matmul :: inner
  runs_at = node_level
  calls = matmul_L2_inst
  tunable U=128 ,X=128 ,V =128
}
instance {
  name = matmul_L2_inst
  task = matmul :: inner
  runs_at = L2_cache_level
  calls = matmul_L1_inst
  tunable U=32 ,X=32 ,V =32
}
instance {
  name = matmul_L1_inst
  task = matmul :: leaf
  runs_at = L1_cache_level
}
```

Listing 4: Matrix multiplication task configuration for a Cluster, taken from [FHK+06].

Sequoia constructs that support irregular parallelism are proposed in [BCSA11], namely the *call-up* and *spawn* constructs. The `Call-up` construct provides subtasks with access to their parent task's heap, which can be used to modify data structures in the latter. Since a task typically launches multiple subtasks, the `call-up` construct introduces concurrency into the Sequoia programming model.

Spawn is a parallel construct that takes two arguments: a task call and a termination test. A `spawn` invocation may launch an arbitrary number of subtasks of the provided task call during its execution. The spawn construct terminates and moves computation into after its scope when two conditions are met: the termination test evaluates true, and all subtasks have finished executing.

2.2.2 Hierarchically Tiled Arrays

Hierarchically Tiled Arrays (HTA) [BGH+06] were proposed in 2006 as a new programming paradigm for expressing parallelism and locality. This new programming model relies on a new object type named *tiled array*. HTA programs are single-threaded programs where parallel computations are represented as array operations.

Semantics of Hierarchically Tiled Arrays Hierarchically Tiled Arrays (HTAs) are arrays partitioned into tiles, which can in turn be either conventional arrays or further tiled arrays. Figure 2.3 shows a recursively tiled HTA model. There are two different ways to create an HTA. The first approach consists on partitioning an existing array into tiles using delimiters for each dimension the array posesses. As an example lets consider a 6×6

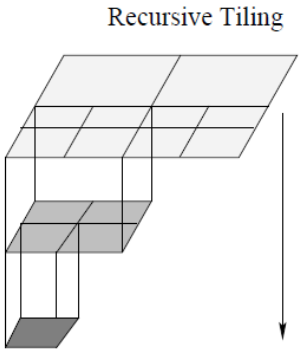

Figure 2.3: Pictorial view of an Hierarchically Tiled Array [BGH+06].

matrix M, whose contents may be posteriorly assigned `hta(M,[1 3 5],[1 3 5])` creates a 3×3 HTA that consists of tiles with 2×2 elements of M each. Figure 2.4-(a) illustrates this scenario along with the resulting tiles. An HTA may also be created as an empty

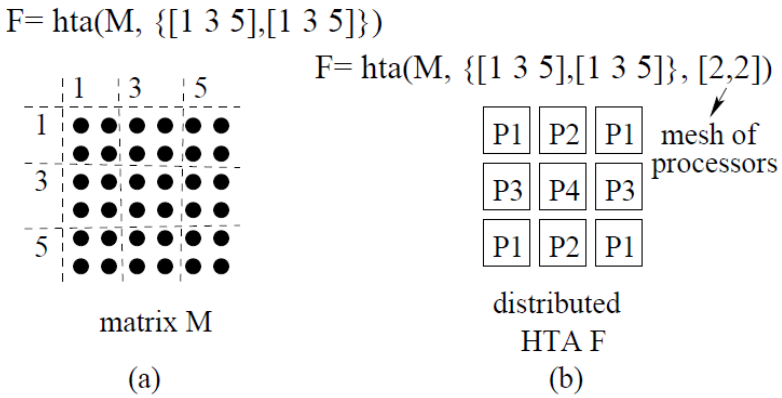

Figure 2.4: HTA construction-(a). Mapping of tiles to processors-(b) [BGH+06].

set of tiles by invoking the HTA constructor with the desired number of tiles. Invoking hta(3,3) creates an HTA with 3×3, whose contents may be posteriorly assigned.

After creating the desired tile topology, tiles can be distributed across processors with an additional constructor argument, as depicted in Figure 2.4-(b). In the example shown the 6×6 matrix is mapped on a 2×2 mesh of processors. The default implementation used in [BGH+06] uses a block cyclic distribution of the HTA tiles, which assigns tile (i, j) to processor $(i \bmod n, j \bmod m)$ in a $n \times m$ processor mesh.

Accessing HTA Components The components of an HTA can be accessed in a way analogous to the conventional array indexing, although some additional indexing notation is added in order to make it easier for programmers to express some more complex accesses and to distinguish between scalar data and tiles accesses. Figure 2.5 presents some examples of HTA components access. The expression `C(5,4)` indexes the scalar element in the fifth row and fourth column of `C`, as if `C` was a conventional array. Accessing

tiles is similar to accessing scalar elements, except that brackets are used instead of parenthesis, which results in the expression `C{2,1}` referring to the lower left tile of `C`. Regions that do not respect the tiling of `C` can be accessed using expression like `C(1:2,3:6)`, which returns a plain standard 2×4 matrix.

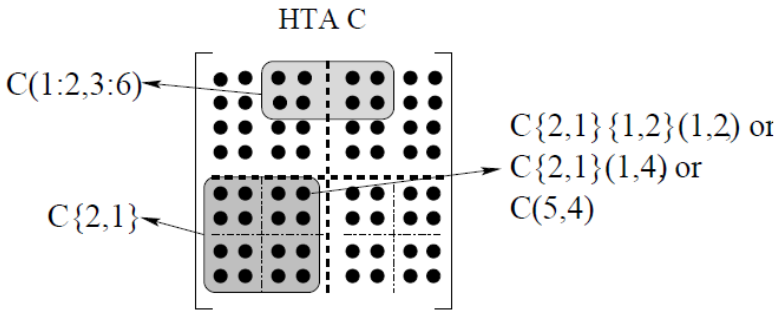

Figure 2.5: HTAs components access [BGH+06].

Communication Operations In HTA programs, communication is expressed using assignments on distributed HTA. Communication may also be expressed using other HTA operations, such as `permute`, `circshift` and `repmat` operations.

The `permute` operation transposes HTAs, altering the shape imposed by previous tilings. A variant named `dpermute` exists that performs a transposition solely over the HTA data, keeping the tiling structure.

The `Circular shift` operation allows a HTA to be circularly shifted, altering its overall topology.

The `repmat` operation is an important communication operation that allows a HTA, or part of it, to be replicated across processors.

Application Example: Matrix Multiplication An implementation of the conventional matrix multiplication algorithm using HTA can be observed in Listing 5.

```
1  function C = matmul (A, B, C)
2    if (level(A) == 0)
3      C = C + A * B;
4    else
5    for i=1:size(A,1)
6      for k=1:size(A,2)
7        for j=1:size(B,2)
8          C{i, j} = matmul(A{i,k}, B{k,j}, C{i,j});
9        end
10     end
11   end
12 end
```

Listing 5: Recursive matrix multiplication that exploits cache locality [BGH+06].

If A, B and C are tiled arrays, the matmul function can be applied in parallel to each triplet of corresponding tiles using the parHTA function with the matmul function and the matrixes to be multiplied as arguments, resulting in the expression:

```
C = parHTA(@matmul, A, B, C)
```

Note that the "+" and "*" operators are overridden to represent the scalar matrix multiplication, thereby making sense of the expression `C = C + A * B` at line 3. The `level` function can be used to obtain, at runtime, the location of the given argument within the tile hierarchy, returning 0 for a leaf (scalar matrix) and non-zero for tiles according to their location in the hierarchy. Spatial locality is exploited since processors access data sequentially in each tile.

2.2.3 Hierarchical Place Trees

Hierarchical Place Trees (HPT) [YZGS10] model, developed in 2010 by a team of researchers from the Department of Computer Science of the Rice University, aims to overcome the restrictiveness of Sequoia's communication mechanisms. Communication in Sequoia is limited to parameter passing during function calls.

HPT supports three different types of communication: implicit access, explicit in-out parameters, and explicit asynchronous transfer. In addition, HPT allows dynamic task scheduling, rather than static task assignment as in Sequoia.

Several concepts used in HPT were introduced by the X10 language [CGS+05], namely the concept of `place` and `activity` (task).

Hierarchical Place Trees Model In the HPT model each memory module is abstracted as a *place*, and therefore a memory hierarchy is abstracted as a `place tree`. Places are tagged with annotations that indicate their memory type and size. A processor core is abstracted as a *worker thread*, which in the HPT model can only be attached to leaf nodes in the place tree. The removal of this last restriction has been considered in order to accommodate *processor-in-memory* hardware architectures in the HPT model.

As in X10, a task can be directed to place PL_i by using a statement of the form `async (`PL_i`)`, which should be read as "move the current task to the place PL_i asynchronously". However, unlike X10, the destination place may be a leaf node or a non-leaf node in the hierarchy. If the target of an `async` statement is a non-leaf place PL_i, then the task can be executed on any worker that belongs to the subtree rooted at PL_i. A consequence of this constraint is that a worker can only execute tasks from its ancestor places. It is also assumed that if a task is suspended while executing at a worker `w0`, it can resume its execution at any worker that belongs to the subtree of the task's original target place.

As an illustrative example, consider the place tree presented in Figure 2.6; a task T assigned to PL_1 can only be executed by workers `w0` or `w1`. Also, if T is suspended while executing at `w0`, it is assumed that T can resume its execution at `w1`.

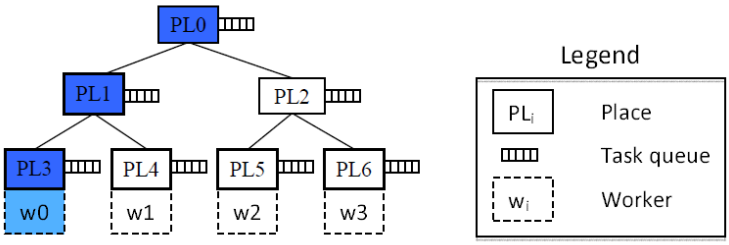

Figure 2.6: A Hierarchical Place Tree example, taken from [YZGS10].

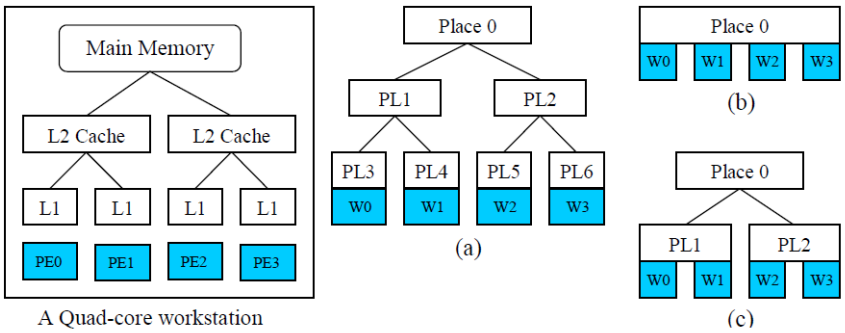

Figure 2.7: Three different HPT abstractions a, b and c of the same machine.

Compilation and Mapping to a Real Architecture The steps required to program and execute an application using the HPT Model are not a simple straightforward compilation process. A machine-independent compilation step handles parallelism and locality, which is abstractly expressed in the program to allow it to work with any *configuration specification*, which is machine-dependent. The configuration specification consists of an HPT model that represents the desired view of the system, and a mapping of the places and workers in the HPT model onto the memories and processor cores in the target machine. This allows the same program to be executed with different configuration specifications.

It is common to use different configurations as abstractions for different hardware systems, yet it is also possible to use different configurations as alternative abstraction of the same physical machine. Consider the example of Figure 2.7; the model `(a)` mirrors the machine's structure, whereas in `(b)` all the memory is seen as a flat shared memory with uniform access, and in `(c)` the memory is seen as a two-level memory hierarchy.

It is not easy to develop an interface for data distribution and transfer that is both portable and can be efficiently implemented across different memory systems. In symmetric multiprocessing (SMP) machines, data distribution follows implicitly from the computation mapping, whereas distributed memory machines and hybrid systems with accelerators require explicit data distributions and transfers. In order to accommodate

this, HPT model builds on the idea of a PGAS, extending it with the idea that the partitioning is not flat and may occur across a place tree hierarchy. Partitioned Global Address Space (PGAS) is a parallel programming model where a global address space is assumed. In the PGAS model each thread is given a global address space in addition to its local address space, which can be accessed by other threads for communication, thereby providing a powerful abstraction to program distributed applications using shared memory semantics. Figure 2.8 illustrates the PGAS model, showing each thread's global and local address spaces. As with distributed address spaces, each portion of the global address

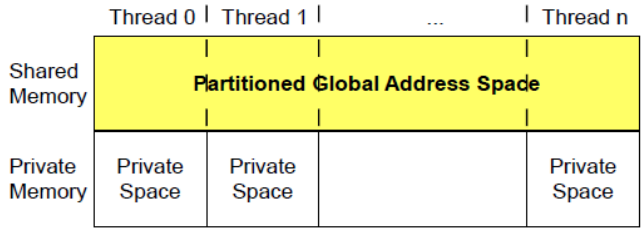

Figure 2.8: PGAS Address Space Model [CDC+99].

space has affinity with a given thread, that thread benefits from locality of reference when accessing data in its portion of the address space. Due to performance reasons, data locality is exposed to the programmer.

The HPT model provides three data transfer mechanisms: 1) data distribution with implicit data transfer; 2) explicit copyin/copyout parameters, and 3) explicit asynchronous data transfer.

All data structures that may be accessed implicitly using global addresses are required to have a well-defined *distribution* across places. Furthermore, each scalar object is assumed to have a single *home place*. Accessing any part of an object results in a data transfer from the home place to the worker that is going to perform the access; the access cost will depend on the distance between the home place and the worker.

Data can also be explicitly synchronously transferred using `IN`, `OUT` and `INOUT`, analogous to a dataflow model. When a task is launched at its destination place, data specified by `IN` and `INOUT` clauses will be copied into the temporary space of the destination place. Once the task finishes its execution, the data specified by the `OUT` and `INOUT` parameters will be copied back to their original locations.

Despite the powerful semantics of the presented mechanisms, there are situations where it is desired to perform the data transfer asynchronously so that it may be performed in parallel with computation in the caller and callee tasks. In order to support such situations, HPT introduces the `asyncMemcpy(dest,src)` which can be used to initiate an asynchronous data transfer between places.

2.2.4 Hierarchical SPMD

Early work that shares our concerns regarding the limitations of the SPMD model and the tuning of parallel applications according to the target hierarchy is first presented in [KY14]. The author also mentions the limitations of the Sequoia model, namely the lack of more powerful communication mechanisms that was also addressed in Section 2.2.3, as well as its intrusion to the programmer, who is obliged to provide hierarchical mappings and tunable configurations for its programs.

Hierarchical Thread Teams One of the most important concepts introduced in Hierarchical SPMD (HSPMD) is that of a team of threads, which was presented in [KY12]. Figure 2.9 shows a thread team hierarchy that results from the division of a initial team of 12 threads labeled from 0...11 into 3 teams, with each subteam being further divided into two uneven teams, with 3 threads and a single thread respectively. Thread teams are supported by additional linguistic constructs for expressing data decomposition among them.

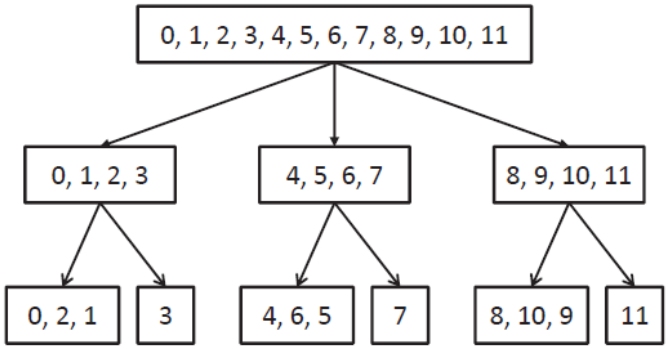

Figure 2.9: Team hierarchy example [KY12].

Task Decomposition When implementing a task parallel program, different parts of an algorithm may be assigned to different threads. The *partition* statement is introduced as a syntactic construct to assign tasks to different subteams of threads:

$$\textbf{partition}(T)\ \{B_0\ B_1\ ...\ B_{n-1}\}$$

where T represents a thread team and each subteam of T is assigned a different codeblock B_x for execution. Once a subteam finishes executing a partition branch, it rejoins the previous thread team. Execution only moves to the code after the partition statement, with the whole team, when all the subteams executing the codeblocks finish their execution.

Data Decomposition Some algorithms can benefit from a domain rather than functional decomposition. From all the features of these hierarchical additions, this is perhaps the most interesting one in the context of this thesis.

The *teamsplit* statement, with the following syntax, allows a data centred decomposition to be expressed:

$$\textbf{teamsplit}(T)\ B$$

Once again T represents a team of threads and the construct causes each thread team in T to execute the codeblock B. Each thread belonging to a subteam t of T executes in the context of t, which means that the thread identifier obtained using the invokation Ti.thisProc () is its offset in t.

An interesting motivational example for the *teamsplit* construct is the implementation of the shared memory sort algorithm. In the shared memory sort algorithm, data is evenly distributed amongst the available threads and each thread executes the sequential quicksort over its own data. The sorted subsets are then merged in parallel in multiple phases, with the number of threads executing the merge being halved in each phase. Figure 2.10 illustrates this algorithm executing with four threads.

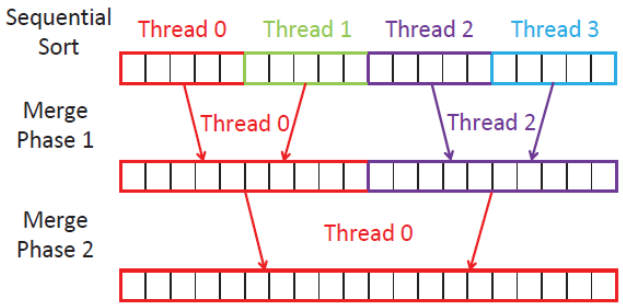

Figure 2.10: Shared memory sorting algorithm using four threads [KY12].

Using a team hierarchy structured as a binary tree, the recursive nature of the sorting can be intuitively expressed. Listing 6 presents the `divideTeam` function that creates such hierarchy.

```
1 static void divideTeam(Team t) {
2   if (t.size () > 1) {
3     t.splitTeam (2) ;
4     divideTeam(t.child (0) ) ;
5     divideTeam(t.child (1) ) ;
6   }
7 }
```

Listing 6: DivideTeam method, taken from [KY12].

After creating the hierarchy using the `divideTeam` method, it can be passed as an argument to the `sortAndMerge` method depicted in Listing 7. The `sortAndMerge` method is recursively called within `teamsplit` statements in order to travel to the bottom of the hierarchy, where teams with a single thread reside. These threads execute a sequential sort algorithm over their assigned chunk of data in parallel, returning to original thread team upon finishing. At each level, the thread with rank 0 in its current team is responsible for merging the results of the other threads, performing the merge previously illustrated in

