Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Accelerating Sensitivity Analysis in Microscopy Image Segmentation Workflows with Multi-level Computation and Data Reuse

Willian de Oliveira Barreiros Júnior

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Orientador
Prof. Dr. George Luiz Medeiros Teodoro

Brasília
2018

Ficha Catalográfica de Teses e Dissertações

Está página existe apenas para indicar onde a ficha catalográfica gerada para dissertações de mestrado e teses de doutorado defendidas na UnB. A Biblioteca Central é responsável pela ficha, mais informações nos sítios:

http://www.bce.unb.br
http://www.bce.unb.br/elaboracao-de-fichas-catalograficas-de-teses-e-dissertacoes

Esta página não deve ser inclusa na versão final do texto.

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Accelerating Sensitivity Analysis in Microscopy Image Segmentation Workflows with Multi-level Computation and Data Reuse

Willian de Oliveira Barreiros Júnior

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Prof. Dr. George Luiz Medeiros Teodoro (Orientador)
CIC/UnB

Prof.a Dr.a Alba Cristina Magalhães Alves de Melo
CIC/UnB

Dr. Eduardo Adílo Pelinson Alchieri
CIC/UnB

Prof. Dr. Bruno Macchiavello
Coordenador do Programa de Pós-graduação em Informática

Brasília, 10 de fevereiro de 2018

Dedicatória

Ao meu pai, que tomou como missão de vida apoiar-me de todas formas a seguir minha paixão nos estudos, e que embora não esteja mais aqui, continua a suceder.

Agradecimentos

Agradeço primeiramente à minha família pelo apoio nesta etapa da minha vida, aos companheiros de laboratório que me auxiliavam irrestritamente, e a meu orientador, George, que de alguma forma conseguiu suportar constantes interrupções minhas enquanto tentava me guiar.

Resumo

Com a crescente disponibilidade de equipamentos de imagens microscópicas médicas existe uma demanda para execução eficiente de aplicações de processamento de imagens *Whole Slide Tissue Images*. Pelo processo de análise de sensibilidade é possível melhorar a qualidade dos resultados de tais aplicações, e subsequentemente, a qualidade da análise realizada a partir deles. Devido ao alto custo computacional e à natureza recorrente das tarefas executadas por métodos de análise de sensibilidade (i.e., reexecução de tarefas), emergem oportunidades para reuso computacional. Pela realização de reuso computacional otimiza-se o tempo de execução das aplicações de análise de sensibilidade. Este trabalho tem como objetivo encontrar novas maneiras de aproveitar as oportunidades de reuso computacional em múltiplos níveis de abstração das tarefas. Isto é feito pela apresentação de algoritmos de reuso de tarefas grão-grosso e de novos algoritmos de reuso de tarefas grão-fino, implementados no *Region Templates Framework*.

Palavras-chave: Reuso Computacional, Analise de Sensibilidade, *Region Templates Framework*

Abstract

With the increasingly availability of digital microscopy imagery equipments there is a demand for efficient execution of whole slide tissue image applications. Through the process of sensitivity analysis it is possible to improve the output quality of such applications, and thus, improve the desired analysis quality. Due to the high computational cost of such analyses and the recurrent nature of executed tasks from sensitivity analysis methods (i.e., reexecution of tasks), the opportunity for computation reuse arises. By performing computation reuse we can optimize the run time of sensitivity analysis applications. This work focus then on finding new ways to take advantage of computation reuse opportunities on multiple task abstraction levels. This is done by presenting the coarse-grain merging strategy and the new fine-grain merging algorithms, implemented on top of the Region Templates Framework.

Keywords: Computation Reuse, Sensitivity Analysis, Region Templates Framework

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introduction

We define algorithm sensitivity analysis (SA) as the process of quantifying, comparing, and correlating output from multiple analyses of a dataset computed with variations of an analysis workflow using different input parameters [31]. This process is executed in many phases of scientific research and can be used to lower the effective computational cost of analysis on such researches, or even improve the quality of the results through parameter optimization.