```
1  static single void sortAndMerge(Team t) {
2    if (Ti.numProcs () == 1) {
3    allRes[myProc] = SeqSort.sort(myData ) ;
4    } else {
5    teamsplit(t) {
6      sortAndMerge(Ti.currentTeam () ) ;
7    }
8    Ti.barrier () ; // ensure prior work complete
9    if (Ti.thisProc () == 0) {
10     int otherProc = myProc + t.child(0).size() ;
11     int[1d] myRes = allRes[myProc] ;
12     int[1d] otherRes = allRes[otherProc] ;
13     int[1d] newRes = target(t.depth() , myRes ,otherRes ) ;
14     allRes[myProc] = merge(myRes , otherRes ,newRes ) ;
15   }
16  }
17 }
```

Listing 7: Shared memory sort implementation with thread teams, taken from [KY12].

Figure 2.10.

2.2.5 Unified Parallel C

UPC [CDC+99] is a parallel extension of the C programming language ISO C99 that adopts the SPMD programming model with a PGAS address space. Hierarchical additions to the UPC language are proposed in [WMEG11].

The authors recognize that, although compiler and runtime optimizations increase the efficiency of a program up to a certain degree, when dealing with architectures with complex memory hierarchies the greatest performance gains can only be attained by optimizing the way the application uses these hierarchies. However, the increasing levels of the hardware hierarchy and the wide variety of system architectures make it difficult for compilers to perform efficient tunnings. Therefore, it is highly unlikely that compiler or runtime systems will perform a good distribution of data and tasks across the hierarchy to meet the user's computational needs. This means that one should emphasize enhancements of the programming model as much as improving the compiler or runtime libraries.

Two complementary approaches to manage and express hierarchical parallelism at the application level are studied in the paper.

Approach 1: Exploiting Hierarchical Parallelism Using Thread Groups The first approach manages the UPC language threads as sets of thread groups. Thread grouping is performed according to the runtime thread distribution, which provides the programmer with thread group identifiers that improve the language another level of parallelism. Using the runtime support, the programmer can specify where on a machine should specific parts of the computation be executed, thereby allowing optimizations on data sharing and communication between related threads to be expressed by placing these into the same level or node in the hierarchy.

During runtime, the location of a thread within the hardware topology can be retrieved using a specific thread layout query function. Though the resolution is low, the function can accurrately distinguish between a remote and a local thread. This functionality allows a programmer to explicitly identify which UPC threads share the same physical address space, enabling optimizations in the access to neighbours's shared arrays and henceforth.

Approach 2: UPC/sub-threads Hybrid Model The second approach proposes the direct addition of nested parallelism in UPC using *sub-threads*. Sub-threads are layered on each SPMD UPC thread and execute within the same partition of the shared address space as the corresponding master thread.

Although this model is similar to the hybrid MPI/threads model, differences exist that set these two models apart:

- Sub-threads in UPC can access remote distributed memory directly using the global address space;
- Global shared arrays are the primary constructs for parallel programming.

UPC sub-threads represent a new level of parallelism that allows the natural parallelism of algorithms to be captured, as UPC programs no longer have to cope with a single-level of execution. Moreover, sub-threads allow local computational resources to be fully exploited in distributed UPC applications.

2.2.6 Fractal Component Model

Hierarchical organization of parallel systems can also be found at integration level. The Fractal model [BCL+06] is a powerful component programming model for diverse complex applications. One of Fractal's most relevant features in the context of hierarchical parallelism is its composition model. Figure 2.11 illustrates some examples of Fractal components.

The component *A* is a primitive component since it is not internally composed by any other component. On the other hand, *C* is a composite component since it is composed by other components that attend different server ports. The sub-components can themselves be primitive, composite or parallel components. Finally *D* is a parallel component because its components do not connect between themselves, attend to the same server port, and output to different client ports.

We can observe that the Fractal composition model allows complex parallel component hierarchies to be created. As an example of a real world application modelled using Fractal components, consider the graphical 3D renderer depicted in Figure 2.12.

Users are also represented as components, with User 2 being connected to the *Dispatcher* component, which in turn connects to the *Renderers* with both a client port for

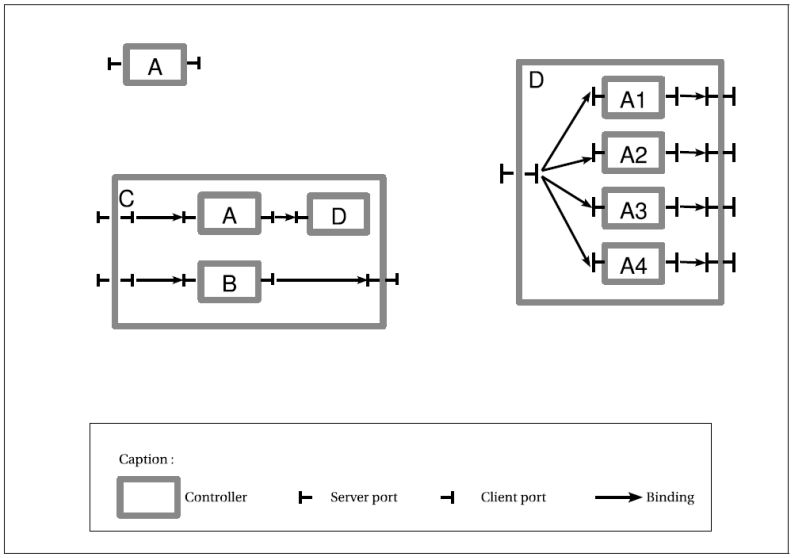

Figure 2.11: Fractal component types examples, taken from [BBC+06].

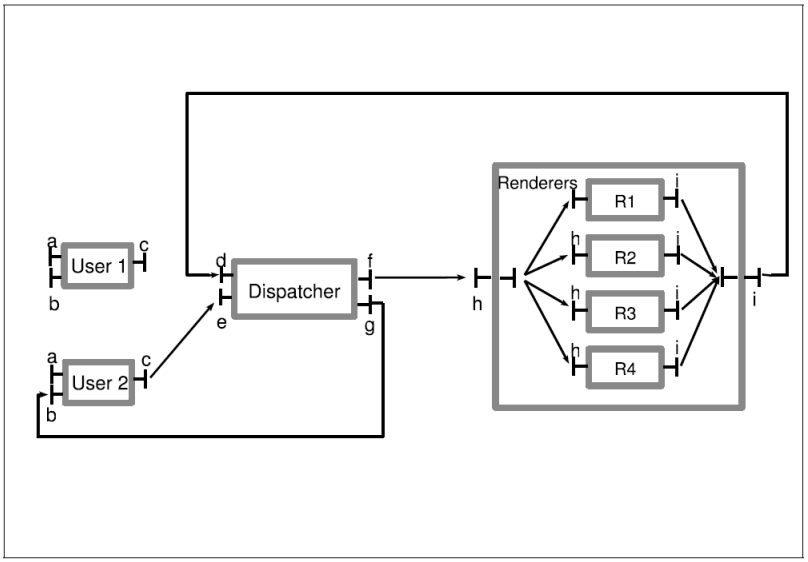

Figure 2.12: 3D renderer component model, taken from [BBC+06].

output and a server port to receive the input from the Renderers. The Renderers themselves are represented as a parallel component with 4 sub-components R1 to R4. From the Dispatcher's point of view, the Renderers function as a primitive component with a client port and a server port, with the Renderer's inside being a black box.

The Fractal model was later instanced in the ProActive [BBC+06] middleware, in the context of its parallel implementation. Component hierarchies are defined using a XML framework.

2.3 Hierarchical Work Distribution

Distributing parallel tasks across processors, in a way that maximally exploits locality of access to data, is not the only concern to have into account when optimizing the execution of parallel applications. There is space for optimizations regarding communication amongst tasks and the even distribution of computation across the available hardware resources, so as to prevent unnecessary high latency communication and idle times on computational resources, respectively.

The paper entitled "Hierarchical Mapping for HPC Applications" [CLZC11], published recently in 2011, addresses the problem of distributing tasks of a parallel application onto physical processors of a computational architecture in a way that not only minimizes the communication costs, but also evenly distributes the computation across the processors.

In this paper an algorithm called "Hierarchical Mapping Algorithm" (HMA) is presented, which given a hardware physical topology modeled as a graph, approximates the optimal distribution of tasks across the topology with the aforementioned concerns in mind. HMA provides better scalability than the static mapping method, which maps tasks into computational resources based on static information. Such method requires a large amount of detailed analysis that can be impraticable both in terms of design complexity and execution time, and also requires the decomposition of the problem and the topology of the host machine to be known beforehand.

The HMA targets large-scaled applications and uses the hierarchical mapping approach. Instead of using static information, HMA builds a graph representative of the parallel tasks and communication based on profile information collected using the MPI tracing tool. The algorithm was designed to support complex systems, therefore it can handle:

- applications with irregular parallelism or complex communication patterns;
- computing systems with high dimensional interconnection;
- mapping efficiency in large scale computing systems.

The algorithm is composed of three stages: task partitioning, initial mapping and fine tuning.

Supernodes An important concept of the HMA is that of a supernode. Supernodes are nodes in the hierarchy topology graph where tasks with strong relations are placed. In the initial mapping phase, "supernodes" are mapped onto the host machine.

For a given k, the task partitioning part of the HMA algorithm clusters V into k groups that form the set of supernodes $\{A_1, A_2, ...A_k\}$.

Graph Construction Before executing the algorithm, a weighted graph $G(V, E, w)$ is built based on profile information collected during run-time using the MPI tracing tool. V is a set of tasks, E represents the communication relations between tasks, and $w(u, v)$ corresponds to the message size exchanged between the tasks u and v.

Task Partitioning During task partitioning the algorithm has two objectives. The first objective is the *cohesion criterion*, which asserts that the communication traffic within a supernode has to be higher than traffic between supernodes, thereby reducing the amount of high latency communication. Second, there is the *equality criterion*, which means that the size of each supernode should be as equal as possible.

Initial Mapping The initial mapping places all *supernodes* onto the same target host machine. Then the algorithm resumes its execution with the criterion of minimizing the communication costs among supernodes.

Fine Tuning Last in the HMA execution flow, the fine tuning phase employs various optimization techniques that include the simplex method, the local search method, or the simulated annealing, so as to improve the mapping generated during the initial mapping phase.

2.4 Discussion

In the presence of multiple CPU cores, the most straightforward approach is to perform a horizontal decomposition of the domain, splitting the original dataset into as many partitions as the number of available workers.

Although this approach makes use of the parallelism available on the underlying hardware architecture, it is based solely on the number of cores available in the machine. Nonetheless, there are other traits that differentiate a particular architecture from others featuring the same number of cores, namely its cache hierarchy. To fully harness the computational power of a machine, one has to employ a decomposition strategy that takes these hardware differences into account.

With this knowledge in mind, a vertical decomposition strategy can be considered to take into account not only the number of processing cores, but also the target machine's cache hierarchy. We may call this a hierarchical decomposition.

The presented state-of-the-art approaches present both means to model machines, and to employ vertical decomposition, namely how to express it and program applications in a machine-independent manner.

The Sequoia language introduced the concept of task variants and tunable variables, which along with a mapping specification provided by the programmer tune applications according to the target machine(s).

HPT proposes modelling a system as a place tree and its processor cores as workers attached to the leaves of the former, defining a view of the system. This view is subsequently mapped onto a real machine according to a configuration specification, provided by the programmer.

Still, all these proposals place a heavy burden upon the programmer, who is required to have a deep knowledge of the machine in hands when defining applications and the mapping of these onto the target machine's hardware.

3

Hierarchical Domain Decomposition

Domain decomposition is the most common and natural of parallel decomposition strategies in data-parallel computing. It consists on decomposing the original domain of an operation into several partitions upon which different instances of the original operation can be applied in a parallel fashion.

Chapter 2 presented the most recent efforts on hierarchical parallelism, and its vertical take on domain decomposition. Common to all the presented approaches is the additional burden they require from the programmer, in order to exploit vertical decomposition, namely the mapping of applications onto the target machine's hardware. The programmer is therefore required to have knowledge of the hardware in hands, whereas its focus should rely solely upon the application's logic and data manipulation. Hence, we believe that it is possible to automatically employ a vertical decomposition approach that:

1. determines the optimal size for partitions of the original domain, to be assigned to each individual task;

2. partitions the original domain into partitions with the pre-determined size;

3. orchestrates the whole execution from the invocation of the original operation to the obtainal of its final result.

An automated approach removes hardware-related concerns from the programmer's shoulders, allowing him/her to focus solely on the application's logic and data representation and manipulation.

3.1 Data-size Driven Decomposition

When decomposing a domain D into a set P of partitions that fit a given target cache level (TCL), one wants to find a number N of partitions into which D can be partitioned so that the following property holds:

$$\forall p \in P,\ size(p) = \frac{size(D)}{N} \leq size(TCL) \tag{3.1}$$

For that sake of generality, we assume that the original domain D may itself be the result of composition of multiple subdomains $D_0, ..., D_{n-1}$. For instance, in the classic matrix multiplication, one can implement a decomposition strategy that decomposed the 3 matrices involved, or decomposed the 3 individually and have a way of combining the resulting individual partitions. To accommodate the latter case, formula 3.1 has te be refined to

$$\forall p \in P,\ size(p) = \sum_{i=0}^{n-1} \frac{size(D_i)}{N} \leq size(TCL) \tag{3.2}$$

It is not always possible to achieve such a obvious solution. To that fact mostly contributes the fact that even though $size(D)$ may be a multiple of N, a non-zero remainder $R_i = size(D_i) \bmod N$ may exist for some sub-domain D_i. In these situations, one may either produce a smaller dataset of size R_i, preserving property 3.2, or one may choose to distribute the bytes of R amongst the regular-sized partitions, which may afterwards contain more bytes than the size of the TCL, violating the property. The first solution can not be applied if a non-zero remainder R_i exists only for some sub-domains of D, which would cause a different number of partitions to be produced from each sub-domain.

Problem-specific constraints may impose further restrictions upon the number of partitions and/or the geometry of the decomposition as a whole. Stencil computations present this kind of restrictions, both in terms of the number of elements and the geometry of a partition. A *stencil* computes an element in a *n*-dimensional grid at time t as a function of adjacent elements of the grid at time $t-1, ..., t-k$. Consider a simple stencil computation that computes the value of an element (i, j) in the grid at time t_1 as

$$g1_{i,j} = \frac{g0_{i,j}}{2} + \frac{g0_{i+1,j}}{16} + \frac{g0_{i-1,j}}{16} + \frac{g0_{i,j-1}}{16} + \frac{g0_{i,j+1}}{16} + \frac{g0_{i+1,j+1}}{16} + \frac{g0_{i-1,j+1}}{16} + \frac{g0_{i+1,j-1}}{16} + \frac{g0_{i-1,j+1}}{16}$$

where $g0$ and $g1$ denote the grid at time t_0 and t_1 respectively. Corner cases (elements without 8 adjacent elements) are computed using different weights for the adjacent values.

The value of an element at time t_1 is a function of the values of its 8 adjacent elements plus itself at time t_0. Therefore, each partition is required to have at least 9 elements and must contain the elements that comply to the presented dependency constraint. Given these restrictions, one can conclude that a valid partition of the grids $g0$ and $g1$ must comprise at least 3 sequentially ordered columns and lines, all equally-sized. Figure 3.1

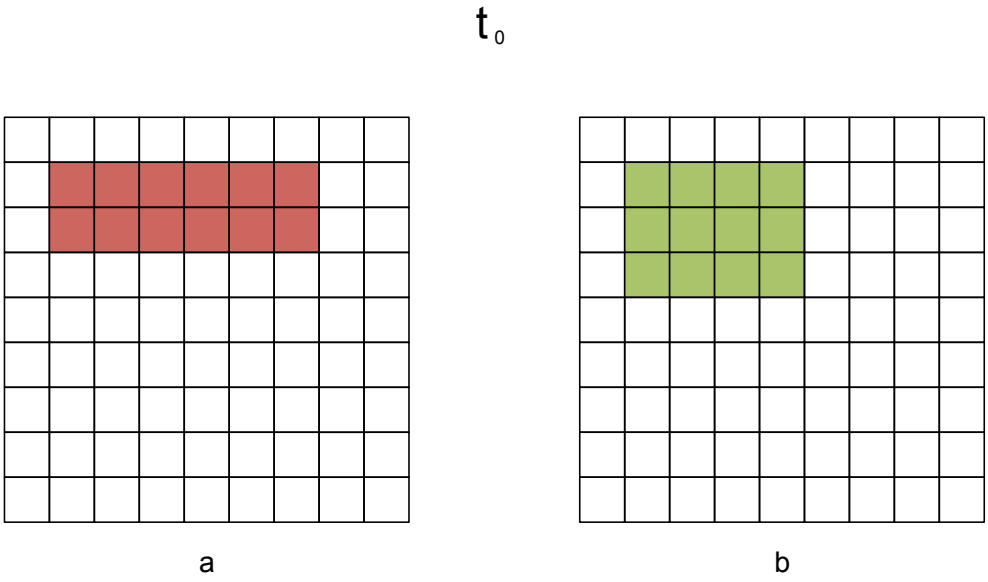

Figure 3.1: Invalid (a) and valid (b) Stencil Partitions

illustrates an invalid (a) and a valid (b) partition of $g0$. Note that the computation to be applied to each partition p of $g0$ only produces a result (to be placed in $g1$) for the inner elements of p.

This kind of information must be supplied in the decomposition algorithms supplied by the programmer, and is therefore included in the interface that regulates the implementation of such algorithms (Listing 8). Method `getAverageLineSize` is of particular relevance. It validates whether the dataset may be split into the supplied number of partitions, and, when such condition holds, returns the average size (in number of elements) of the first dimension of such partitions. We are assuming a row-major order memory layout, which is typically the case in most programming languages, including Java, in which we implemented our proposal. This information is therefore relevant to understand the decomposition of a partition into cache lines. The use of the average value conveys some extra information to the system when the size of the partitions is irregular. This situation may occur, for instance, in the aforementioned situation, where the remainder R is positive and one chooses to distribute these bytes across regular-sized partitions.

Algorithm 1 builds upon the information provided by the distributions to determine the maximum size of a workingset that fits the given TCL. A workingset is defined as the set of partitions, one from each sub-domain (handled by the respective distribution), upon which an individual parallel task operates. To determine the maximum size of a workingset, the algorithm continuously decreases the size of the partitions of each individual sub-domain, so that their total size (*workingSetSize* at line 10) is valid with respect to each distribution involved (line 8), and fits the TCL (line 11). The algorithm assumes that the input argument `TCL_PER_CORE` is the amount of memory expected to be available

```
1  public interface Distribution<T> {
2
3    /**
4     * Partitions the input domain into nParts partitions.
5     * @param nParts the number of partitions to be produced
6     * @return the partitions
7     */
8    T[] partition(int nParts);
9
10   /**
11    * Returns the average size of a partition of T (in number of elements)
12    * @return size of P
13    */
14    float getAveragePartitionSize(int nParts);
15
16   /**
17    * Returns the average size of line of a partition of T (in number of elements)
18    * @return size
19    */
20   float getAverageLineSize(int nParts);
21
22   /**
23    * Returns the size of an element of T (in bytes)
24    * @return size
25    */
26   int getElementSize();
27 }
```

Listing 8: The `Distribution` interface

for each core sharing a TCL, calculated as $^{TCL\_SIZE}/_{CORES\_PER\_TCL}$.

Accordingly, the number of partitions ($nParts$), into which each domain can be decomposed, ranges from the number of workers assigned to the execution ($nWorkers$) up to a value n that represents a valid solution, meaning that should each domain D_i be decomposed into n partitions all these would fit the TCL. Values of $nParts$ lower than the number of available workers are not considered since these would not fully exploit the parallelism available on the machine. Although this prevents any single worker from having no tasks to execute, it does not guarantee that the number of tasks assigned to each thread is evenly balanced. This kind of will be discussed in Section 3.2.

Since the size of each individual partition decreases with the increase of $nParts$, which is continuously incremented in each iteration, the first valid value of $nParts$ is the optimal solution, that is, each partition has the maximum size that fits the TCL.

Central to the algorithm is the computation of the size of each individual partition that compose a working set (lines 5 to 9). The procedure starts by invoking the method `getAverageLineSize` to validate the current value of $nParts$. Depending on the returned value, a different course of action is taken. Value 0 invalidates the current value of $nParts$ as a solution, causing the algorithm to procede to the next value.

A negative answer invalidates the current proposal for $nParts$, as well as any value $n' > nParts$ causing the algorithm to stop since no valid solution will be found. When this occurs over a distribution of a given domain D it means that should D be partitioned into n or any number $n' > n$ partitions, each partition would not meet the geometrical restrictions, or contain the minimal number of elements imposed by the considered

Algorithm 1: Working set size estimation

Input: `TCL_PER_CORE` - Size in bytes of the TCL per core
Input: `CACHE_LINE_SIZE` - Size in bytes of a cache coherence line
Input: $nWorkers$ - Number of CPU cores available on the machine
Input: $nDatasets$ - Number of datasets to be decomposed
Input: $dists$ - Vector holding the distribution algorithms for each dataset
Input: φ - Function that determines the size of a partition

```
1  for nParts ← nWorkers to ∞ do
2      workingSetSize ← 0;
3      valid ← true;
4      for i ← 0 to nDatasets do
5          partLineSize ← dists[i].getAverageLineSize(nParts);
6          if partLineSize = -1 then return -1;
7          valid ← (partLineSize > 0);
8          if valid = false then break;
9          partSize ← φ(CACHE_LINE_SIZE, dists[i], partLineSize, nParts);
10         workingSetSize ← workingSetSize + partSize;
11     if valid and workingSetSize ≤ TCL_PER_CORE then return nParts;
```

problem.

A positive answer (> 0) is used to compute the size of a partition (in bytes) of the current distribution. The actual calculus of the partition size is delegated on function φ, which is supplied as a parameter in order to allow for different approaches to be employed. The definition of this function implies a trade-off between accuracy, computational overhead, and wasted cache space.

A simple definition of φ is one that computes the number of bytes in a partition, without taking in consideration both the size of the target architecture's cache line size (`CACHE_LINE_SIZE` in the algorithm) nor the partition's average line size:

$$\varphi(cacheLineSize, dist, partLineSize, nParts) = dist.\texttt{getElementSize}() \times \lfloor dist.\texttt{getAveragePartitionSize}(nParts) + 0.5 \rfloor$$

The average size of a partition is rounded to the closest integer by adding 0.5 to the result and subsequently applying the *floor* operator to the result. Rounding the result to the closest integer guarantees that most of the partitions will conform to the computed partition size. This definition of φ is oblivious to the granularity of data fetching from main memory into the caches by cache controllers, which is that of a cache line's size. This means that even if the size of a contiguous sequence of bytes in a partition (the line retrieved from the distribution algorithm) is a multiple of the cache line's size, every cache line spanned by the byte range of the partition's line will be fetched into the cache.

To illustrate this problem, consider an architecture with a cache line size of 64 bytes, and a partition line L with size 448 bytes. The best case scenario for the space occupied by L in the cache occurs when L is memory aligned and the size of its byte range is a multiple of the cache line size, which is illustrated in Figure 3.2. In this situation, L occupies as

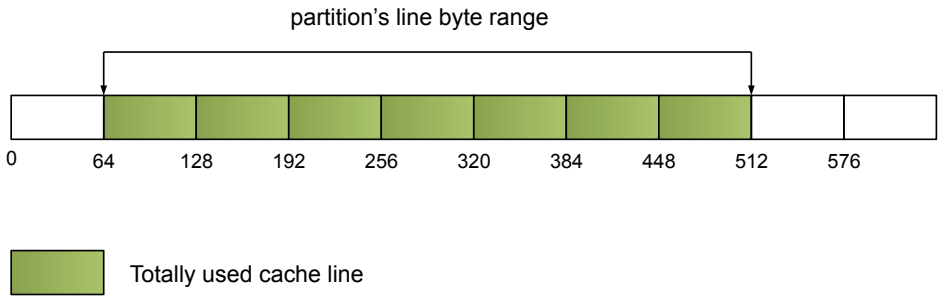

Figure 3.2: Best case scenario of a partition's line mapping onto cache lines

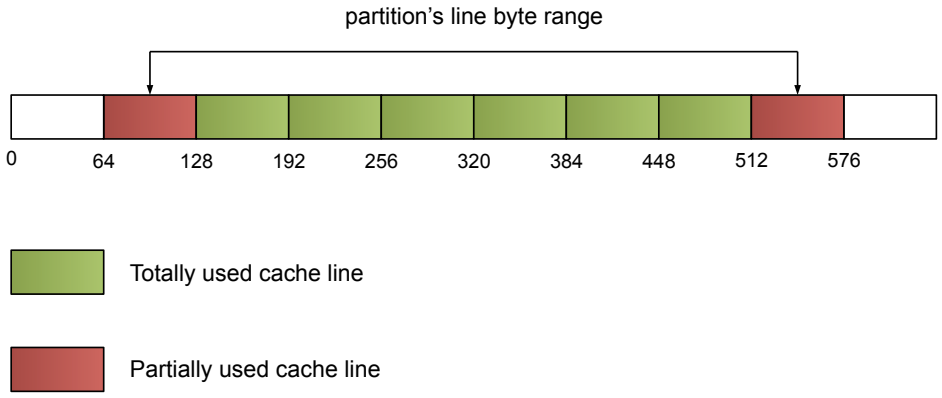

Figure 3.3: Worst case scenario of a partition's line mapping onto cache lines

much space in cache as its length in bytes, which equals 448 bytes = 7 cache lines. The above definition of φ is accurate if and only if this situation holds for every line in every partition. Other situations have to be considered, one of these being when a partition line L is memory aligned, but its size is not a multiple of the cache line size, which results in L occupying $(\frac{448}{64}+1)$ cache lines in cache. The worst case scenario for the space occupied by L in cache occurs when both L is not memory aligned and its byte range intersects one additional cache line; suppose that L starts at address 96, as depicted in Figure 3.3. Given these conditions, L will occupy $(\frac{544-96}{64}+2)$ cache lines in cache. This leads us to a more conservative definition of φ, whose estimation may take these dimensions into account, at the expense of more computational overhead:

$$\varphi(cacheLineSize,dist,partLineSize,nParts) = cacheLineSize \times \lceil \frac{dist.\texttt{getAveragePartitionSize}(nParts) \times dist.\texttt{getElementSize}()}{partLineSize \times cacheLineSize} \rceil + 1$$

The size of the partition's first dimension is adjusted to the boundaries of the cache line. Furthermore, an extra cache line is added to considerate the eventual misalignment of the partition to such boundaries. This approach is likely to ensure that the entire working set fits the TCL, but its conservative nature may eventually waste more space than the first approach. Since most programming languages allocate memory on memory aligned

positions, one can expect a partition whose lines start on the first position of arrays to follow a best case scenario if their size is also a multiple of the cache line size. However, if the latter does not hold, these lines and all subsequent lines would conform to one of the other two cases.

None of the presented strategies take into consideration the cache associativity, which relates to the cache collisions of different memory addresses. We assume that if the size of a workingset is lower than the amount of memory in the TCL, all data in the workingset will co-exist on the TCL at some point in time. Although this is true in a fully associative cache, it may not hold in a n-way associative cache when the workingset is not composed of totally contiguous data. Furthermore, we assume a cache replacement algorithm of the *Least-Recently-Used* (LRU) family.

3.2 Scheduling

Once an operation's domain is decomposed into the multiple workingsets, the original operation will be carried out multiples times, in parallel, upon each of these workingsets. We will denote these pairs (instance of the operation and associated worksingset) as tasks. The act of mapping tasks onto worker threads is called *scheduling*.

There are two main types of scheduling: static and dynamic. In the first, the set of tasks assigned to a worker is statically defined before the latter begins its execution, whereas in dynamic scheduling, this set of tasks is built dynamically throughout the execution, depending on non-deterministic factors. A simple example of a dynamic scheduling situation occurs when workers fetch tasks from a shared task queue for execution, until the queue is empty.

In the presence of a horizontal domain decomposition approach, each worker thread is usually assigned a small set of tasks (typically comprising a single task) for execution. When employing a hierarchical domain decomposition approach, this task assignment strategy cannot be generally applied, since the amount of tasks will largely exceed the number of available workers (which is ruled by the number of CPU cores).

As a concrete example, consider a block decomposition for the parallel computation of the classic matrix multiplication problem, which is illustrated in Figure 3.4. A workingset must comprise a block of each input matrix and space for the computed result block, which will be placed in the output matrix. Note, however, that every block partition of the first matrix (A) must be paired with all the block partitions that compose a line of the second (B). In the example of Figure 1, all block partitions of matrix A are paired with three block partitions of matrix B, for instance block A1 with blocks B1 to B3.

Consider now a concrete instance of the problem where both A and B are 1024×1024 square matrices of 4-byte-long integers. Moreover, assume a TCL with 64KBytes. The workingsets resulting from the vertical decomposition of this input domain (comprising three matrices) will feature three blocks whose cumulative size equals 65368 bytes. In order to produce blocks with this size, each matrix will be divided into $14 \times 14 = 196$

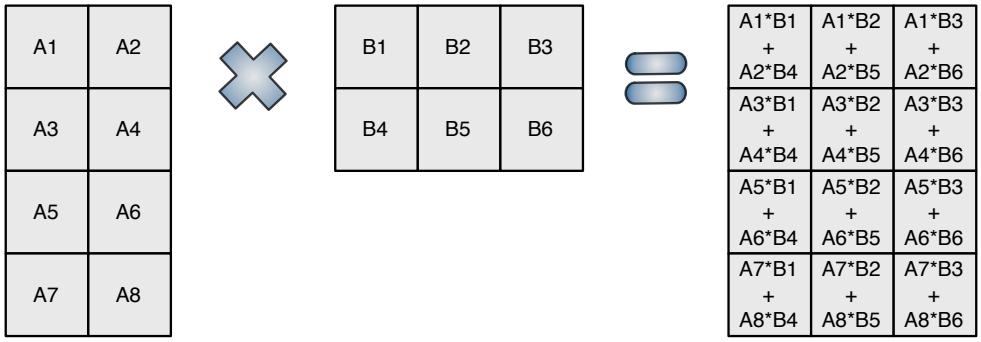

Figure 3.4: Block decomposition for the matrix multiplication problem

blocks. Since each block of A will have to be combined with 14 blocks of B, applying a one-to-one mapping from working sets to tasks will result in a total of $14 \times 14 \times 14 = 2744$ tasks.

Faced with this situation, one has to either launch more workers than the available cores on the machine to match the number of tasks, enabling a 1-1 thread-to-task mapping; or somehow schedule these tasks onto the the available workers. The former option is not viable in this context because having the number of workers far exceeding the number of computational resources penalizes performance, since the worker threads will have to spend time sharing the CPUs (due to context-switching by the OS) rather than executing tasks.

An alternative is to place all tasks in a shared queue and allow for the workers to consume tasks from this queue. This approach would not be feasible though, due to the small granularity of the tasks, which would cause workers to spend a considerable amount of time waiting for synchronized access to the queue.

To avoid this overhead, our approach is to perform an initial distribution of the tasks among the workers, so that each worker thread can be assigned a coarser grained task that will sequentially operate upon those groups.

We advocate that this initial work distribution increases the overall performance of the system. However, when in the presence of irregular computations, dynamic scheduling techniques can help to balance the load. We have not explored these techniques even though we considered them, namely the work-stealing and dynamic load balancing techniques alluded in [jMIY11] and [ZMBK10] that already embedded some hierarchical concerns.

In the static scheduling techniques we present, the challenge is to trade-off the efficiency of the mapping against the overhead (temporal and spatial) that the determination of the next task to execute might impose on the overall execution. More complex clustering strategies are expected to perform more calculations and require more control variables, thus stealing space in the cache for the actual working sets.

It is worth noting that since the task scheduling is done statically during runtime, the access to the global task set can be free of any synchronization mechanism, avoiding any

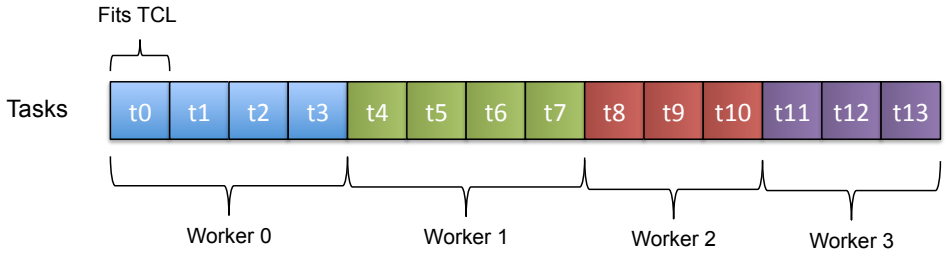

Figure 3.5: Contiguous Clustering: Worker-Tasks Mapping

overhead that could result from synchronization mechanisms.

3.2.1 Contiguous Clustering (CC)

Our first implemented clustering strategy assigns an equally-sized contiguous cluster of tasks to each worker. Given a set of workers $W = \{w_0, w_1, ...w_{n-1}\}$ and a set of tasks $T = \{t_0, t_1, ..., t_{m-1}\}$ each worker w_i is assigned a cluster of tasks ranging from task $(i \times \frac{m}{n})$ to task $(((i+1) \times \frac{m}{n}) - 1)$, where $\frac{m}{n}$ represents the integer division of m by n.

Situations may arise where the size of T is not a multiple of the size of W, in other words $R = m \bmod n \neq 0$. To cope with these situations, workers $\{w_0, ..., w_{R-1}\}$ are assigned one extra task for execution.

Figure 3.5 illustrates the worker-tasks assignment when employing contiguous clustering with 14 tasks and 4 workers. The rationale behind this strategy is twofold: 1) introduce minimal overhead during scheduling, 2) exploit spatial locality between task executions when contiguous tasks operate over contiguous data. The distributions we implemented for the studied problems guarantee that, whenever possible (line transitions in matrices pose exceptions), sequentially ordered partitions of a given domain contain contiguous data.

3.2.2 Sibling Round-Robin (SRR) Clustering

Seeking to tackle more traits of cache hierarchies, we devised the Sibling Round-Robin (SRR) clustering strategy. Whereas contiguous clustering is driven solely by the total number of workers, which in our implementation by default equals the number of cores, SRR clustering takes into consideration how these are distributed across the memory hierarchy, more specifically, how these share the Last Level Cache(s) (LLC).

Consider a machine with two CPUs, illustrated in Figure 3.6, each CPU features 4 cores and its own L3 cache, which is also the LLC. If two or more workers run in cores sharing data, the number of LLC misses decrements, reducing the number of accesses to main memory. Once again, the matrix multiplication example is paradigmatic, as multiple workingsets share blocks of both input matrices. Consider that the partition algorithm iteratively combines each block of matrix A to its counterparts in matrix B,

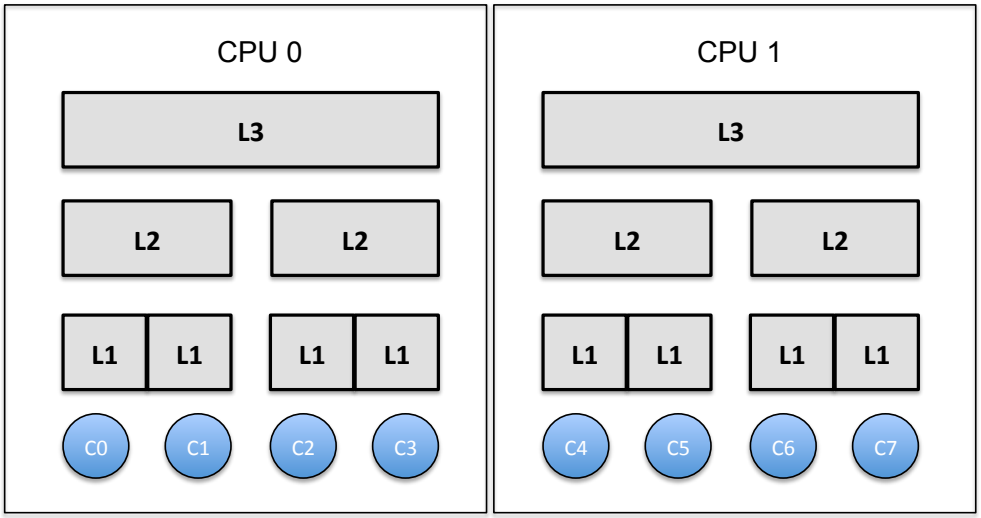

Figure 3.6: Example Cache Hierarchy

spawning a sequence of working sets with the following pattern:

$$(A_1, B_1), (A_1, B_2), \ldots, (A_1, B_p), (A_2, B_{p+1}), (A_2, B_{p+2}), \ldots, (A_2, B_{2*p}) \ldots$$

where A_i and B_i denote blocks of both matrices according to the layout depicted in Figure 3.4, and p corresponds to the number of blocks from B with which a block of A has to be combined. Revisiting the example of Figure 3.4, if the working sets comprising blocks (A_1, B_1), (A_1, B_2) and (A_1, B_3) are assigned to three different workers running on sibling cores that share the same LLC cache, the number of LLC misses generated by accesses to block A_1 will be limited to the number of cache lines the block spans.

From the choice of having the number of workers match the number of cores follows that the set of workers W has the same cardinality as the set of cores $Q = \bigcup_{i=0}^{n-1} Q_i$, where each set Q_i represents a set of cores sharing a LLC. Hence W can be defined as $W = \bigcup_{i=0}^{n-1} W_i$, where each set W_i has the same cardinality as Q_i. We will assume that the cardinality $|Q_i|$ of every subset $Q_i \subset Q$ is the same.

The assignment of tasks to workers is composed of two distinct assignment levels: *Outer-cluster-assignment*, the assignment of clusters to sets of workers; and *Inner-cluster-assignment*, the assignment of tasks to workers inside a cluster.

Given a set of tasks T with cardinality $|T|$, the set will be clustered into clusters of $N = {}^{size_{LLC}}/_{size_{TCL}}$ tasks, producing a set of clusters $C = \{c_0, c_1, ..., c_{k-1}\}$, where each cluster will be assigned to a single set of workers. Each set of workers $W_i \subset W = \{W_0, W_1, \ldots, W_{n-1}\}$ will be responsible for executing a set of task clusters $C_i \subset C$ such that

$$C_i = \{c_j \in C | j \bmod n = i \wedge j < k - k \bmod n\}$$

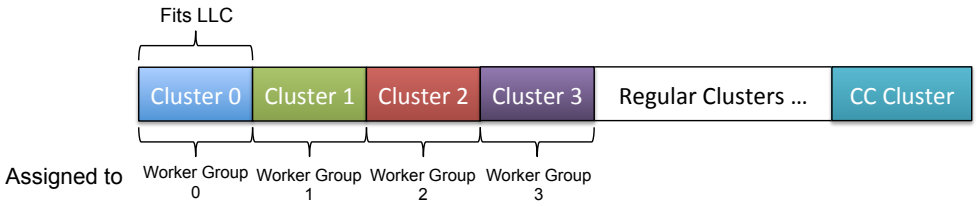

Figure 3.7: Task Clusters

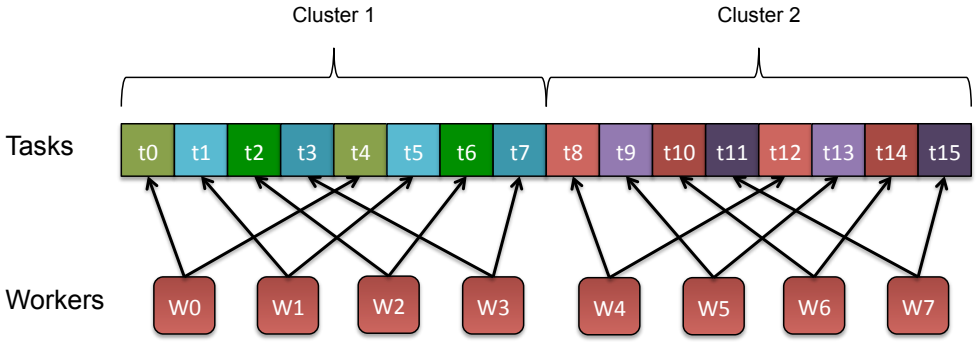

Figure 3.8: Sibling Round-Robin Clustering: Worker-Tasks Mapping

Since the number of workingsets that fit the LLC may not be a multiple of the number of cores sharing it, in order to prevent an uneven assignment of tasks to workers, the size N of the clusters is adjusted to $N = \frac{size_{LLC}}{size_{TCL}} + (Q_{LLC} - (\frac{size_{LLC}}{size_{TCL}} \bmod Q_{LLC}))$ tasks, where Q_{LLC} denotes the number of cores that share the LLC. This adjusts the considered number of working sets per LLC to the closest multiple of Q_{LLC}, greater than or equal to the number of workings sets that actually fit the LLC.

Figure 3.7 illustrates a clustered set of tasks for a machine with 4 groups of sibling cores sharing 4 different LLCs, along with the assignment of these to the worker groups. Note the additional cluster named *CC Cluster* depicted in the figure. This cluster has size equal to $N \times (|T| \bmod (N \times n)) + |T| \bmod N$ tasks and is composed of the tasks that could not form a cluster ($|T| \bmod N$) plus the clusters that could not be evenly assigned to the worker groups ($N \times (|T| \bmod (N \times n))$). This prevents an excessively uneven assignment of tasks to workers. The name of the cluster comes from the scheduling strategy employed to assign its tasks to the workers, which is the CC clustering strategy of Subsection 3.2.1.

The *Inner-task-assignment* of a cluster is performed in a *round-robin* fashion among the group of workers (who are siblings among themselves) operating over the cluster, which gives rise to the name of this strategy. Each worker w_i on the worker group $W' = \{w_0, w_1, ..., w_{n-1}\}$ that was assigned the cluster of tasks $T = \{t_0, t_1, ..., t_{k-1}\}$ will execute the subset of T containing all tasks t_j such that $j \bmod n = i$. Figure 3.8 illustrates the inner-cluster-assignment of two clusters to two groups of workers.

The step across the tasks array that is performed by each worker is not regular, a worker requires two loops to iterate over the whole set of tasks assigned to it, an outer

loop performs the outer-cluster-level iteration while an inner loop performs the inner-cluster-level iteration. From this results that the SRR clustering strategy introduces additional scheduling-overhead when compared to contiguous clustering, since more control variables and conditional statements are required to implement the nested loop.

3.3 On the Affinity between Workers and Cores

Another important aspect to explore when performing hierarchical optimizations pertains to the affinity between workers and processors' cores. This is particularly important when employing the SRR clustering strategy, which assumes that the workers operating over a given task cluster execute on cores sharing a LLC. Therefore, it is important to map the affinity between workers and cores according to the assumptions of the employed clustering strategy. Note that this affinity mapping makes sense mostly on HPC, where the entirety of the machine's resources are available for use by the problem application alone.

For this purpose, we devise a strategy that: enforces the assumptions of SRR clustering, provides workers with some degree of movement between cores and maximizes the exploitation of previously built caches. The strategy assumes that the machine's underlying cache hierarchy is *strictly inclusive*, which is the case in most modern CPU architectures. In strictly inclusive caches, the content of a given cache level L is also contained in every cache level $L' > L$. Exclusive caches, on the other hand, guarantee that data is in at most one of the L1, L2 and L3 caches, never in all at the same time.

To help understand the problems to handle and the rationale behind our strategy, consider the situation depicted in Figure 3.9. We can observe that every worker already has its current workingset in the L1 cache of the core where it is currently executing. If for some reason the operating system reschedules worker w_3 to core c_4, two performance-wise negative effects result: 1) w_3 will experience full cache misses at every cache level, 2) worker w_4's cache in core c_4 will be overwritten and will not be available for w_4 if it resumes execution on the same core.

The aforementioned problems could have been mitigated if worker w_3 had been rescheduled onto a core which shared cache level(s) with the core where it was executing, namely cores 0, 1 and 2 (which are siblings of core 3) since subsequent memory accesses by w_3 would initially be misses on L1, but could possibly be hits on L2, experiencing a faster memory access than w_3 would otherwise experience on core c_4, since data would have to be fetched all the way through the cache hierarchy from the RAM memory. One may note that this option would also overwrite the data on the L1 cache of the core onto which w_3 would be scheduled, say core c_2, trashing the workingset of worker w_3. Although this is true, if we also guarantee that w_3 may only be afterwards scheduled onto cores 0, 1, 2 or 3, worker w_3 will also benefit from the previously mentioned higher cache level hits.

On the other side of the coin, situations may arise where a core is free for a worker w to execute, but w will not be scheduled onto the core because it has no affinity to it.

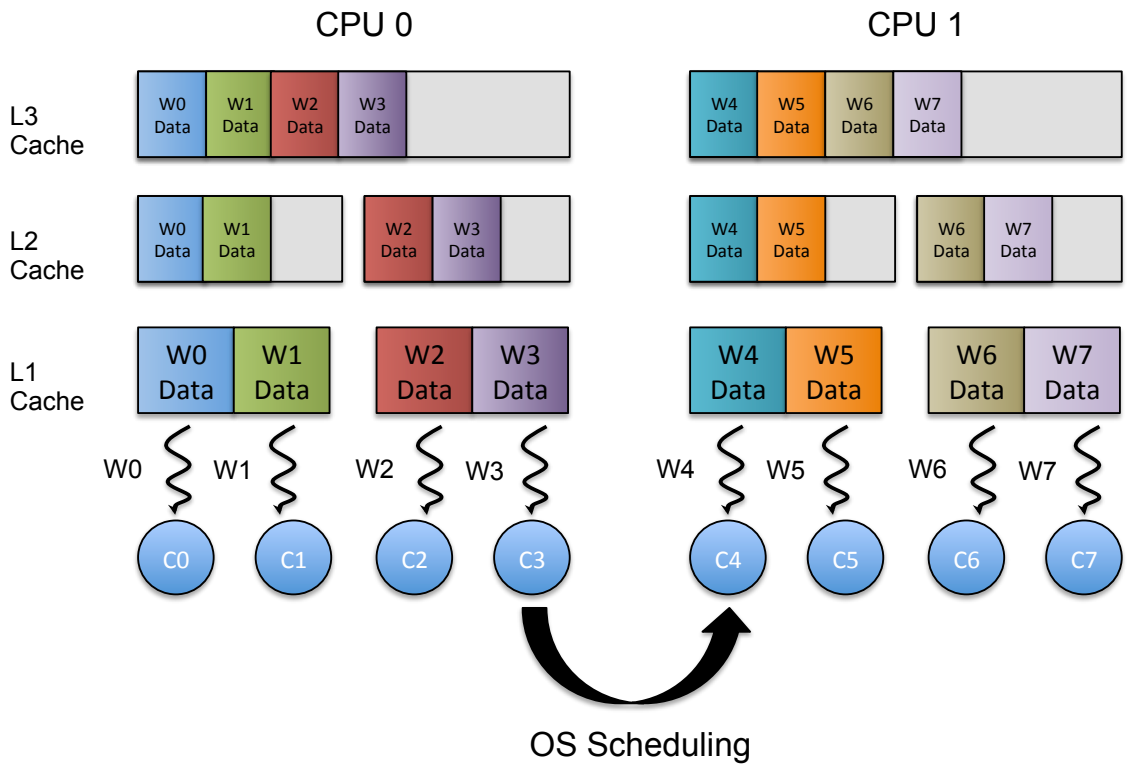

Figure 3.9: Operative System rescheduling a worker

3.3.1 Lowest-Level-Shared-Cache Affinity Mapping

Our affinity mapping strategy builds upon groups of cores sharing cache levels, more specifically, the lowest level shared caches. Given the rationale presented in the previous section, having the affinity of w_3 set to cores $\{0, 1, 2, 3\}$ opens up the possibility that its current workingset still resides at the L2 or L3 cache once w_3 resumes execution. Setting the affinity of w_3 to cores $\{2, 3\}$ will force data to be first searched in the L2 cache, and only then in the L3 cache if it was not found in the L2 cache, providing a possibly faster memory access while bringing w_3's workingset to the L1 cache.

The set of all the machine's cores Q can be obtained by the union of the sets of cores sharing each lowest level shared cache, hence $Q = \bigcup_{i=0}^{n-1} Q_i$ where each set Q_i has cardinality s and represents a set of sibling cores sharing a lowest level shared cache. Given the set of workers $W = \{w_0, ..., w_{m-1}\}$, the set of cores Q' to which a worker w_j has affinity is given by a function α which can be defined as $\alpha(w_j) = Q_{(j/s)}$, where the operator "/" denotes the integer division.

Figure 3.10 depicts this affinity mapping strategy for 8 workers on a 8-core machine, with the L2 cache being the lowest shared cache, shared by two cores each.

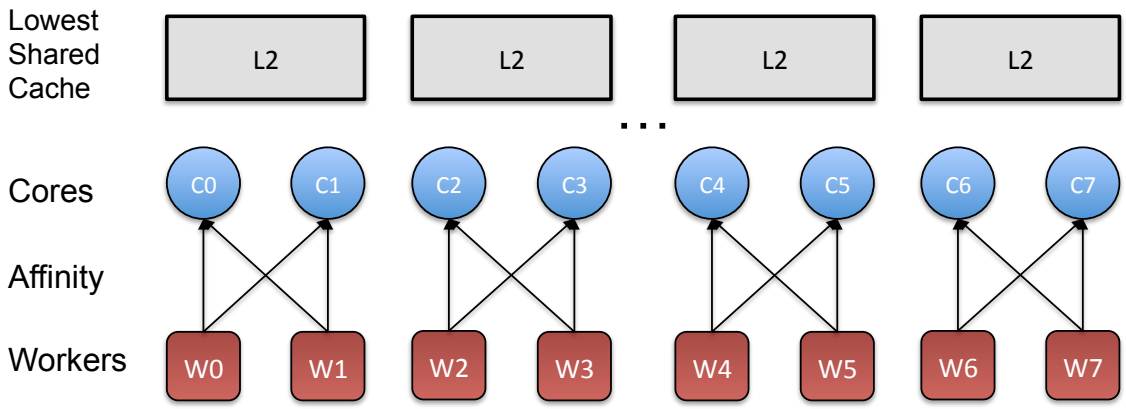

Figure 3.10: Lowest Shared Cache Affinity Mapping

3.4 Concluding Remarks

In this chapter we introduced data-size driven domain decomposition, presenting the challenges involved and introducing the `Distribution` interface that regulates the definition of domain decomposition algorithms, which have to be provided by the programmer. Furthermore, we presented an iterative algorithm that estimates the appropriate workingset size for tasks, based on the TCL of the underlying machine and an estimation function φ whose defition implies a trade-off between performance, accurracy and wasted cache space.

Subsequently, we proposed different scheduling strategies are presented to map a number of parallel tasks greater than the number of CPU cores onto these cores. These scheduling strategies present an additional trade-off between execution overhead and cache hierarchy exploitation.

Finally, we explained the importance of the affinity between worker threads and CPU cores in the hierarchical decomposition context and presented an affinity mapping strategy that attempts to provide a faster memory access in HPC environments.

On the following chapter we will describe how all these proposals have been concretized on top of the Elina Java parallel programming framework.

4

Implementation in the Elina Framework

The implementation of the proposals elaborated in the scope of this dissertation was performed on top of the Elina Framework [SMP12], a middleware designed with the goal of efficiently supporting the execution of Java applications across heterogeneous execution environments, namely heterogeneous clusters and nodes.

The main reason for choosing the Elina Framework as the basis for the implementation of our vertical decomposition strategies pertains to the platform's modularity. Elina's architecture contains a layer of abstraction that allows different modules to be plugged into the middleware on its initialization, adapting its behaviour to the specifies of the application and/or of the underlying target hardware.

The possibility of altering Elina's execution workflow through the insertion of pluggable modules allowed us to experiment different decomposition strategies, without altering the applications' source code.

4.1 The Elina Framework

Application development in Elina is centred on the concept of active objects [LS96] (referred as *services* in Elina), which provide a high-level flexible model for programming parallel systems. Once defined, services are deployed according to a configurable predefined distribution policy across the *nodes* where the computations will take place. It is possible to explicitly compose services at language level, allowing the creation of new services that inherit the interfaces of those composing them.

4.1.1 Parallel Programming in Elina

Elina provides two different ways for programmers to express parallelism. A service can have its interface's methods annotated as *tasks* through the @Task annotation, informing the compiler that the method executes concurrently within a place. Alternatively, programmers may choose to express parallelism through the explicit creation and submission of Elina tasks, via the framework's API. Tasks in Elina may be distinguished into regular and SOMD tasks, being the latter data-parallel tasks whose execution follows the SOMD model to be described in the following subsection.

Task execution in Elina is asynchronous by default. The submission process returns a future object, which can latter be used for data-centric synchronization. The realization of this asynchronous invocation mechanism is delegated to the annotation processor, which instrumentates the method to introduce implicit futures.

4.1.1.1 SOMD: Single Operation Multiple Data

The Single Operation Multiple Data (SOMD) [MP12] is an execution model that applies a MapReduce-like approach to the execution of subroutines (methods in Java terminology). The execution of a method in this context is carried out by multiple parallel execution flows, each operating over a subset of the methods input data. Accordingly, the method's original input is partitioned into several subsets, which are then used as the input for the same number of instances of the method (Method Instances, MIs from now on). Each MI executes the method over its respective input, computing partial results which may afterwards require a *Reduction* operation to produce the result for the initial invocation. Figure 4.1 illustrates the SOMD execution model, from the original method's invocation to the obtainal of the method's result.

A method defined as a task can be provided with additional annotations to employ a SOMD execution model. Elina exposes the SOMD model to the programmer as a three stage process, the description of each stage follows:

1. **Distributing and Combining** - input arguments are partitioned to produce a collection of partitions of the arguments. These partitions are then combined (one from each input argument) to produce MIs' workingsets;

2. **Mapping** - applies the annotated method concurrently over each workingset;

3. **Reducing** - results produced by the mapping stage are reduced to produce the original method invocation's result.

The *Distributing and Combining* and the *Mapping* stages are mandatory when employing a SOMD execution. In turn, the *Reduction stage* is optional, existing only when the reduction of partial results into a single result is required by the problem. The specification of the *Distributing* stage is independent of the *Combining* stage which, unless specified, is assumed to be a 1-1 mapping between the partitions of the different input arguments.

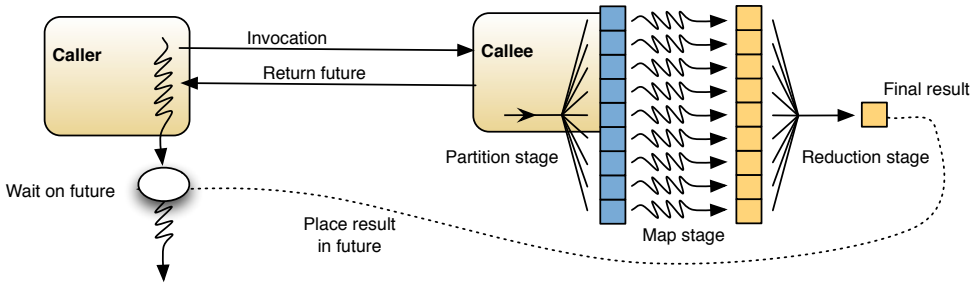

Figure 4.1: The SOMD Execution Model