The main motivation of this work is the use of image analysis workflows for whole slide tissue images analysis [20], which extracts salient information from tissue images in the form of segmented objects (e.g., cells) as well as their shape and texture features. Imaging features computed by such workflows contain rich information that can be used to develop morphological models of the specimens under study to gain new insights into disease mechanisms and assess disease progression.

A concern with automated biomedical image analysis is that the output quality of an analysis workflow is highly sensitive to changes in the input parameters. As such, adaptation of SA methods and methodologies employed in other fields [27, 39, 6, 16], can help understanding an image analysis workflows for both developers and users. In short, the benefits of SA include: (i) better assessment and understanding of the correlation between input parameters and analysis output; (ii) the ability to reduce the uncertainty / variation of the analysis output by identifying the causes of variation; and (iii) workflow simplification by fixing parameters values or removing parts of the code that have limited or negligible effect on the output.

Although the benefits of using SA are many, its use in practice is onerous given the data and computation challenges associated with it. For instance, a single study using a classic method such as MOAT (Morris One-At-Time) [27] may require hundreds of runs of the image analysis workflow (sample size). The execution of a single Whole Slide Tissue Image (WSI) will extract about 400,000 nuclei on average and can take hours on a single

computing node. A study at scale will consider hundreds of WSIs and compute millions of nuclei per run, which need to be compared to a reference dataset of objects to assess and quantify differences as input parameters are varied by the SA method. A single analysis at this scale using a moderate sample size with 240 parameter sets and 100 WSI would take at least three years if executed sequentially [35]. Given how time consuming such analysis is, there is a demand to develop mechanisms to make it feasible, such as parallel execution of tasks and computation reuse.

The information generated by a SA method is retrieved by executing or evaluating the same workflow with slight changes on its parameters set. As such, there are several parameters sets which have parameters with similar values. The workflows used on this work are hierarchical and, as such, can be broken down in routines. As such, it would be wasteful if one of these routines were to be executed on two or more evaluations generated by the SA method with the same parameters values and other inputs. Thus, the re-executions of a given routine could reuse the results of the first execution in order to reduce the overall cost of the application.

Formally, computation reuse is the process of reusing routines results instead of re-executing them. Computation reuse opportunities arise when distinct computation tasks have the same input parameters, resulting in the same output, and thus making the re-execution of such task unnecessary. Computation reuse can also be classified by the level of abstraction of the reused tasks. Furthermore, these tasks can be combined on hierarchical workflows, with the routines and sub-routines of which they are composed by, being able to be fully or partially reused. Seizing reuse opportunities is done by a merging process, in which two or more task are merged together, after which the repeated portions of the merged tasks are set to execute only once.

Computation reuse on this work will be accomplished with the use of finer-grain tasks merging algorithms, to be integrated on top of the Region Templates Framework (RTF) platform. This platform is responsible for the distributed execution of hierarchical workflows in large scale computation environments, also abstracting dependency resolution and data management. It is important to highlight that the RTF already implements a coarser-grain reuse algorithm, and that the new merging algorithms proposed here improve the performance of SA applications alongside the existing coarse-grain merging algorithm.

Other works have studied computation reuse as a means to reduce overall computational cost in different ways [28, 32, 30, 38, 26, 33, 17, 18]. Although the principle of computation reuse is rather abstract, its implementation on this work is distinct from all found existing methods. Some of these methods resort to hardware implementations [32, 30], which are not generic or flexible enough for the given problem. Some apply reuse

by profiling the application [38], which is also impracticable on the SA domain. Finally, most of them rely on caching systems of distinct levels of abstraction to reduce the overall cost of the applications [26, 33, 17, 18], being too expensive to employ on the desired scale of distribution. Given that none of the previous approaches could perform reasonably well on the desired problem, any direct comparison would be at least unfair.

Summarizing, this work focuses on two ways of accomplishing computation reuse in SA applications for the RTF: (i) coarse-grain tasks reuse and (ii) fine-grain tasks reuse. The main differences between them is the granularity of the tasks to be reused and the underlying restrictions of the system used to execute these tasks. The reuse of coarse-grain tasks can offer a greater speedup when reuse happens, but there are less reuse opportunities. With fine-grain tasks these reuse opportunities are more frequent, however, more sophisticated strategies need to be employed in order to deal with dependency resolution and to avoid performance degradation due to the impact of excessive reuse to the parallelism.