```
1  @Reduce(MatrixBlockSumReduction)
2  @Combine(class = MatrixBlockCombination, parameters="A,B")
3  int[][] matmult(@Dist(class="Index2DBlockDist") int[][] A, @Dist(class="Index2DBlockDist
       ") int[][] B) {
4    int[][] result = new int[A.length][B[0].length];

6    for(int i=0; i<A.length; i++)
7     for(int j=0; j<B[0].length; j++) {
8       result[i][j] = 0;
9       for(int k=0; k<A[0].length; k++)
10        result[i][j] += A[i][k] * B[k][j];
11    }
12   return result;
13 }
```

Listing 9: Matrix Multiplication Example

When such does not hold, the programmer is required to provide a concrete specification for the *Combining* stage.

Listing 9 presents the SOMD annotated code of the matrix multiplication problem. Note that the code itself contains no references to the parallelism available on the target machine's hardware, let alone the cache hierarchy.

Matrices `A` and `B` are annotated as subject to `Index2DBlockDist` distributions (line 3). The `Index2DBlockDist` distribution partitions the input matrix into equally sized blocks, as depicted in Figure 3.4. The produced partitions of these two matrices are then combined through a `MatrixBlockCombination` combination (line 2), which combines the partitions of matrices *A* and *B* according to pattern depicted in the result matrix of Figure 3.4. Finally, the partial results from every MI are reduced through a `MatrixBlockSumReduction`, which sums the values of partial results associated with a same block in the final result matrix, as illustrated, once more, in the result matrix of Figure 3.4

4.1.2 Runtime System

Elina's system architecture follows a layered model composed of several layers, as depicted in Figure 4.2. Applications execute on top of the Elina platform by requesting services from the *Interface Layer*, which serves as the front door for applications to access Elina's functionalities.

The *Core Layer* holds all the technology independent logic of the middleware, in what

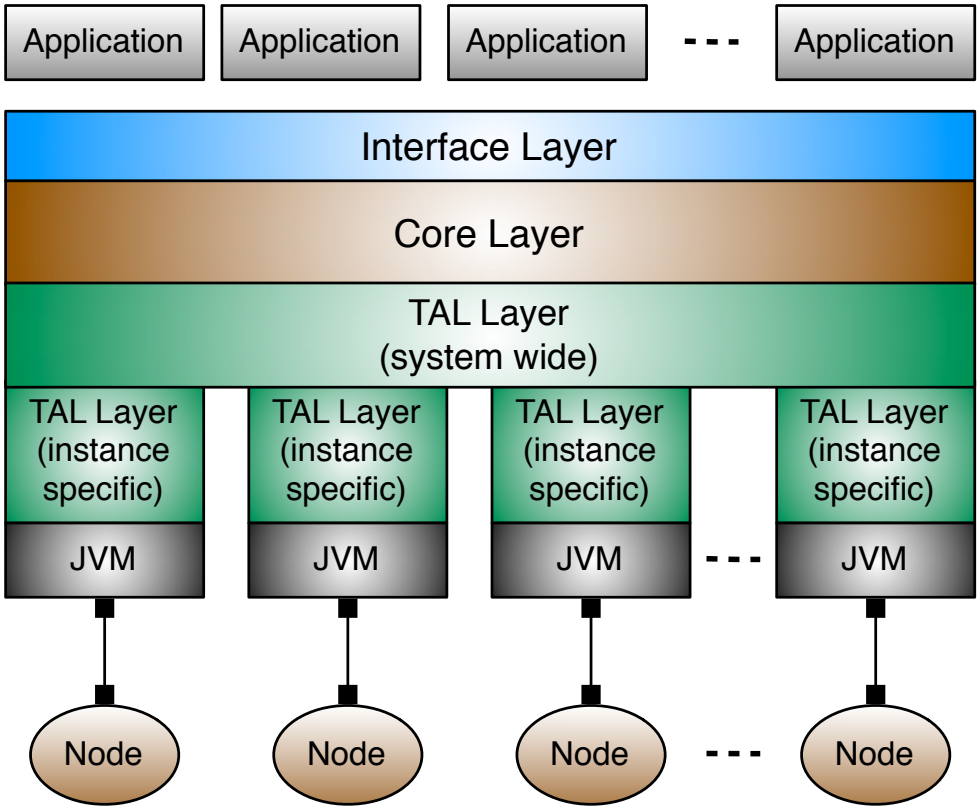

Figure 4.2: The Elina Framework Architecture, taken from [SMP12]

concerns, application deployment, work scheduling, data distribution, communication, among others. The *Technology Abstraction Layer* (TAL) modules are responsible for implementing a set of Java interfaces that determine the agreement between Elina and the technologies to be used. This is an adapter layer that provides the means to adapt the framework according to the target execution environment.

The TAL layer modules can be separated into two major categories: modules that affect the whole system (system-wide), for instance the communication protocol between nodes; and modules with a local system impact (instance-specific), which include the modules for decomposition and affinity mapping, among others.

4.1.3 Elina Initialization

On Elina's deployment, several initialization steps are performed. These include loading the configurable adapters to instantiate the TAL layer, and initializing the modules of the Core layer, which require the subsequent initialization of the TAL layer adapters they require. In the particular context of our work, we highlight the launching of the pool of worker threads and the mapping of the affinity of these accordingly. Figure 4.3 depicts the initialization workflow of Elina's relevant modules in the vertical decomposition context.

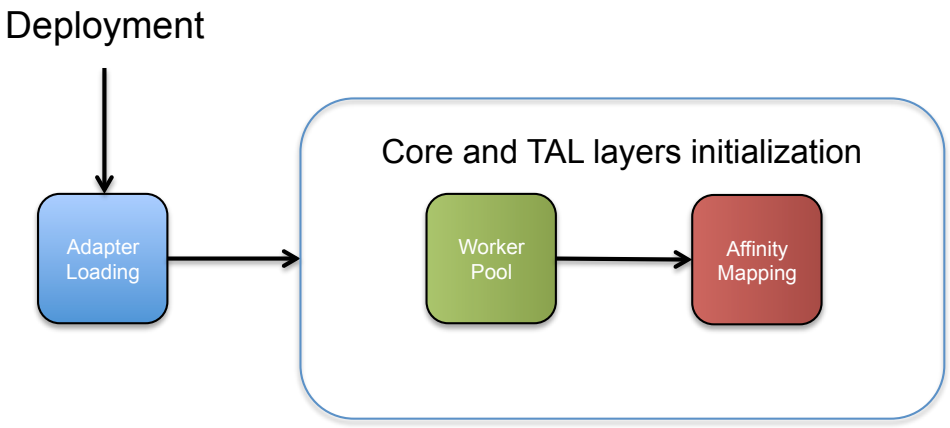

Figure 4.3: Elina Initialization Workflow

4.1.4 Elina Execution Workflow

The parallel execution of Elina tasks comprises several stages, from the actual invocation of the annotated method (or with the explicit submission of the task) to the computation of the final result. These stages are depicted in Figure 4.4.

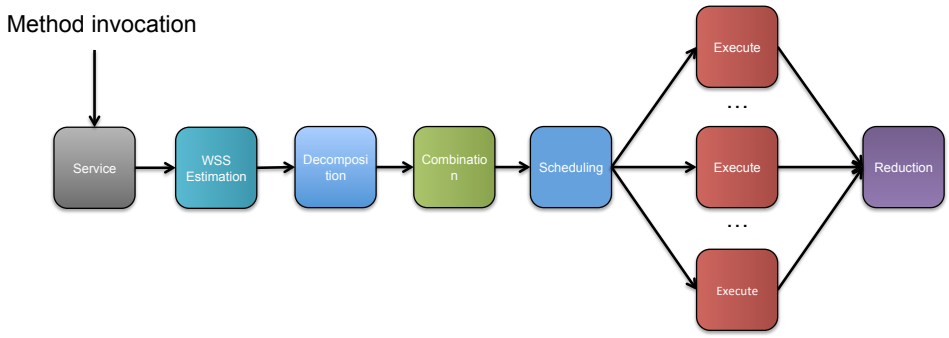

Figure 4.4: Elina Execution Workflow

Once a SOMD annotated method is invoked, a *Workingset Size Estimation* stage determines into how many partitions should the method´s input arguments be decomposed. The *Decomposition* stage then takes place, decomposing each input argument by applying the associated distribution according to the previously determined number of partitions.

The next stage applies the optional *Combination* step to the previous decomposition of the input arguments, to produce the workingsets for parallel tasks to execute. The *Scheduling* stage follows, assigning to workers a number of tasks for execution. All stages up to this point execute sequentially.

Workers then enter a parallel *Execution* stage, where these execute the tasks assigned to them, executing the original problem's operation over each workingset.

Once the result of every execution of the problem's operation over the workingsets is produced, if previously defined, a *Reduction* stage occurs that produces the result of the execution of the problem's operation over the original input arguments.

```
1  interface HierarchyReadDriver {
2
3    /**
4     * Returns a HierarchyLevel object representative of the root of the memory hierarchy.
5     * @return Hierarchy root
6     */
7    HierarchyLevel getHierarchyRoot();
8
9  }
```

Listing 10: The `HierarchyReadDriver` interface

4.2 Vertical Decomposition in Elina

Given the aforementioned modularity of the Elina Framework, most of the work developed during the duration of this thesis consisted on developing new adapters that implement the strategies discussed in Chapter 3.

Most adapters feature a debug counterpart, which contains control code that delimits the execution time of different stages of the Elina workflow. This allowed us to perform the breakdown of the total execution time of an application from the invocation of the SOMD execution until the return of its result. This breakdown is important to help us understand the impact that each stage has in the overall execution time, which helped us understand when the loss of performance was due to sequential stages (decomposition, combining, reduction) taking more time when employing a vertical decomposition.

4.2.1 New Adapter Types

Although some aspects of vertical decomposition were introduced into Elina through the simple implementation of new adapters for existing behaviours, namely the decomposition stage, other behaviours and workflow stages had to be added into the middleware's core. This resulted in the introduction of new adapters into the TAL layer, as well as modifications on the Core layer to support the additional logic involved.

4.2.1.1 Hierarchy Read Adapter

Since our approach revolves around optimizations based on the cache hierarchies, we required a way for the middleware to somehow obtain the information representative of the underlying machine's hierarchy. This necessity led to the definition of a new interface `HierarchyReadDriver`, which is implemented by classes that produce the hierarchy information for the middleware to use. The interface is presented in Listing 10.

We may observe that the method `getHierarchyRoot` in the `HierarchyReadDriver` interface returns an object of type `HierarchyLevel`. This class is used to represent the hierarchy information during runtime, and its relevant fields and methods are presented in Listing 11.

Although we have performed some experimentation on automatic hierarchy inferral,

```
class HierarchyLevel {
  //Fields
  ...
  /**
   * HierarchyLevel object representative of the child level of the current level.
   */
  private HierarchyLevel child;

  //Methods
  /**
   * Returns an array of arrays containing the sibling cores sharing memory on the
       current level.
   * @return Array of arrays of sibling cores in the current level
   */
  public int[][] getSiblings();

  /**
   * Returns the size (in bytes) of the current memory level.
   * @return Size (in bytes)
   */
  public long getSize();

  /**
   * Returns the size (in bytes) of a coherency line in the current memory level.
   * @return Coherency line size (in bytes)
   */
  public int getCacheLineSize();

  /**
   * Returns the child level of the current memory level.
   * @return Child level
   */
  public HierarchyLevel getChildren();

  /**
   * Returns the Nth descendant level of the current memory level.
   * @param N number of the descendant level
   * @return The Nth descendant level
   */
  public HierarchyLevel getLevel(int N);

  /**
   * Returns the bottom-most shared memory level, starting on the current level.
   * @return The Bottom-most shared memory level
   */
  public HierarchyLevel getBottomUpFirstShared();
```

Listing 11: The `HierarchyLevel` class

the support for different operating systems posed problems since no universal tool existed to obtain the hierarchy information. Nevertheless, in Debian environments (which powered the machines where we performed our experimentations) we manually retrieved this information from the files located in the `/sys/devices/system/cpu/cpu*` directories. We ended up implementing an adapter `HierarchyReadImpl` that reads the hierarchy information from a local JSON file. These JSON files are composed of nested objects with the following fields:

- **size** - size of the memory level (in bytes)
- **cacheLineSize** - size of a cache coherency line (in bytes). This field is present only if the memory level represents a cache level

```
{
    "siblings": [[0,2,4,6],[1,3,5,7]],
    "size": 6291456,
    "cacheLineSize": 64,
    "child": {
      "siblings": [[0],[1],[2],[3],[4],[5],[6],[7]],
        "size": 524288,
        "cacheLineSize": 64,
        "child": {
            "siblings": [[0],[1],[2],[3],[4],[5],[6],[7]],
            "size": 65536,
            "cacheLineSize": 64,
            "child": null
        }
    }
}
```

Listing 12: 8-core Machine Hierarchy Representation

- **siblings** - array of arrays of sibling cores sharing each copy of the memory level
- **child** - object containing the memory level information of the child level (can be null if the current level is the bottom-most in the hierarchy)

Concrete examples of these JSON files are presented in Listings 12 and 13, which model, respectively: a 8-core machine with dedicated L1 and L2 caches and two L3 caches shared by 4 cores each; and a 64-core machine with dedicated L1 caches, thirty two L2 caches shared by 2 cores each and eight L3 caches shared by 8 cores each. Note once more that the definition of the `HierarchyReadDriver` interface allows for multiple definitions of the adapter. The implementation of the `getHierarchyRoot` method can resort to any means to build the `HierarchyLevel` object, including JNI calls to tools that retrieve CPU information.

4.2.1.2 Workingset Size estimation Driver

Chapter 3 presented the problem of data-size driven decomposition, which we aimed to perform for a data size dependant on the target machine's TCL size. The original adapter suite of Elina contained only horizontal decomposition adapters, which decomposed the applications' input arguments into as many partitions as the number of available workers.

In the vertical decomposition context, it is necessary to determine the number of partitions into which each input argument should be split so that the cummulative size of a partition from each argument fits the TCL. The algorithm that determines this number of partitions was presented in Chapter 3, more concretly in Algorithm 1. To this end, we defined a new adapter type, the *Workingset Size Estimation* adapter and the associated interface `WSEstimationDriver`, presented in Listing 14.

We defined a simple realization of this interface that implements Algorithm 1, the *IterativeWSEstimator* class.

```
{
    "siblings": [[0,1,2,3,4,5,6,7],[8,9,10,11,12,13,14,15],
            [16,17,18,19,20,21,22,23],[24,25,26,27,28,29,30,31],
            [32,33,34,35,36,37,38,39],[40,41,42,43,44,45,46,47],
            [48,49,50,51,52,53,54,55],[56,57,58,59,60,61,62,63]],
    "size": 6291456,
    "cacheLineSize": 64,
    "child": {
      "siblings": [[0,1],[2,3],[4,5],[6,7],[8,9],[10,11],[12,13],[14,15],
        [16,17],[18,19],[20,21],[22,23],[24,25],[26,27],[28,29],
        [30,31],[32,33],[34,35],[36,37],[38,39],[40,41],[42,43],
        [44,45],[46,47],[48,49],[50,51],[52,53],[54,55],[56,57],
        [58,59],[60,61],[62,63]],
        "size": 2097152,
        "cacheLineSize": 64,
        "child": {
            "siblings":
                [[0],[1],[2],[3],[4],[5],[6],[7],[8],[9],[10],[11],[12],[13],[14],[15],
              [16],[17],[18],[19],[20],[21],[22],[23],[24],[25],[26],[27],[28],[29],
              [30],[31],[32],[33],[34],[35],[36],[37],[38],[39],[40],[41],[42],[43],
              [44],[45],[46],[47],[48],[49],[50],[51],[52],[53],[54],[55],[56],[57],
              [58],[59],[60],[61],[62],[63]],
            "size": 16384,
            "cacheLineSize": 64,
            "child": null
        }
    }
}
```

Listing 13: 64-core Machine Hierarchy Representation

```
interface WSEstimationDriver {

  /**
   * Returns the appropriate number of partitions for the provided arguments.
   * @return the number of partitions
   */
  int getNparts(Distribution<?>[] distributions, HierarchyLevel hierarchy, SOMDTask<?>
      task) throws IllegalArgumentException;

  /**
   * Returns the TCL size used during the algorithm's decisions.
   * @return the TCL size used
   */
  int getTCLSize();

}
```

Listing 14: The `WSEstimationDriver` interface

```
1  interface AffinityMappingDriver {
2
3    /**
4     * Sets the desired affinities for the worker pool managed by taskManager.
5     * @param taskManager TaskExecutorDriver object responsible for the worker pool
6     */
7    void setAffinities(TaskExecutorDriver taskManager);
8
9  }
```

Listing 15: The `AffinityMappingDriver` interface

4.2.1.3 Affinity Mapping Driver

Elina's initial version performed only horizontal domain decomposition and performed no hierarchy-aware optimizations. In the presence of horizontal decomposition, the affinity between workers and CPU cores is not as relevant as in vertical decomposition, since workingsets likely exceed the cache size by several orders of magnitude. Consequently, workers are expected to overlap each other's space in shared caches, causing the mapping to provide minimal benefits or no benefits at all.

Since we are studying vertical decomposition, the affinity between workers and CPU cores takes a major toll, and, hence, we wanted to study its real impact in our approach. To introduce this feature in Elina we defined the `AffinityMappingDriver` interface, which regulates the implementation of adapters responsible for mapping worker pools onto the underlying machine's CPU cores. The interface is presented in Listing 15.

Two realizations of the `AffinityMappingDriver` were implemented. The first one imposes no restrictions on the affinity (`AllAffinityMapper`), and exists only to provide a `AffinityMappingDriver` for horizontal decomposition. The second employs the strategy discussed in Subsection 3.3.1 (`JStackParsingAffinityMapper`), resorting to the external commands `jstack` (from which its name derives) and `taskset` to, respectively, obtain the workers' correspondence to OS threads and set the affinity of these according to the underlying machine's hierarchy. A more detailed description of these two commands follows:

- **jstack** `[option] pid` - prints the Java stack trace of the Java threads associated with the Java process with pid `pid`. The trace includes the mapping between Java threads and OS threads.

- **taskset** `[options] -p [mask] pid` - sets the affinity of the process/thread identified by `pid` to the set of cores represented by the bitmask `mask`, where the n_{th} bit (counted from the right to the left) being at 1 enables affinity to the core $n - 1$. The bitmask is provided in hexadecimal.

Note that while the `jstack` tool is cross-platform, the `taskset` command exists only in (most) Linux distributions, hence this adapter can only be used in such systems.

We have also implemented the debug counterparts of these adapters, respectively, the `DebugAllAffinityMapper` and `DebugJStackParsingAffinityMapper`.

```
1  interface DomainDecompositionDriver {
2
3    /**
4     * Decomposes the domains handled by the distributions stored in distr.
5     * @param distr array holding the distributions
6     * @param combin combination to be applied to the partitions returned by the
          distributions
7     * @param task task that will be executed
8     * @param nWorkers number of workers that will execute the task
9     * @return Array of workingsets
10    */
11   <R> Object[][] decompose(Distribution<?>[] distr, Combination<?> combin, SOMDTask<R>
         task, int nWorkers);
12
13 }
```

Listing 16: The `DomainDecompositionDriver` interface

4.2.2 New Adapter Implementations

The *Decomposition* and *Scheduling* stages were originally merged together in a single execution stage, which performed both steps using a single adapter. This was reasonable in the horizontal decomposition context since each worker computed a single task over a single workingset.

In this vertical decomposition context we felt the need to have these two stages dissociated. This dissociation would allow us to employ different scheduling strategies when assigning the workingsets produced during the decomposition stage to the existing workers. These strategies correspond to the *CC Clustering* and the *SRR Clustering* strategies presented in Chapter 3.

This dissociation resulted in the definition of two new adapters types, responsible for the *Decomposition* and *Scheduling* stages. Nevertheless, these two stages already existed in some way in Elina, hence we decided to consider these as new implementations for existing adapter types.

4.2.2.1 Domain Decomposition Driver

The interface representative of the decomposition adapters, `DomainDecompositionDriver`, is presented in Listing 16. A single method `decompose` exists in the interface. The distributions representative of the datasets to decompose are passed as arguments to the method, along with the `SOMDTask` to execute and the number *nWorkers* of workers that will perform the computation. The `SOMDTask` is required by the decomposition in order to deal with dynamic data allocation, which will be presented in Subsection 4.2.3. Although the number of available workers is globally accessible in the middleware, one may be interested in using only a subset of the pool of workers when employing a given computation, hence the number of workers to use is supplied as a parameter.

Two decomposition adapters were implemented, one that performs horizontal decomposition (`HorizontalDomainDecomposer`), and one that employs a vertical decomposition (`VerticalDomainDecomposer`). The implementation of these two adapters is almost

```
1  interface SchedulingDriver<R> {
2
3    /**
4     * Schedules the workingsets for the task to be executed among the
5     * workers available on the middleware's worker pool.
6     * @param task task to be executed
7     * @param partitions partitions of the different input arguments
8     * @param red reduction to be applied to the partial results produced
9     * @param nWorkers number of workers that will execute
10    * @return Future for access to the final result once it is produced
11    */
12   IFuture<R> schedule(SOMDTask<R> task, Object[][] workingsets,
13       Reduction<R> red, int nWorkers);
14
15 }
```

Listing 17: The SchedulingDriver interface

identical, the only difference pertains to the determination of the number of partitions *nParts* to apply to the provided distributions. The HorizontalDomainDecomposer simply uses the value of nWorkers, whilst the VerticalDomainDecomposer resorts to the configured WSEstimationDriver to determine the appropriate value of $nParts$.

The respective breakdown variants of these adapters were also implemented, namely the DebugHorizontalDomainDecomposer and the DebugVerticalDomainDecomposer adapters.

4.2.2.2 Scheduling Driver

The interface representative of the scheduling adapters, SchedulingDriver, is presented in Listing 17. In the adapters we devised, the method schedule launches a single task per worker, which is responsible for distributing the workingsets according to the clustering strategies presented in Section 3.2. The arguments passed to the method include the SOMDTask to execute, the workingsets that will be used as the input for independant executions of the task, the reduction to employ, the array where partial results will be placed by workers, and the number of workers that will participate in the computation.

Two scheduling adapters were implemented, one that employs the contiguous clustering strategy (ContiguousScheduler), and one that employs the sibling round-robin clustering strategy (SiblingRoundRobinScheduler). The respective breakdown variants of these adapters were also implemented, namely the DebugContiguousScheduler and the DebugSiblingRoundRobinScheduler adapters.

4.2.3 Supporting Dynamic Memory Allocation

The programming example presented in Listing 9 has no mention to the memory occupied by the matrix result, which is allocated during the task's execution. This poses a problem in the vertical decomposition context, since the estimation computed by the WSEstimationDriver will assume that only the partitions of A and B will fill the TCL.

We have solved this problem, but not at the language level. We defined a new distribution `DynamicIndex2DBlockDist` that determines the amount of memory (in bytes) allocated internally by the task, given a number of partitions `nParts` and the dimensions of the matrices `A` and `B`. This distribution does not produce actual partitions of any domain, serving solely as a mean for the `WSEEstimationDriver` to take into account the amount of data allocated dynamically by the task, during the estimation of the workingset size.

Although we devised a strategy to account for dynamically allocated memory, during the evaluation process of vertical decomposition we faced another problem. Due to the high amount of tasks and the small granularity of these, workers spent most of their time prompting the Java memory manager, in order to allocate the `result` matrix required by each individual task. This led to a performance loss, due to the sequential bottleneck imposed by the memory manager. This was particularly noticeable when we were experimenting on the 64-core machine, where 43 was the observable ceiling for the active thread acount.

To cope with this problem, we decided to preallocate the `result` matrices, treating these as input arguments and combining them with the partitions of the other input arguments. This required the definition of even another distribution `2DBlockAllocator`, one that resorts to the `DynamicIndex2DBlockDist` distribution to both determine the size and produce the partitions corresponding to the result matrices of each individual task.

Internally, the `2DBlockAllocator` distribution performs a simulation to determine the number of partitions that would be generated by the `WSEEstimationDriver`, according to the previous strategy. Once this number of partitions is determined, the distribution resorts to the distributions of `A` and `B` and the combination `MatrixBlockCombination` to obtain the final pairings of partitions of `A` and `B`, in order to preallocate the respective `result` matrix for each of these pairings.

This allowed us to assess the performance of vertical decomposition without the bottleneck imposed by the memory manager. Alternatively, we could have used a Java Virtual Machine with per thread memory management.

Once again, no annotation support has yet been implemented to express pre-allocations, however a possible solution could be to introduce a new annotation `@Alloc` which has two parameters: the class that performs the allocation (`class`) and the class that determines the size of the allocated data (`dynamicDataSize`), depending on the number of partitions into which `A` and `B` are partitioned. Since the size of the allocated matrix depends on the partitions created by the distributions of `A` and `B`, these have to be provided to the `dynamicDataSize` class. The compiler would then instrumentate the code to dynamically allocate the matrix `C` at the beginning of the task's execution. The previously presented matrix multiplication example would then be implemented as shown in Listing 18.

```
1  @Reduce(MatrixBlockSumReduction)
2  @Combine(class = MatrixBlockCombination, parameters="A,B")
3  int[][] matmult(@Dist(class="Index2DBlockDist") int[][] A, @Dist(class="Index2DBlockDist
       ") int[][] B, @Alloc(class="2DBlockAllocator", dynamicDataSize="
       DynamicIndex2DBlockDist", params=A,B) int[][] C) {