1.1 The Problem

Because of high computing demands, sensitivity analysis applied to microscopy image analysis is unfeasible for routinely use when applied to whole slide tissue images.

1.2 Contributions

This work focuses on improving the performance of SA studies in microscopy image analysis through the application of finer-grain computation reuse on top of the already proposed coarse-grain computation reuse.

The specific contributions of this work are presented below with a reference to the section in which they are described:

1. A graphical user interface for simplifying the deployment of workflows for the RTF, which is coupled with a code generator that allows the flexible use of the RTF on distinct domains [Section 3.1];
2. The development and analysis of multi-level reuse algorithms:
   (a) A coarse-grain merging algorithm was implemented [Section 3.2];
   (b) A fine-grain Naïve Merging Algorithm was proposed and implemented [Section 3.3.1];

(c) The fine-grain Smart Cut Merging Algorithm was proposed and implemented [Section 3.3.2];

(d) The fine-grain Reuse-Tree Merging Algorithm was proposed and implemented [Section 3.3.3];

3. Proposal and implementation of the Task-Balanced Reuse-Tree Merging Algorithm that reduces the issue of loss of parallelism due to load imbalance provoked by the Reuse-Tree Merging Algorithm [Section 3.3.4];

4. The performance gains of the proposed algorithms with a real microscopy image analysis application were demonstrated using different SA strategies (e.g MOAT and VBD) at different scales.

The contributions of Sections 3.1, 3.2 and 3.3.3 were published on the IEEE Cluster 2017 conference, with the contributions of Section 3.3.4 being currently drafted for a submission for a periodic publication.

1.3 Document Organization

The next section describes the motivating application, the theory behind computation reuse and the Region Templates Framework (RTF), which was used to deploy the application on a parallel machine and is also the tool in which the merging algorithms were incorporated. After these considerations more relevant work is analyzed. Section III describes the proposed solutions for multi-level computation reuse, their implementations and optimizations. On Section IV the experimental procedures are described and the results are analyzed. Finally, Section V closes this work with contributions and possible future goals for its continuation.

Capítulo 2

Background

This chapter describes the motivating application along with the Region Templates Framework, in which this work is developed, and some basic concepts of sensitivity analysis and computation reuse. This chapter then closes with the analysis of some relevant work on computation reuse.

2.1 Microscopy Image Analysis

It is now possible for biomedical researchers to capture highly detailed images from whole slide tissue samples in a few minutes with high-end microscopy scanners, which are becoming evermore available. This capability of collecting thousands of images on a daily basis extends the possibilities for generating detailed databases of several diseases. Through the investigation of tissue morphology of whole slide tissue images (WSI) there is the possibility of better understanding disease subtypes and feature distributions, enabling the creation of novel methods for classification of diseases. With the increasing number of research groups working and developing richer methods for carrying out quantitative microscopy image analyses [14, 29, 22, 10, 12, 8, 9, 24] and also the increasingly availability of digital microscopy imagery equipment, there is a high demand for systems or frameworks oriented towards the efficient execution of microscopy image analysis workflows.