5    for(int i=0; i<A.length; i++)
6     for(int j=0; j<B[0].length; j++) {
7       C[i][j] = 0;
8       for(int k=0; k<A[0].length; k++)
9         C[i][j] += A[i][k] * B[k][j];
10    }
11   return C;
12 }
```

Listing 18: Matrix Multiplication Example (Dynamic Memory Allocation Support)

4.2.4 Discussion

Elina proved itself as a useful framework for experimentation on data parallel computations. The framework's modularity was of major importance to our work, since it allowed us to experiment different execution configurations without altering a single line in most applications' source code. The exception was the matrix multiplication problem, which required major source code modifications to accomodate dynamically allocated memory, as previously presented and discussed.

The dissociation of the *Decomposition* and *Scheduling* stages, performed during the period of this dissertation, contributed greatly to the framework's modularity and ease of modification when experimenting different distribution and scheduling techniques.

Although we wanted all our adapters to be cross-platform, this goal could not be achieved due to the lack of cross-platform means to: obtain machine hierarchy information, obtain the correspondence between Java threads and OS threads, map the affinity between threads and CPU cores.

The next chapter will present a performance evaluation of this prototype implementation, with special focus on the gains obtained by our vertical decomposition strategies when compared to the previously existing horizontal approach.

5

Experimental Evaluation

In this chapter we present the evaluation we performed of our vertical decomposition approach, implemented on the Elina framework. We will present the methology we employed for the evaluation, the benchmarks we used to assess the performance of vertical decomposition against horizontal decomposition, the test infrastructure that we used for the evaluation and finally, the experimental results we obtained.

5.1 Methodology

Our evaluation encompassed several different aspects, which correspond to the multiple aspects covered by our automated approach:

1. Efficiency
2. Performance portability
3. Ideal TCL size
4. Impact of the different clustering strategies

The evaluated benchmarks can be separated into two major categories: the ones that feature both spatial and temporal locality, in which we expect performance gains resulting from vertical decomposition; and benchmarks featuring only spatial locality, where vertical decomposition is expected to introduce overhead only. These benchmarks will be presented with further details in Section 5.2

The efficiency of our approach was assessed through a comparative performance analysis against the pre-existing horizontal work distribution featured in Elina. For each

benchmark (and respective classes) and runtime system configuration, we present the speedup of the execution time of the vertical decomposition execution of the benchmark, against the execution time of the horizontal decomposition execution of the same. The execution time for both approaches is the average execution time of 50 runs. Elina's portable programming model and runtime system allowed us to use the same source code in both settings. We simply deployed the runtime system with different instances of several modules.

To assess performance's portability, we performed our evaluation across different machines, featuring distinct cache hierarchies and a different number of CPU cores. This permitted us to evaluate the aptness of our runtime system for differing architectures. These machines will be presented in-depth in Section 5.3.

In order to determine the ideal TCL size for the computations and the appropriate clustering strategy, the benchmarks were executed with different runtime system configurations. These configurations are characterized by the clustering strategy (Contiguous Clustering or SRR Clustering) and the TCL size, which ranges from the size of the L1 cache to the size of the L3 cache on each machine.

5.2 Benchmarks

Matrix multiplication (MatMult), of the form $C = A \times B$, features temporal locality because the lines of A and the columns of B are iterated several times during the algorithm's execution, hence having all the lines and columns of A and B in the cache avoids having to re-fetch lines and columns that have previously been in the cache but had to be removed to accommodate new data, hence improving the cache hit-ratio and consequently reducing the algorithm's execution time.

Matrix transpose (MatTrans) benefits from data locality due to the algorithm's iteration sequence across the result matrix A^T and the granularity of data fetch from main memory into the cache; if A is iterated across its lines, A^T is forcibly iterated across its columns. Thus, when accessing a position in a column of A^T, a cache line will be brought into the cache and its remaining data will not be used to produce cache hits, unless the iteration of the current line of A stalls and proceeds to the next, before space is made for the new cache lines fetched along the iteration of the current column of A^T. By dividing the original transposal into smaller transposals whose working set size fits the cache, the aforementioned stall and transition to the next line occurs, thereby exploiting cache locality.

The problem sizes for MatMult and MatTrans are defined by single parameter that represents the side length of the matrices involved (only square matrices are considered).

Gaussian Blur (GaussianBlur) blurs an image (represented as a matrix) by convolving the image with a Gaussian Function. The benchmark receives three parameters: an image M, an integer R (blur window radius) and a scalar σ. Since σ is used only to produce the weights matrix of the Gaussian Function, which we precompute before the parallel

execution of the algorithm, we use the same value $\sigma = 1.5$ for all problem sizes. We represent the problem sizes for this benchmark in the form S-R, where S represents the side length of the image matrix and R is the aforementioned radius parameter. This benchmark is a Stencil, therefore featuring temporal locality throughout its execution.

SAXPY and JavaGrande-Series are benchmarks that iterate sequentially over data without revisiting previously accessed data, thereby not benefiting from temporal locality. SAXPY receives three arguments: arrays x and y, scalar α and computes for each same position $y = \alpha \times x + y$. Similarly, JavaGrande-Series computes the first N Fourier coefficients of the function $f(x) = (x + 1)^x$ on the interval [0,2].

5.3 Test Infrastructure

All measurements were performed on three shared-memory systems running the Elina framework. The label and specifications of these systems follows:

- **System 1 (S1)** - [8-core node (8 hardware threads)] 2 Quad-Core AMD Opteron Processor 2376 with three cache levels: a 64KBytes L1 data cache per core, a unified 512KBytes L2 cache per core, and a unified 6MBytes L3 cache per processor
- **System 2 (S2)** - [4-core node (8 hardware threads)] 2 Dual-Core Intel(R) Xeon(R) CPU X3450 hyperthreaded with three cache levels: a 32KBytes L1 data cache per core (2 hardware threads), a unified 256KBytes L2 cache per core (2 hardware threads), and a unified 8MBytes L3 cache
- **System 3 (S3)** - [64-core node (64 hardware threads)] 4 16-Core AMD Opteron Processor 6272 with three cache levels: 16KBytes L1 data cache per core; a unified 2MBytes L2 cache per two cores, and one unified 6MBytes L3 cache per eight cores

Systems 1 and 2 are powered by Debian with Linux kernel version 2.6.26-2- amd64. System 3 is powered by Debian with Linux kernel version 3.2.0-0.bpo.4-amd64. The installed Java platform is OpenJDK 7 (version 1.7.0 21).

Systems S1 and S2 are both 8-core machines, but feature different sharing configurations at every cache level. The L1 and L2 cache levels are shared between two logic CPU cores in S2 due to hyperthreading, whilst in S1 both levels are dedicated to a single CPU core. Additionally, while a single L3 cache exists for the whole S1, two distinct L3 caches exist for each processor in S2, shared by 4 cores each.

The configuration of system S3 is the most unique among the three systems. This machine features 64 cores, which contrasts with the 8 cores (real or virtual) of the other machines. Additionally, diversity of cache sharing by the cores exists among levels.

5.4 Vertical vs Horizontal Decomposition

An important aspect that we had in account during our evaluation of vertical decomposition against horizontal decomposition, pertains to the elimination of uneven workload

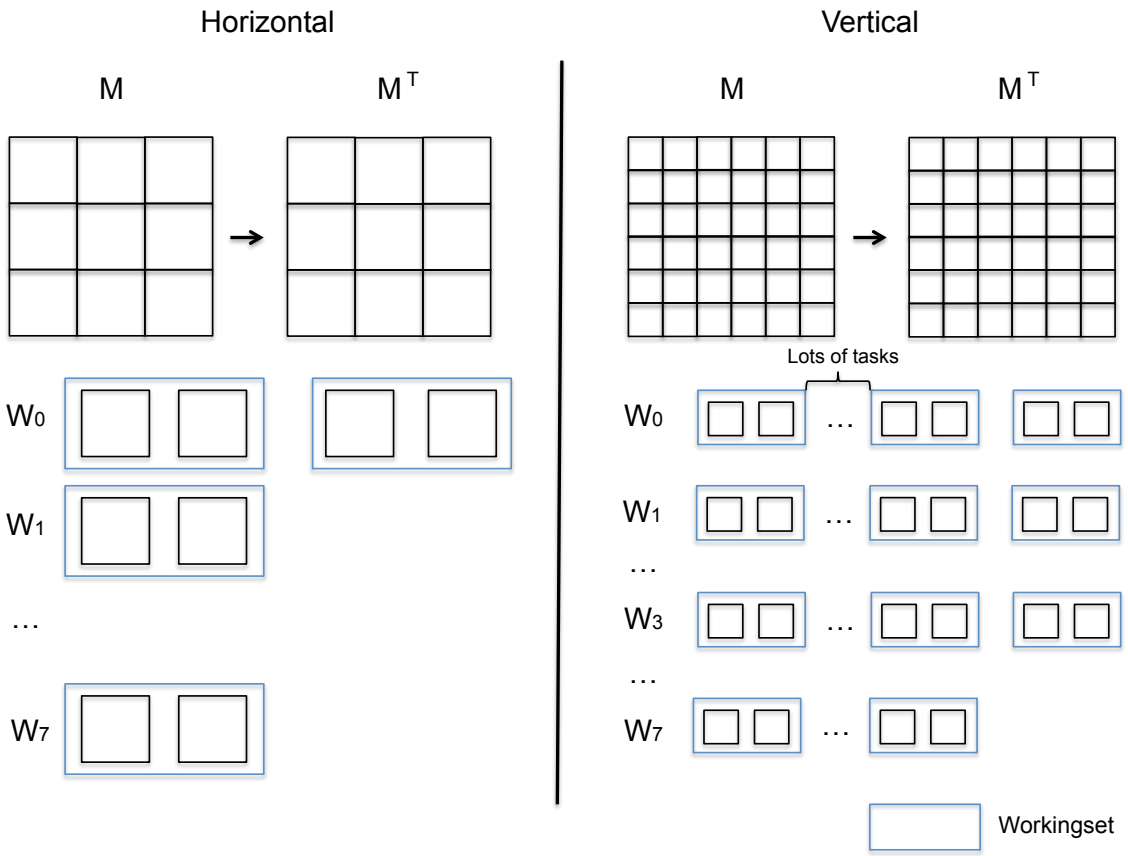

Figure 5.1: Horizontal vs Vertical decomposition: workingset granularity

situations. To understand the problem and the rationale behind the solution, consider the horizontal and vertical block decomposition of the matrix transpose problem, depicted in Figure 5.1. Let's further assume that 8 workers are employed to perform the computation. The horizontal decomposition approach would divide `M` and $\texttt{M}^T$ into 9 blocks, which is the first integer greater than 8 whose square root is an integer. Hence, each workingset would be composed of 1 block of `M` and 1 block of $\texttt{M}^T$, resulting in 9 workingsets. Scheduling these 9 workingsets across the 8 workers would result in the depicted situation, where worker w_0 has one more task for execution than the other workers.

Now consider the vertical decomposition approach, where both matrices will be divided into a number of blocks n such that: a block of `M` plus a $\texttt{M}^T$ fit the TCL, the square root of n is an integer. For the sake of presentability, we present both matrices partitioned into 36 blocks, though the number of blocks is expected to be a lot bigger. In the presented scheduling, workers w_0 to w_3 receive one more workingset than the remaining workers, similarly to the horizontal decomposition situation.

Although both situations have some workers have more workingsets for tasks to operate upon than the others, the granularity of the additional workingset is a lot smaller in the vertical decomposition context than in the horizontal one. This may cause the execution time of the horizontal approach to take at least twice the time to execute than it

Benchmark	Class	$S1$	$S2$	$S3$
MatMult	1000	16.9884	7.9839	–
	1500	89.794	37.0932	–
	2000	233.0346	105.4666	211.8696
	3000	–	–	820.2287
	4000	–	–	1893.1196
MatTrans	3500	0.4624	0.1905	–
	5000	1.1473	0.4417	0.7591
	10000	5.8783	2.5946	4.4231
	20000	–	–	25.1872
GaussianBlur	1000-15	8.9130	3.3650	3.8498
	1000-20	15.4344	5.9149	6.7924
	1000-25	23.7354	8.9523	10.2454
SAXPY	1000000	0.0027	0.0013	0.0024
	10000000	0.0244	0.0129	0.0213
	100000000	0.2493	0.1157	0.1388
Series	10000	9.9905	5.4257	11.9705
	100000	102.1343	56.6607	120.5451
	1000000	1664.5894	889.6919	1843.2300

Table 5.1: Benchmarks sequential execution time (seconds)

would had the number of blocks been a multiple of the number of workers. The vertical approach is less prone to these effects due to the small granularity of the workingsets.

To avoid these situations, which could lead us to wrong conclusions regarding the performance of horizontal decomposition, we implemented new distributions that always partition the problem's domain into a number of partitions multiple of the number of workers. These distributions are used only by horizontal decomposition specific versions of the benchmarks.

Table 5.1 presents the sequential execution time for the evaluated benchmarks on all systems. These values are useful as a reference for the following subsections, which present the speedup results for the decomposition strategies relative to one another.

5.4.1 Matmult, Transpose, Gaussian Blur

Figures 5.2 through 5.18 present the results for the MatMult, MatTrans and GaussianBlur benchmarks. Our initial intuition was that for problems benefiting from data locality, the optimal size for a workingset would be the size of the L1 data cache. However, cache hierarchies such as the one featured in S3 have an enormous discrepancy between the size of the L1 caches and the remaining cache levels. This drove us to experiment TCL sizes that do not correspond to the size of any real cache level. To this end we ranged the considered TCL size from the size of the L1 cache to the size of the L3 cache, on each machine. Note that the speedup values for GaussianBlur with TCL sizes smaller than the size of a blur window are not present in the charts, since a blur window is the minimal workingset size for an individual blur computation.

		S1		S2		S3	
Benchmark	Class	CC	SRRC	CC	SRRC	CC	SRRC
MatMult	1000	128k	128k	64k	64k	-	-
	1500	L1	128k	64k	64k	-	-
	2000	128k	L1	64k	64k	256k	256k
	3000	-	-	-	-	128k	128k
	4000	-	-	-	-	128k	128k
MatTrans	3500	256k	192k	128k	64k	-	-
	5000	192k	192k	64k	128k	32k	128k
	10000	256k	192k	64k	128k	192k	256k
	20000	-	-	-	-	256k	192k
GaussianBlur	1000-15	192k	128k	192k	192k	128k	128k
	1000-20	192k	L1	192k	192k	L2	L2
	1000-25	128k	L1	L2	128k	192k	192k

Table 5.2: Best performance TCL size configurations

The results are very insightful, as the optimal TCL size lies somewhere between the size of the L1 and L2 caches. This optimal TCL size however, varies according to the benchmark and respective classes. Furthermore, this also varies depending on the employed clustering strategy. Table 5.2 presents the summary of the best TCL size for each benchmark and respective classes, on each system, for each clustering strategy.

As a first remark in the analysis of the speedup results, we may observe that for the majority of the benchmarks and respective problem sizes, there is at least one TCL configuration for each system such that employing vertical decomposition provides a speedup greater than 1 against horizontal decomposition. The exception to this is the GaussianBlur benchmark on S3 with parameters 1000-15, where individual tasks operate over a small amount of data and are distributed amongst a large pool of threads. In these conditions, any performance gains attained during the execution (which is short in duration) are easily suppressed by the overhead that results from vertical decomposition and task scheduling.

The major speedups occur for MatMult in systems S1 and S2, where the speedups reach the 700% and 600% mark respectively. In S3 the speedups are not so significant, peaking slightly above 300% (which is good nonetheless). These speedups represent major performance boosts, which were attained through hierarchy-based optimization rather than extra parallelization.

The lower magnitude of the speedups in S3 are not exclusive to the MatMult benchmark. The speedups of the MatTrans benchmark reach the 600% and 500% mark in systems S1 and S2 respectively, but reach only the 140-160% mark in S3. The explanation that we have found for this behaviour pertains to the characteristics of S3's hardware: the L2 and L3 caches are considerably large, thereby reducing the overall number of LLC misses; also, the parallelism available on S3 is much higher than on S1 and S2, which reduces the computational weight assigned to each worker.

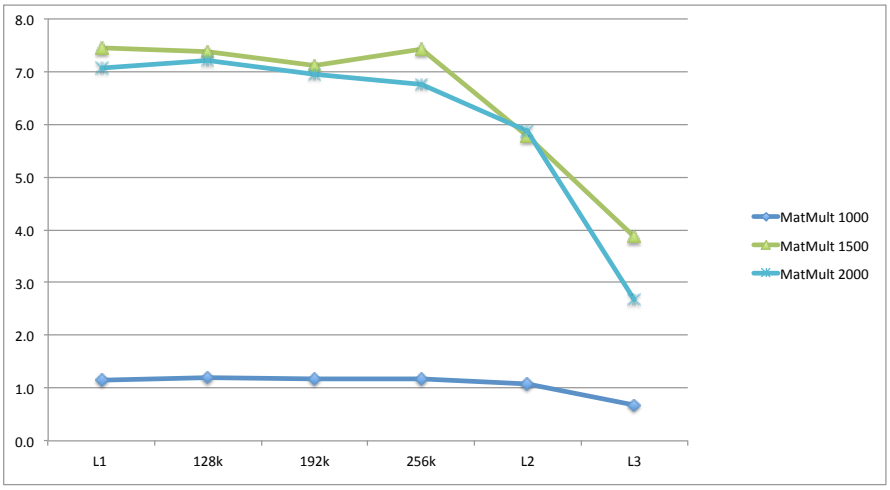

Figure 5.2: S1 Speedups: MatMult (Contiguous Clustering)

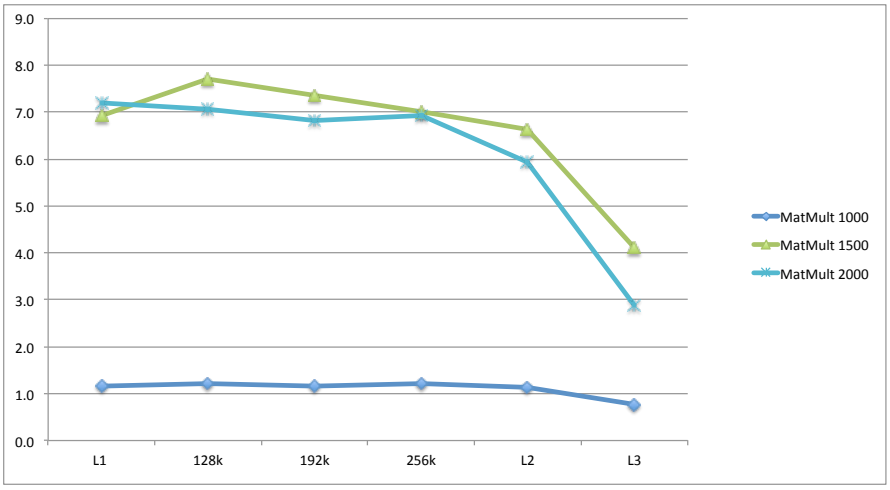

Figure 5.3: S1 Speedups: MatMult (SRR Clustering)

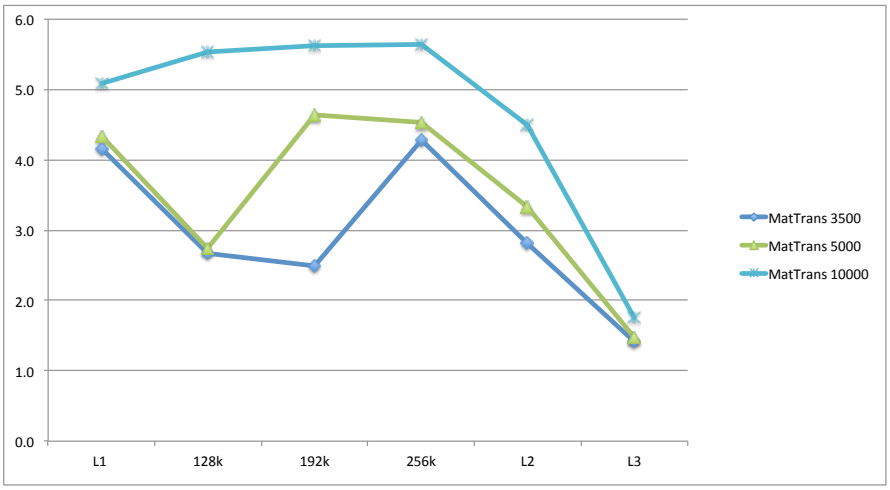

Figure 5.4: S1 Speedups: MatTrans (Contiguous Clustering)

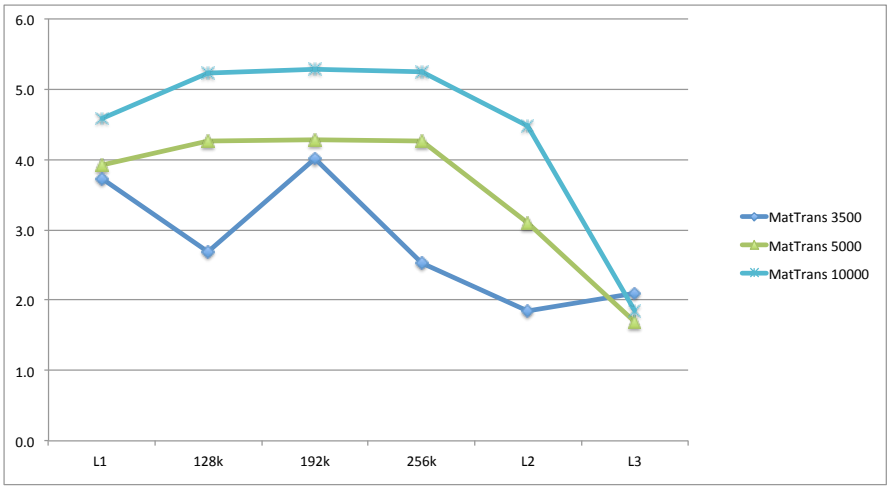

Figure 5.5: S1 Speedups: MatTrans (SRR Clustering)

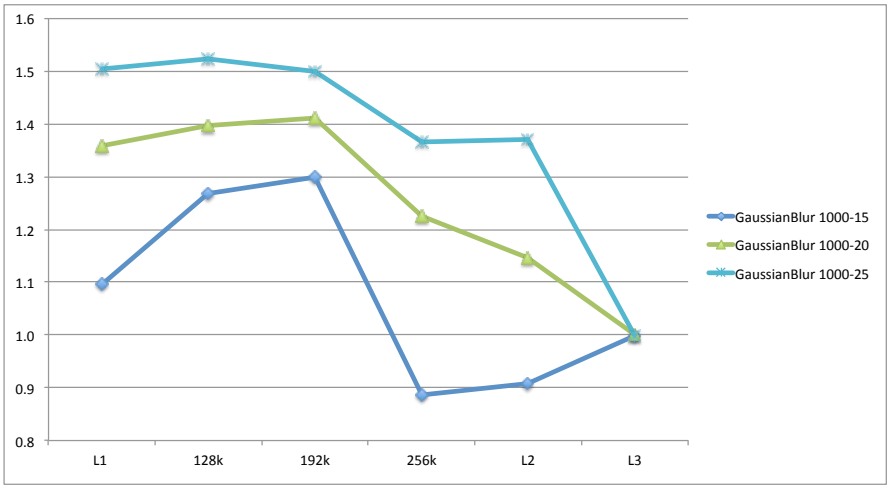

Figure 5.6: S1 Speedups: GaussianBlur (Contiguous Clustering)

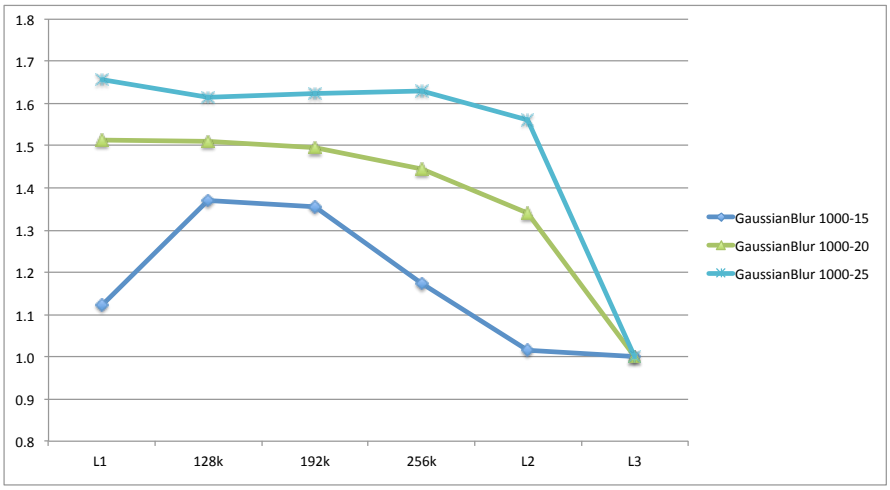

Figure 5.7: S1 Speedups: GaussianBlur (SRR Clustering)

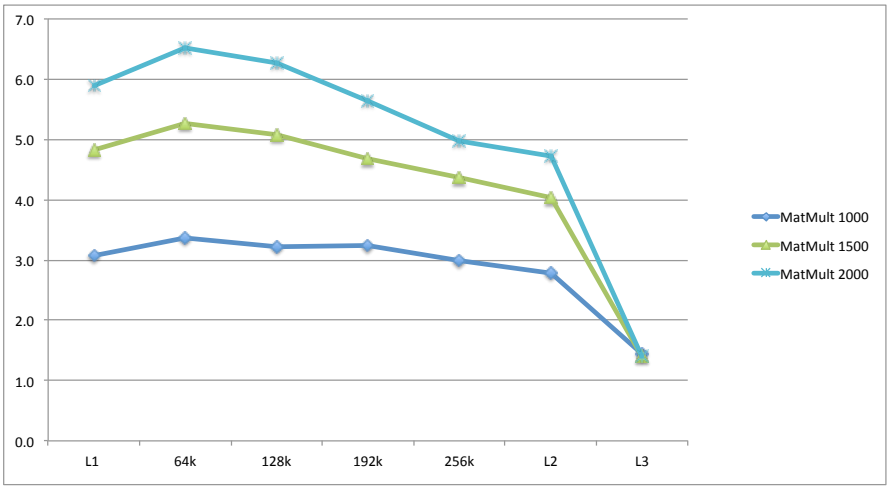

Figure 5.8: S2 Speedups: MatMult (Contiguous Clustering)

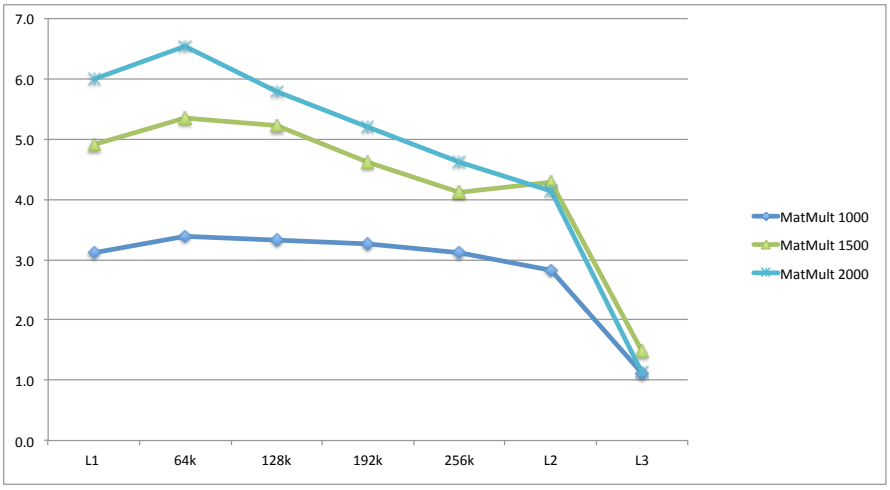

Figure 5.9: S2 Speedups: MatMult (SRR Clustering)

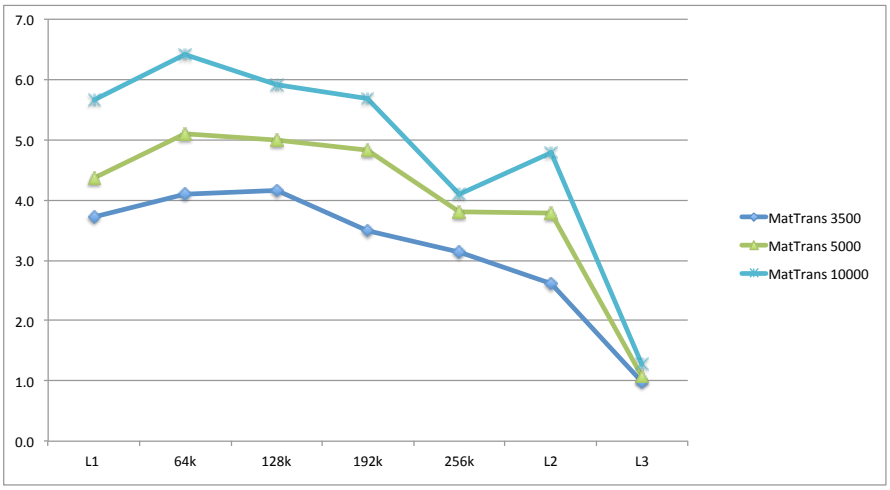

Figure 5.10: S2 Speedups: MatTrans (Contiguous Clustering)

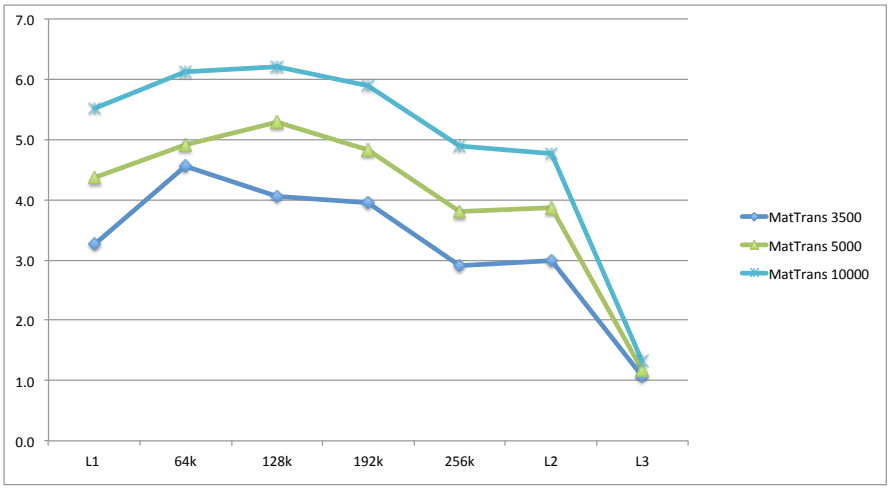

Figure 5.11: S2 Speedups: MatTrans (SRR Clustering)

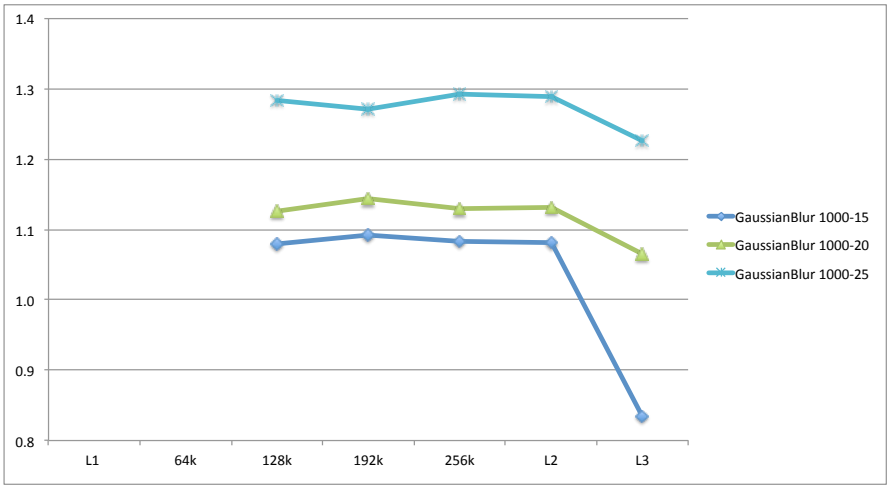

Figure 5.12: S2 Speedups: GaussianBlur (Contiguous Clustering)

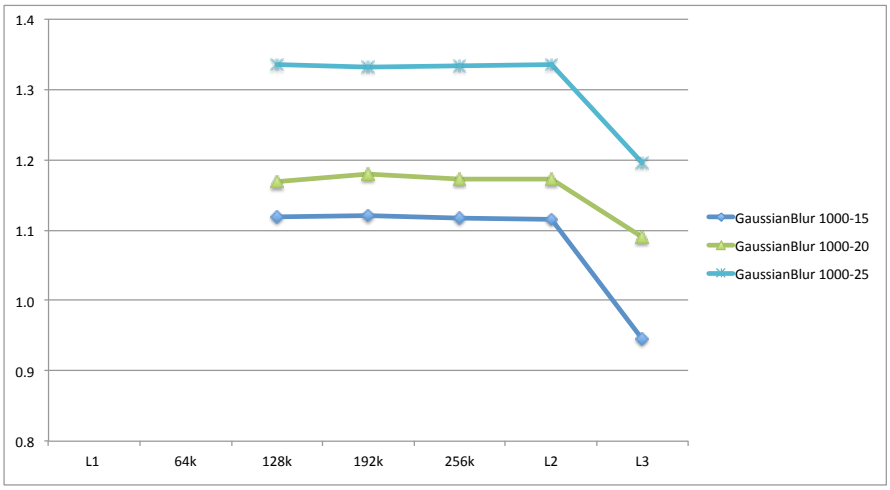

Figure 5.13: S2 Speedups: GaussianBlur (SRR Clustering)

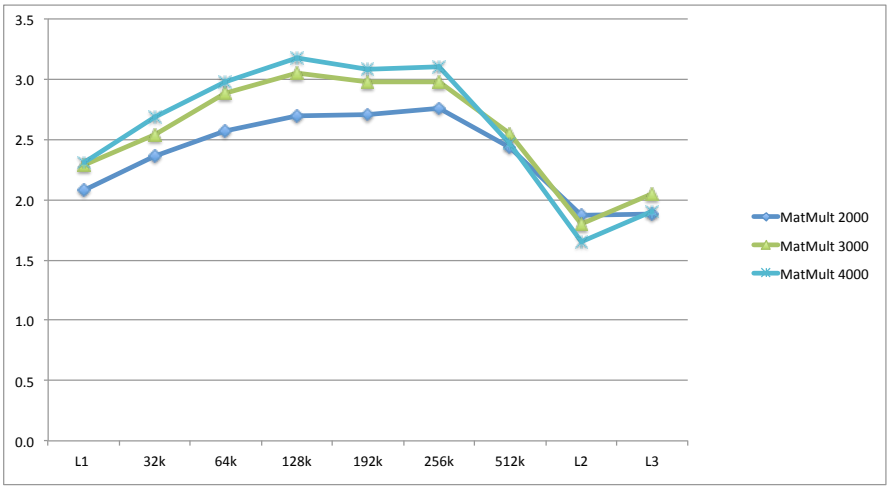

Figure 5.14: S3 Speedups: MatMult (Contiguous Clustering)

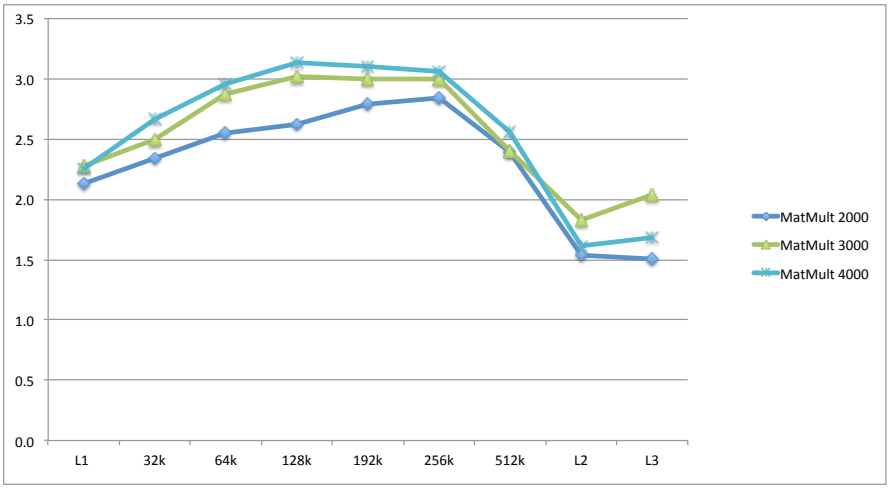

Figure 5.15: S3 Speedups: MatMult (SRR Clustering)

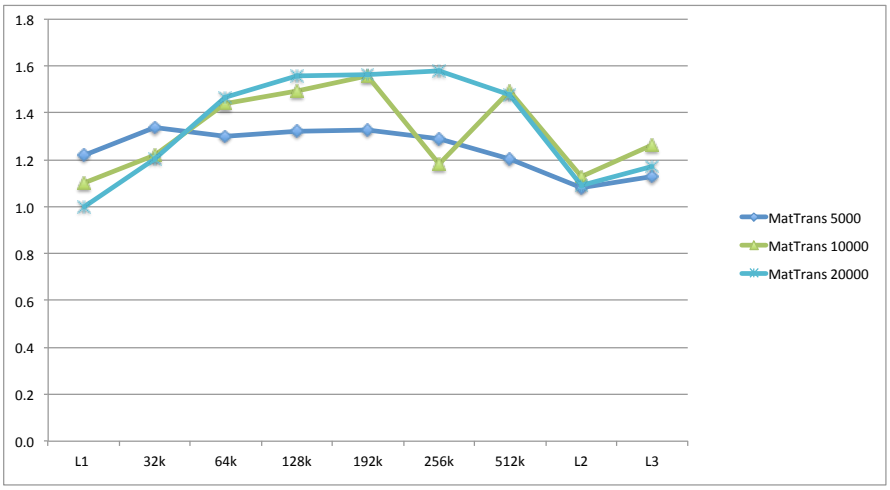

Figure 5.16: S3 Speedups: MatTrans (Contiguous Clustering)

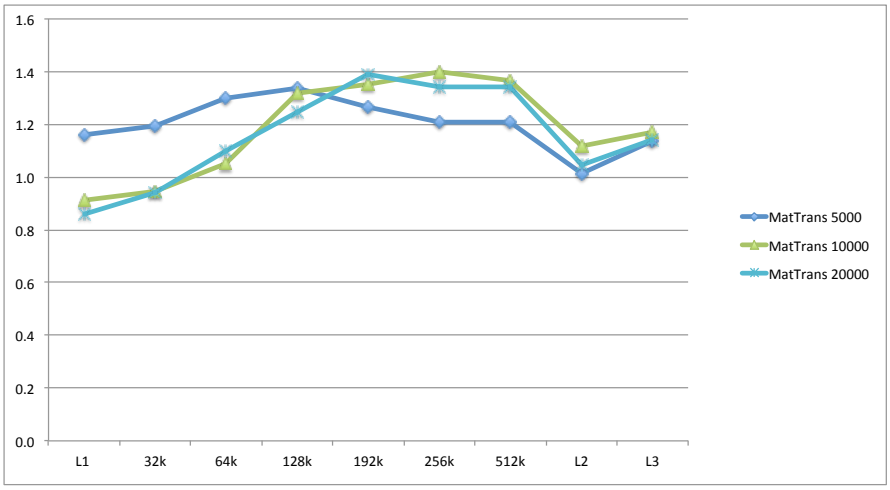

Figure 5.17: S3 Speedups: MatTrans (SRR Clustering)

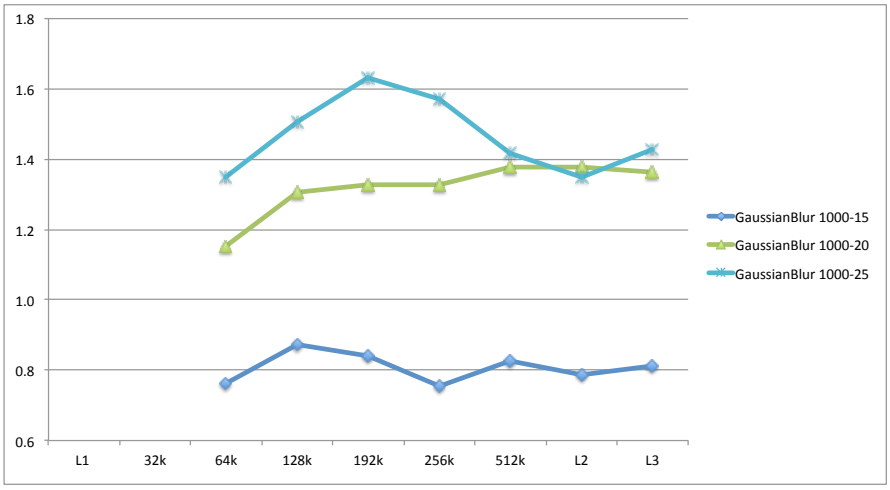

Figure 5.18: S3 Speedups: GaussianBlur (Contiguous Clustering)

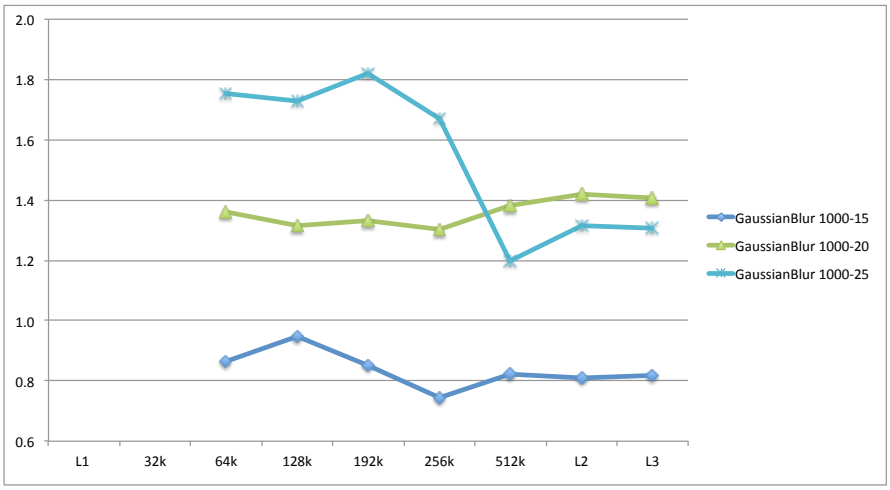

Figure 5.19: S3 Speedups: GaussianBlur (SRR Clustering)

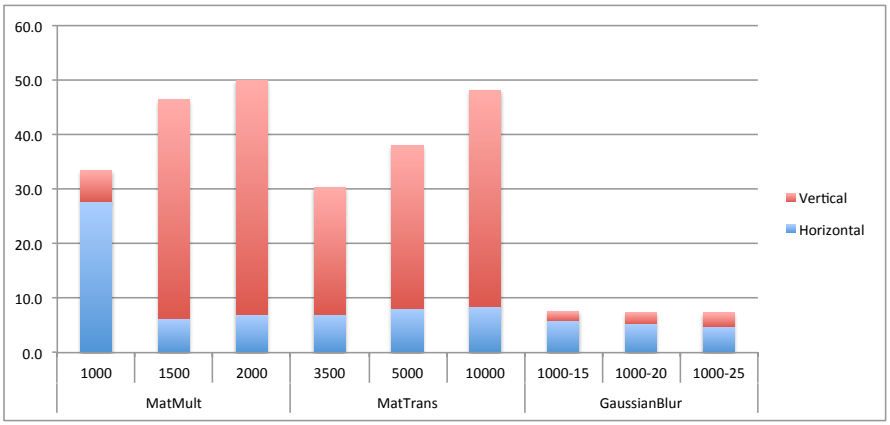

Figure 5.20: S1 Speedups: Horizontal and Vertical (Contiguous Clustering) vs Sequential

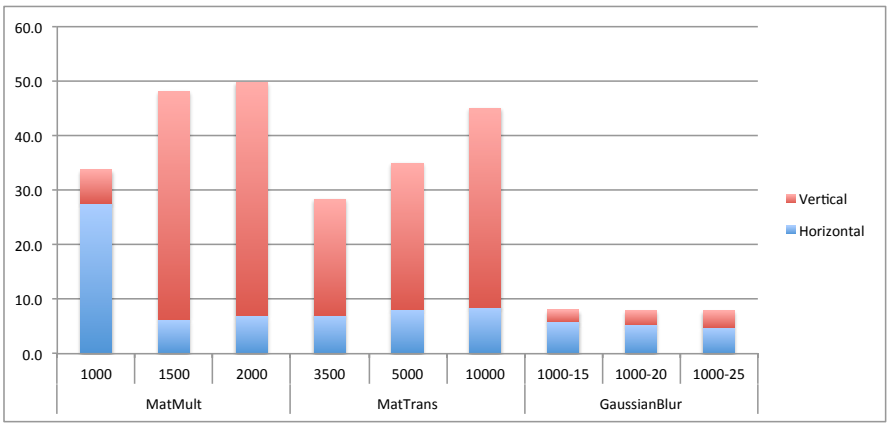

Figure 5.21: S1 Speedups: Horizontal and Vertical (SRR Clustering) vs Sequential

Regarding the GaussianBlur benchmark, in general, the results are not as favorable as in the other benchmarks. In all the evaluated systems, the peak speedups for the major classes of the benchmark reside between the 130% and 190% marks. The reason for these results is related to the irregular computational weight of the partitions generated; some of the generated partitions contain the borders (corner cases) of the input matrices, which contain additional elements to be computed. Hence, workers that are assigned inner partitions will have a smaller workload compared to the remaining workers, whose execution time will determine the overall execution time.

The difference in performance when employing Contiguous Clustering or SRR Clustering is minimal in most cases. Although the performance peak for each clustering strategy may be different, even the peak values have a small difference, which allows us to conclude that the essential performance gains result from the base vertical decomposition strategy. GaussianBlur always benefited from SRR Clustering for higher values of R since a larger blur window increases the amount of data that is shared by contiguous blur tasks.

We also evaluated the benchmarks using a cache-line-aware partitioning function, but

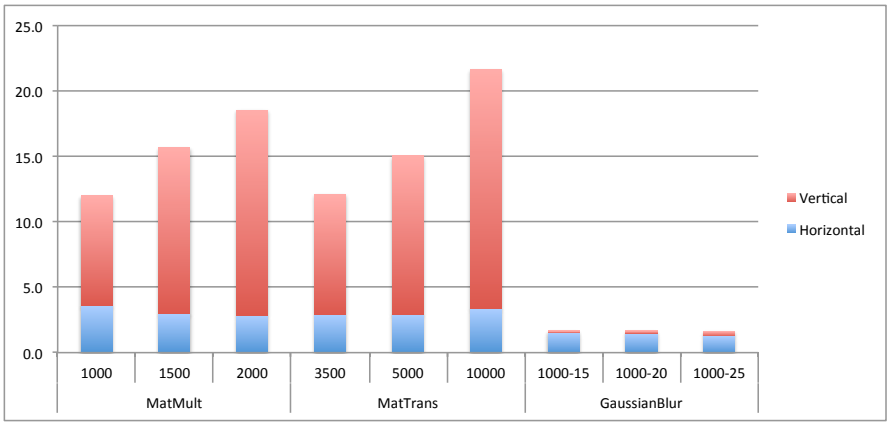

Figure 5.22: S2 Speedups: Horizontal and Vertical (Contiguous Clustering) vs Sequential

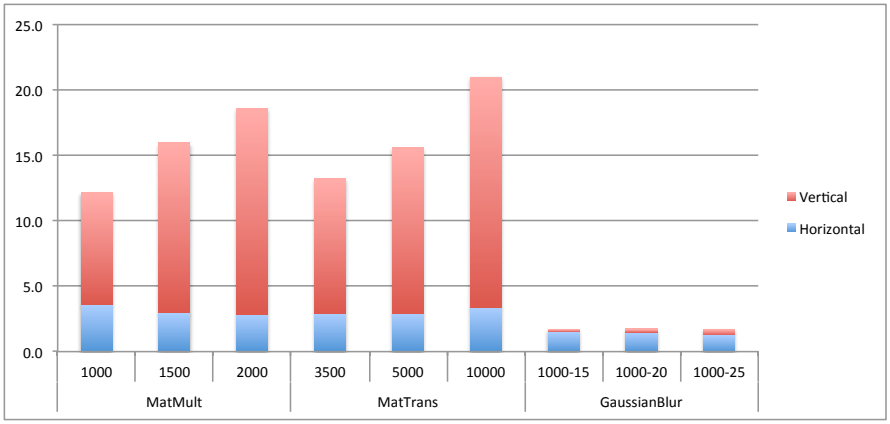

Figure 5.23: S2 Speedups: Horizontal and Vertical (SRR Clustering) vs Sequential

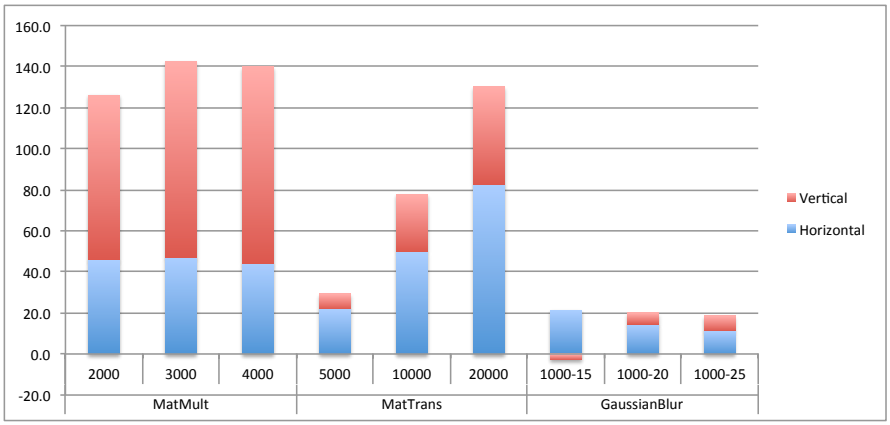

Figure 5.24: S3 Speedups: Horizontal and Vertical (Contiguous Clustering) vs Sequential

the results we obtained did not improve the performance for any of the assessed cache sizes, introducing only overhead and wasted cache space.

To close the results analysis, we present the speedups of both the horizontal decomposition and vertical decomposition executions against the sequential execution of the benchmarks, depicted in Figures 5.20 through 5.25. We present these speedups in order

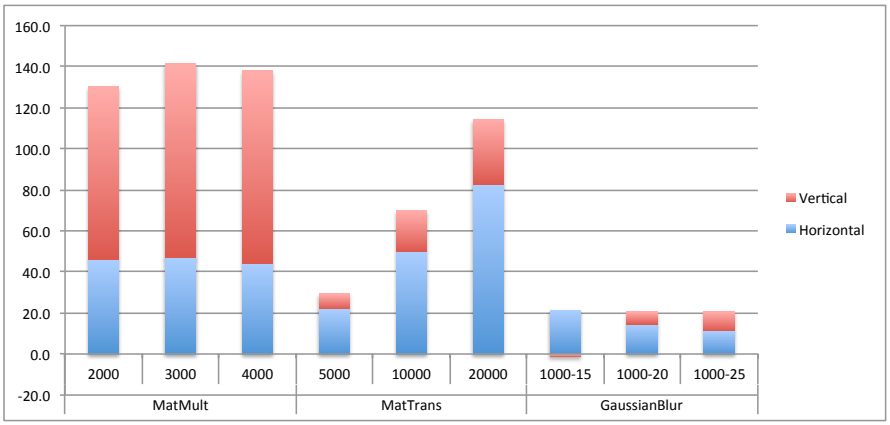

Figure 5.25: S3 Speedups: Horizontal and Vertical (SRR Clustering) vs Sequential

to prove that the implemented horizontal decomposition does in fact employ parallelism, and also to further illustrate the order of magnitude of the attained speedups. The former is confirmed by the fact that all the horizontal decomposition bars have their peak above 1.

5.4.2 Saxpy, Series

Since these benchmarks iterate data sequentially (exploiting spatial locality) without revisiting data, no benefits are attained from enforcing temporal locality through the creation of workingsets fitting the TCL. Given these properties, one can only expect to introduce overhead when employing vertical decomposition, due to the increased number of tasks. Consequently, the best cache configuration for each machine is the one that minimizes the number of created tasks in each system: L3 cache level for systems S1 and S3; and the L2 cache for S3, due to having a larger amount of space for each core sharing a L2 cache than a L3 cache.

Figures 5.26, 5.27 and 5.28 present the speedup charts for these benchmarks on systems S1, S2 and S3 respectively. The obtained speedups are close to 1 but present fluctuations that result from the non-determinism on executions.

5.4.3 Breakdown

We present the breakdown of some relevant benchmark executions, in order to evaluate the impact that the Elina's execution stages have on the overall execution time. These breakdowns are presented in Figures 5.29 through 5.32, and represent the execution breakdowns for the best execution configuration of, respectively: MatMult with N=2000 in S1, MatTrans with N=10000 in S1, Matmult with N=4000 in S3, and MatTrans with N=20000 in S3. These two benchmarks, MatMult and MatTrans, are interesting in the context of execution breakdowns because these correspond to problems, respectively, with and without a reduction stage.

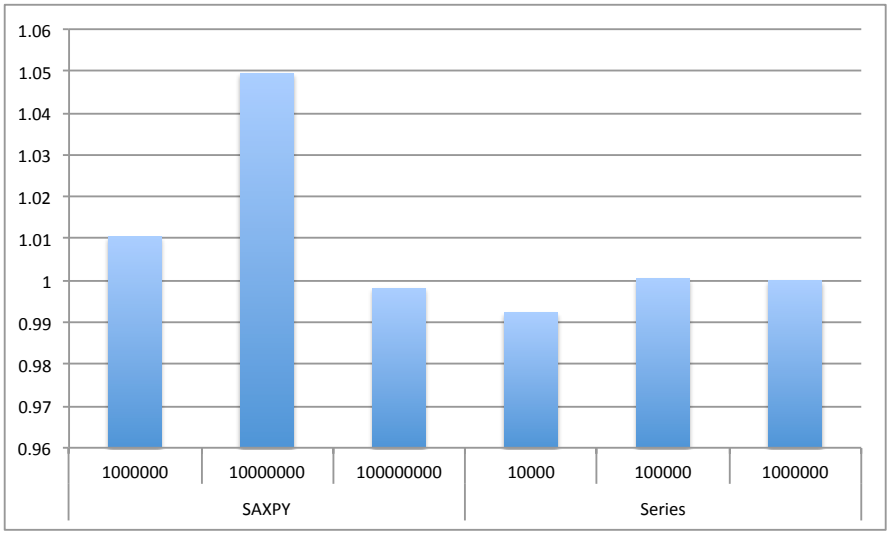

Figure 5.26: SAXPY and Series S1 best configuration speedups

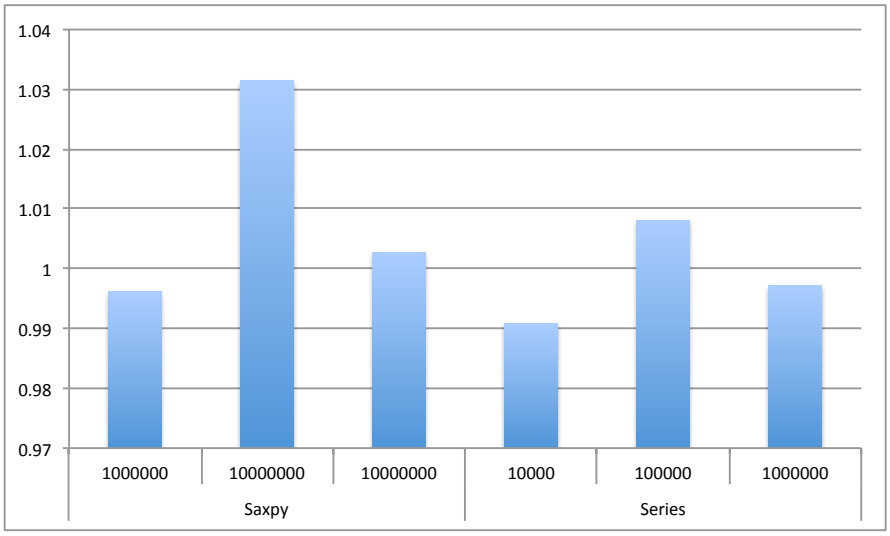

Figure 5.27: SAXPY and Series S2 best configuration speedups

Since in all the presented breakdowns, more than 99% of the total execution time corresponds to the execution of tasks, we can conclude that the achieved speedups result from the exploitation of spatial and temporal locality on the access to data during the execution. Vertical decomposition generates more tasks than horizontal decomposition, which takes its toll on the reduction stage where more partial results have to be reduced. To conclude, we can observe that the decomposition and scheduling stages do not have a major impact on the overall execution time, pertaining to less than 1% of the total execution time in all the decomposition/scheduling configurations.