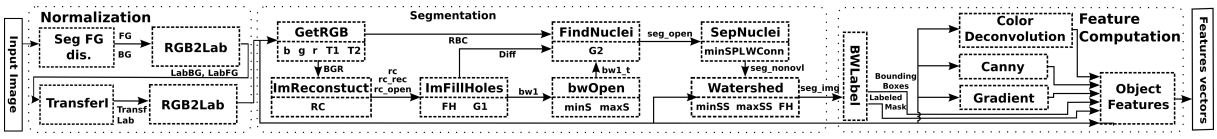
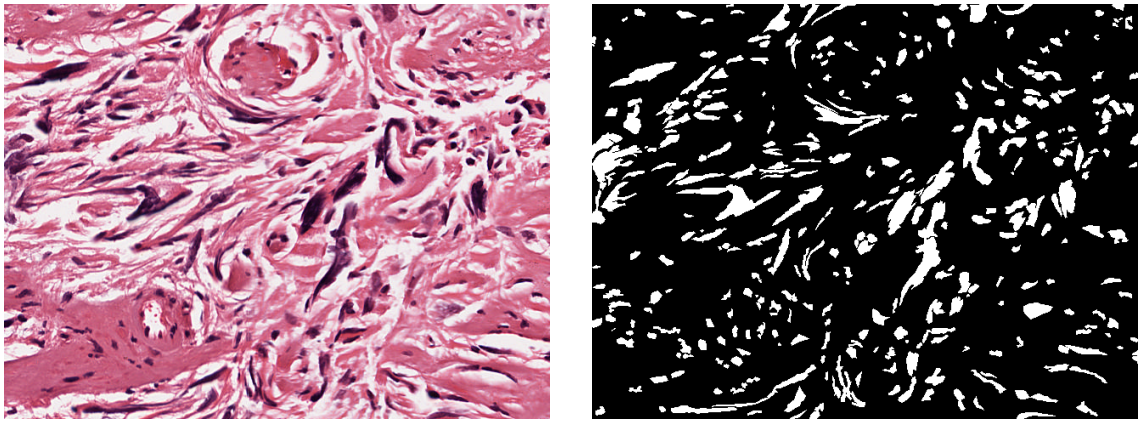


Figura 2.1: An example microscopy image analysis workflow performed before image classification. Image extracted from [3].



(a) A tissue image. (b) The segmented tissue image.

Figura 2.2: An example of tissue image segmentation.

The microscopy image analysis workflow used on this work is presented in Figure 2.1 and was proposed by [21]. This workflow consists of normalization, segmentation, feature computation and final classification, being the first three analysis stages the most computationally expensive phases. The first stage is responsible for normalizing the staining and/or illumination conditions of the image. The segmentation is the process of identifying the nucleus of each cell of the analyzed image (Figure 2.2). Through feature computation a set of shape and texture features is generated for each segmented nucleus. Finally, the final classification will typically involve using data mining algorithms on aggregated information, by which some insights on the underlying biological mechanism that enables the distinction of subtypes of diseases are gained.

The quality of the workflow analysis is, however, dependent of the quality of the parameters values, with them described in Table 2.1. Therefore, in order to improve the effectiveness of the analysis the impact of these parameters on the output of the used workflow (Figure 2.1) should be analyzed. This impact analysis is known as sensitivity analysis and is detailed on the following section.

2.2 Sensitivity Analysis

We define Sensitivity Analysis (SA) as the process of quantifying, comparing and correlating the input parameters of a workflow with the intent of quantifying the impact of each input to the final output of the workflow [31]. This process is applied on several phases of scientific research including, but not limited to model validation, parameter studies and optimization, and error estimation [1].

Usually, the computational cost for performing SA on a workflow is directly proportional to the number of parameters it has. One way to simplify the analysis on applications

Parameter	Description	Range Values
B/G/R	Background detection thresholds	$[210, 220, ..., 240]$
T1/T2	Red blood cell thresholds	$[2.5, 3.0, ..., 7.5]$
G1/G2	Thresholds to identify candidate nuclei	$[5, 10, ..., 80]$ $[2, 4, ..., 40]$
MinSize(minS)	Candidate nuclei area threshold	$[2, 4, ..., 40]$
MaxSize(maxS)	Candidate nuclei area threshold	$[900, .., 1500]$
MinSizePl (minSPL)	Area threshold before watershed	$[5, 10, ..., 80]$
MinSizeSeg (maxSS)	Area threshold in final output	$[2, 4, ..., 40]$
MaxSizeSeg (minSS)	Area threshold in final output	$[900, .., 1500]$
FillHoles(FH)	propagation neighborhood	[4-conn, 8-conn]
MorphRecon(RC)	propagation neighborhood	[4-conn, 8-conn]
Watershed(WConn)	propagation neighborhood	[4-conn, 8-conn]

Tabela 2.1: Definition of parameters and range values: parameter space contains about 21 trillion points.

with large numbers of parameters, thus reducing its cost, is through the removal of parameters whose effect on the output is negligible.