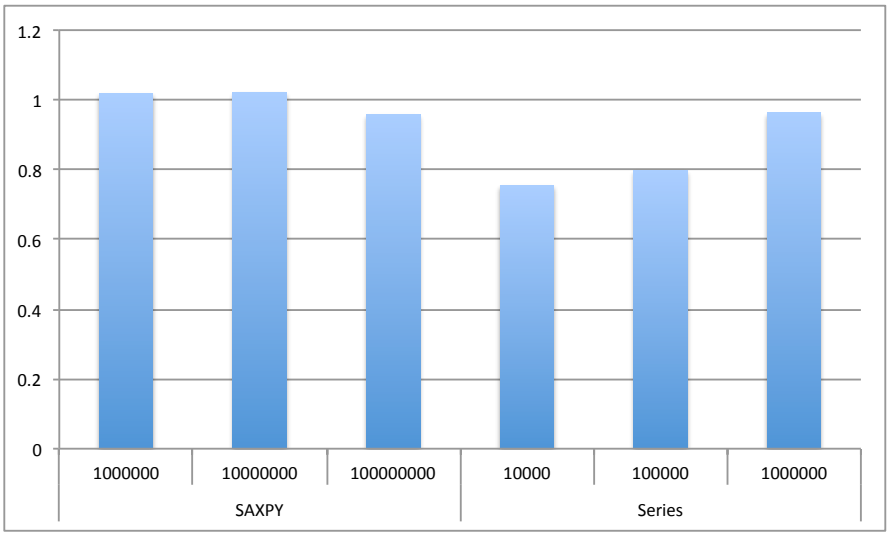

Figure 5.28: SAXPY and Series S3 best configuration speedups

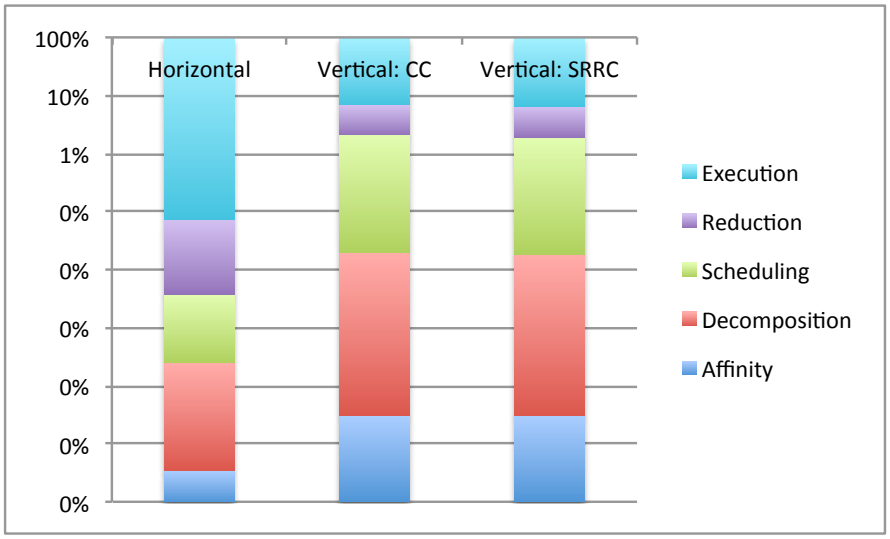

Figure 5.29: S1 Breakdown: MatMult N=2000

5.5 Discussion

The results show that vertical decomposition, regardless of the employed clustering strategy, provides significant speedups relative to horizontal decomposition. On the other hand, the combination of decomposition strategy, clustering strategy, and TCL size that produces the best speedup varies depending on the problem in hands, therefore it is not possible to systematically use the same execution settings independently of the problem. To mitigate this problem, one can augment the execution platform with a auto-learning stage that, over time, learns the best configurations to be applied for each individual problem and respective input sizes, applying these settings upon a request for execution of the given problem.

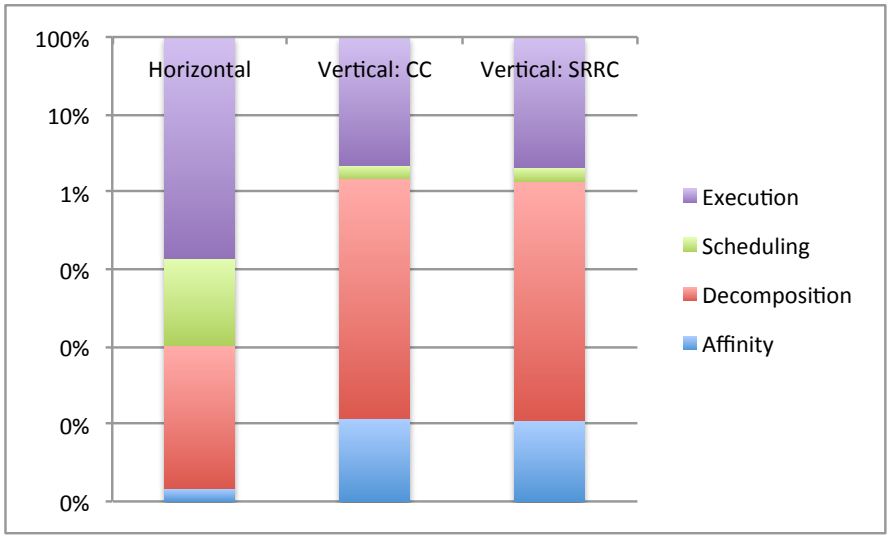

Figure 5.30: S1 Breakdown: MatTrans N=10000

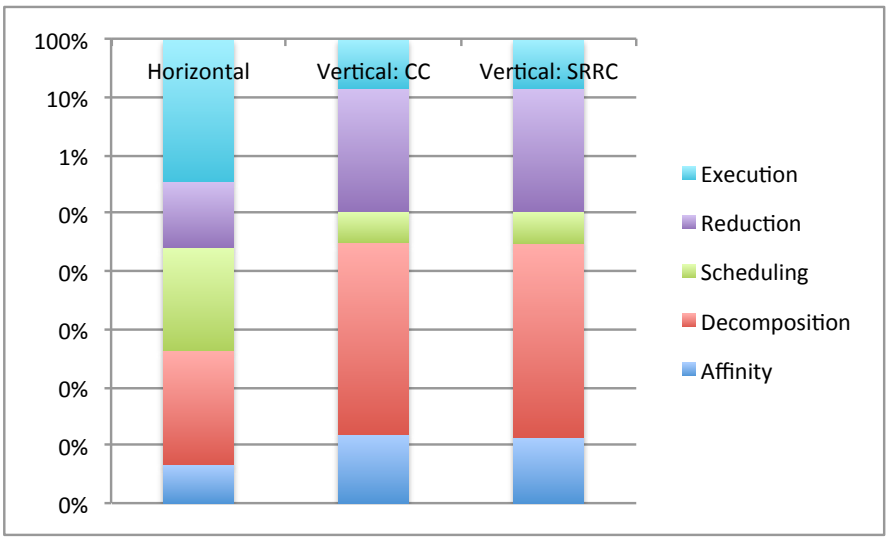

Figure 5.31: S3 Breakdown: MatMult N=4000

It may also be possible that the optimal execution settings, for each problem, can be determined if more cache hierarchy information is taken into account, namely the size of the cache groups.

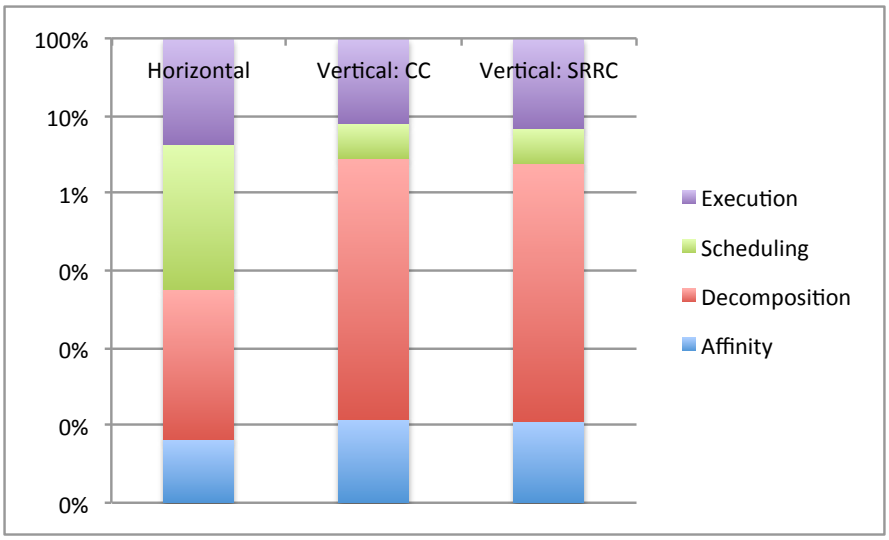

Figure 5.32: S3 Breakdown: MatTrans N=20000

6

Conclusions

The work developed during the duration of this dissertation encompassed several different aspects of vertical decomposition. Regarding the contributions that we have initially defined for this dissertation, we have accomplished the entirety of these. In the first place, Chapter 3 presented the concept and theory behind vertical decomposition, for which we identified three aspects: workingset size estimation, scheduling and affinity mapping. We devised algorithms for these aspects, discussing their accuracy and the trade-offs involved when we presented multiple algorithms.

With the theoretical concepts explained and the algorithms presented, we moved to the concrete implementation of these in our framework of choice for this dissertation, the Elina framework. The original implementation of Elina lacked some modifiability we required to experiment our strategies, namely the different approaches to task scheduling. Some of the implementation effort was spent altering Elina's core logic and involved interfaces, though in the end we made Elina a more powerful framework, particularly in what pertains to optimizations at the multiple stages of data-parallel computations. With this, we attained our objective of providing a framework for programmers to map applications onto a target machine's hierarchy, automatically and with minimal intervention from their part.

In our experimental evaluation we attempted to be exhaustive, tackling each dimension of our proposal: performance and portability. The results obtained confirmed that our approach is performant, with speedups ranging from 130% to 700% compared to horizontal decomposition. Moreover, our evaluation process employed machines with differing hierarchies, giving strength to the performance portability of our approach.

To the best of our knowledge, the study we performed in Chapter 5 is the first comparison between the two parallel decomposition approaches: horizontal and vertical.

The other studies present in the literature [BGH+06, FHK+06] focus on the gains delivered by vertical decompositions relatively to the sequential executions of the problems, something that, in our opinion, is not enough to prove the added value of vertical decomposition relative to the straightforward horizontal decomposition. With this evaluation we provided the last contribution that we defined for this dissertation.

During the implementation stage, problems sprouted and new challenges were unveiled as we progressed, especially those related to dynamic memory allocation. The anomalies that resulted from the bottleneck of memory allocation were, at first, difficult to understand, and even then the solution was not trivial.

We believe we have met the thesis statement of this dissertation. Significant performance gains were obtained through the systematic decomposition of the computation's domain.

Vertical decomposition can be further extended to the cluster level, which is the focus of future work. This transposition however, is not a trivial task, in the most part due to the heterogeneity that is typical across the machines composing a cluster. This will require the definition and implementation of a new kind of algorithms that estimate the performance of machines, so as to determine the amount of workingsets that should be assigned to each individual machine.

There is still space for optimizations at the single machine level though, particularly in the subject of dynamic load balancing and work stealing techniques, which has a total lack of Elina currently. Although these were not of major importance in the studied application problems, they will become crucial if perfomant results are to be achieved in problems featuring irregular parallelism.

As a final remark, we have identified a limitation on vertical decomposition for which we have yet to consider possible solutions: the support for constructs, such as synchronization barriers. Data-parallel applications exist that require a synchronization step, for instance, at the end of each iteration of a loop. Given that vertical decomposition produces more parallel tasks than the number of workers, and these tasks are clustered and subsequently assigned these same workers, it is not possible to synchronize and keep execution state information for each individual task at the same time.

Bibliography

[ACF93] Bowen Alpern, Larry Carter, and Jeanne Ferrante. Modeling parallel computers as memory hierarchies. In *In Proc. Programming Models for Massively Parallel Computers*, pages 116–123. IEEE Computer Society Press, 1993.

[BBC+06] Laurent Baduel, Francoise Baude, Denis Caromel, Arnaud Contes, Fabrice Huet, Matthieu Morel, and Romain Quilici. Programming, composing, deploying for the grid. In *GRID COMPUTING: Software Environments and*. Springer Verlag, 2006.

[BCL+06] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The fractal component model and its support in java. *Softw., Pract. Exper.*, 36(11-12):1257–1284, 2006.

[BCSA11] Michael Bauer, John Clark, Eric Schkufza, and Alex Aiken. Programming the memory hierarchy revisited: supporting irregular parallelism in sequoia. In Calin Cascaval and Pen-Chung Yew, editors, *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2011, San Antonio, TX, USA, February 12-16, 2011*, pages 13–24. ACM, 2011.

[BGH+06] Ganesh Biksh, Jia Guo, Daniel Hoeflinger, Gheorghe Almasi, Basilio B. Fraguela, María J. Garzarán, David Padua, and Christoph Von Praun. Programming for parallelism and locality with hierarchically tiled arrays. In *In Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 48–57, 2006.

[CDC+99] William Carlson, Jesse M. Draper, David E. Culler, Kathy Yelick, Eugene Brooks, and Karen Warren. Introduction to UPC and Language Specification. Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, 1999.

[CGS+05] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10:

An object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM.

[CLZC11] I-Hsin Chung, Che-Rung Lee, Jiazheng Zhou, and Yeh-Ching Chung. Hierarchical mapping for hpc applications. In *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '11, pages 1815–1823, Washington, DC, USA, 2011. IEEE Computer Society.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[FHK+06] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Memory - 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*, page 83. ACM Press, 2006.

[HPR+08] Mike Houston, Ji Young Park, Manman Ren, Timothy J. Knight, Kayvon Fatahalian, Alex Aiken, William J. Dally, and Pat Hanrahan. In Siddhartha Chatterjee and Michael L. Scott, editors, *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2008, Salt Lake City, UT, USA, February 20-23, 2008*, pages 143–152. ACM, 2008.

[jMIY11] Seung jai Min, Costin Iancu, and Katherine Yelick. Hierarchical work stealing on manycore clusters. In *In Fifth Conference on Partitioned Global Address Space Programming Models*, 2011.

[KY12] Amir Ashraf Kamil and Katherine A. Yelick. Hierarchical additions to the spmd programming model. Technical Report UCB/EECS-2012-20, EECS Department, University of California, Berkeley, Feb 2012.

[KY14] Amir Kamil and Katherine Yelick. Hierarchical computation in the spmd programming model. In *Proceedings of the 26th International Workshop on Languages and Compilers for Parallel Computing, LCPC 2013, San Jose, CA, USA, September 25-27, 2013*, 2014. To appear.

[LS96] R. Greg Lavender and Douglas C. Schmidt. Active object: An object behavioral pattern for concurrent programming. In *Pattern languages of program design 2*, pages 483–499. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[MP12] Eduardo Marques and Hervé Paulino. Single operation multiple data - data parallelism at subroutine level. In Geyong Min, Jia Hu, Lei (Chris) Liu, Laurence Tianruo Yang, Seetharami Seelam, and Laurent Lefevre, editors, *14th IEEE International Conference on High Performance Computing and Communication & 9th IEEE International Conference on Embedded Software and Systems, HPCC-ICESS 2012, Liverpool, United Kingdom, June 25-27, 2012*, pages 254–261. IEEE Computer Society, 2012.

[SMP12] João Saramago, Diogo Mourão, and Hervé Paulino. Towards an adaptable middleware for parallel computing in heterogeneous environments. In *2012 IEEE International Conference on Cluster Computing Workshops, CLUSTER Workshops 2012, Beijing, China, September 24-28, 2012*, pages 143–151. IEEE, 2012.

[TBA13] Sean Treichler, Michael Bauer, and Alex Aiken. Language support for dynamic, hierarchical data partitioning. In *OOPSLA*, pages 495–514, 2013.

[WMEG11] Lingyuan Wang, Saumil Merchant, and Tarek El-Ghazawi. Exploiting hierarchical parallelism using upc. In *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '11, pages 1216–1224, Washington, DC, USA, 2011. IEEE Computer Society.

[YZGS10] Yonghong Yan, Jisheng Zhao, Yi Guo, and Vivek Sarkar. Hierarchical place trees: a portable abstraction for task parallelism and data movement. In *Proceedings of the 22nd international conference on Languages and Compilers for Parallel Computing*, LCPC'09, pages 172–187, Berlin, Heidelberg, 2010. Springer-Verlag.

[ZMBK10] Gengbin Zheng, Esteban Meneses, Abhinav Bhatele, and Laxmikant V. Kale. Hierarchical load balancing for charm++ applications on large supercomputers. In *Proceedings of the 2010 39th International Conference on Parallel Processing Workshops*, ICPPW '10, pages 436–444, Washington, DC, USA, 2010. IEEE Computer Society.