This work focus on extending the already existing system, the Region Templates Framework (RTF) [35, 36], which performs sensitivity analysis in two phases. On the first phase the 15 input parameters (Table 2.1) are screened with a light, or less compute demanding, SA method, used to remove the so called non-influential parameters from the next phase. Afterwards, a second SA method is executed on the remaining parameters, on which both first-order and high-order effects of these on the application output are quantified. This two-phase analysis is performed since the cost of more specific approaches (e.g., VBD) are prohibitively expensive.

This multi-phase sensitivity analysis process is approached on [1] as an alternative to cope with costly analysis. The application case of this work uses a complex model with several input parameters (see Table 2.1) and a high execution cost. As such, it is recommended that a lighter preliminary analysis method should be executed on the full range of input parameters, only to reduce these to a smaller subset of important parameters. As a way to further reduce the analysis complexity on this first screening analysis is to also drop inputs' correlation analysis. After the execution of a screening method, more complex and comprehensive analysis methods can be performed on a subset of the input parameters. The chosen SA methods for this work were Morris One-At-A-Time as a screening method [27], and Variance-Based Decomposition as a more complete analysis.

The light SA method, Morris One-At-A-Time (MOAT) [27], performs a series of runs of the application changing each parameter individually, while fixing the remaining parameters in a discretized parameter search space. Each of the k analyzed parameters values ranges are uniformly partitioned in p levels, thus resulting in a p^k grid of parame-

ter sets to be evaluated. Each evaluation output x_i of the application creates a parameter elementary effect (EE), calculated as $EE_i = \frac{y(x_1,\ldots,x_i+\Delta_i,\ldots,x_k)-y(x)}{\Delta_i}$, with $y(x)$ being the application output before the parameter perturbation. In order to account for global SA the RTF uses $\Delta_i = \frac{p}{2(p-1)}$ [36]. The MOAT method requires $r(k+1)$ evaluations, with r in the range of 5 to 15 [15].

The second SA method, Variance-Based Decomposition (VBD) is preferably performed after a lighter SA screening method, as the MOAT method. This is done since VBD requires $n(k+2)$ evaluations for k parameters and n samples, with n lying in the order of thousands of executions [39]. Thus, it is interesting to use a reduced number of parameters for feasibility reasons. VBD, unlike MOAT, discriminates the the output uncertainty effects among individual parameters (first-order) and high-order effects.

Regardless of the SA method chosen, the use of large set of parameters (Table 2.1) results in the unpractical task of performing SA on the workflow of Figure 2.1 due to the expected cost of evaluating such large search domain. For the sake of extenuating this infeasibility issue for performing SA on the presented workflow we can execute the analysis on high-end distributed computing environments. Also, computation reuse can be employed to reduce the computational cost without the need of application specific optimizations. Both mentioned methods are described in the next sections.

2.3 Region Templates Framework (RTF)

The Region Template Framework (RTF) abstracts the execution of a workflow application on distributed environments [35]. It supports hierarchical workflows that are composed of coarse-grain stages, which in turn are composed by fine-grain tasks. The dependencies between stages, and tasks of a single stage are solved by the RTF. Given a homogeneous environment of n nodes with k cores each, any stage instance must be executed on a single node, with its tasks being executed on any of the k cores of the same node. It is noteworthy that, not only any node can have more than one stage instance executing on it, but also, there may be more than one task from the same stage running in parallel, given that the inter-tasks dependencies are respected.

The main components of the RTF are: the data abstraction, the runtime system, and the hierarchical data storage layer [35]. The runtime system consists of core functions for scheduling of application stages, transparent data movement and management via the storage layers. Figure 2.3 shows an example of the dispatch of a stage to a worker with the data exchanges in the RTF storage layer. The RTF, with its centralized Manager, distributes the stages to be executed to Worker nodes across the network. The hierarchical workflow representation allows for different scheduling strategies to be used at each level

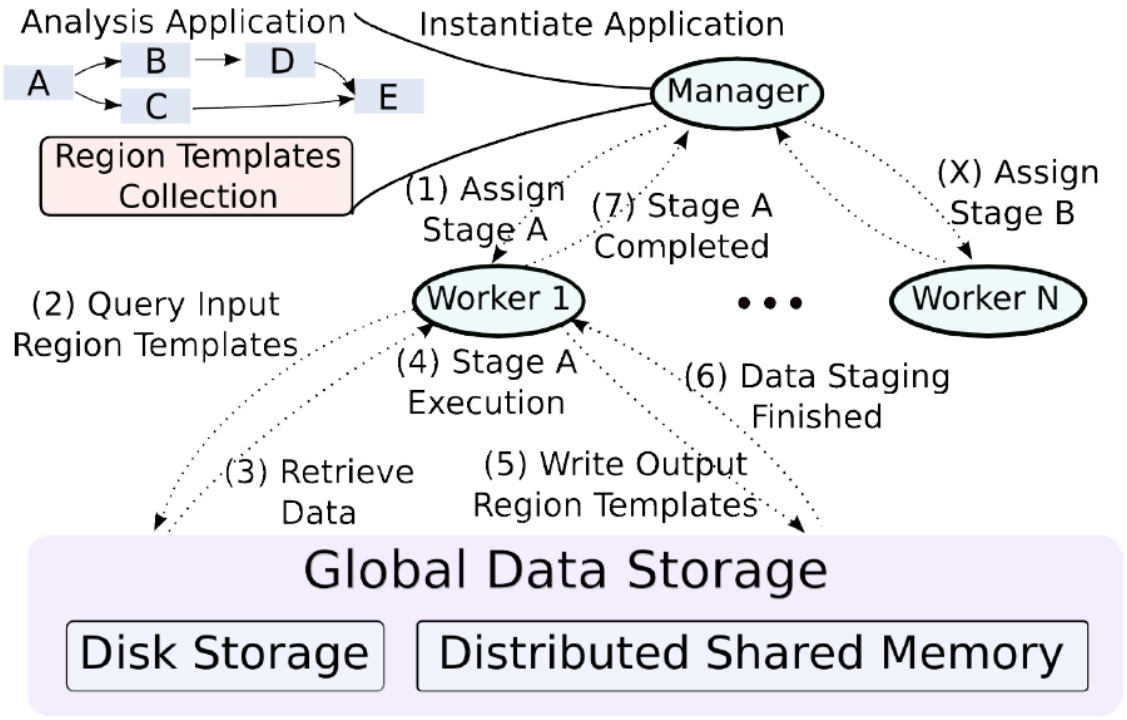


Figura 2.3: The main components of the Region Templates Framework, highlighting the steps of a coarse-grain stage instance execution. Image extracted from [35].

(stage-level and task-level). Fine-grain scheduling is possible at task-level in order to also exploit variability in performance of application operations in hybrid systems. In Figure 2.4 a stage A is sent to a worker node for execution, which tasks are scheduled locally.

Still on the scheduler, the Manager schedules stages to Workers on a demand-driven basis, with the Workers requesting work from the Manager until all stages are executed. Since the Worker decides when they request more work, a Worker can execute one or more stage at any given time instance, based on its underlying infrastructure. Being a stage composed of tasks, these are scheduled locally by the Worker executing them. These tasks differ in terms of data access patterns and computation intensity, thus, attaining different speedups if executed on co-processors or accelerators. In order to optimize the execution of tasks a Performance Aware Task Scheduling (PATS) was implemented [35]. With PATS, tasks are assigned to either a CPU or GPU core based on its estimated acceleration and the current device load.

On the data storage layer the Region Templates (RT) data abstraction is used to represent and interchange data (represented by the collection of objects of an application instance and the stored data of Figure 2.3). It consists of storage containers for data structures commonly found in applications that process data in low-dimensional spaces (1D, 2D or 3D spaces) with a temporal component. The data types include: pixels, points,

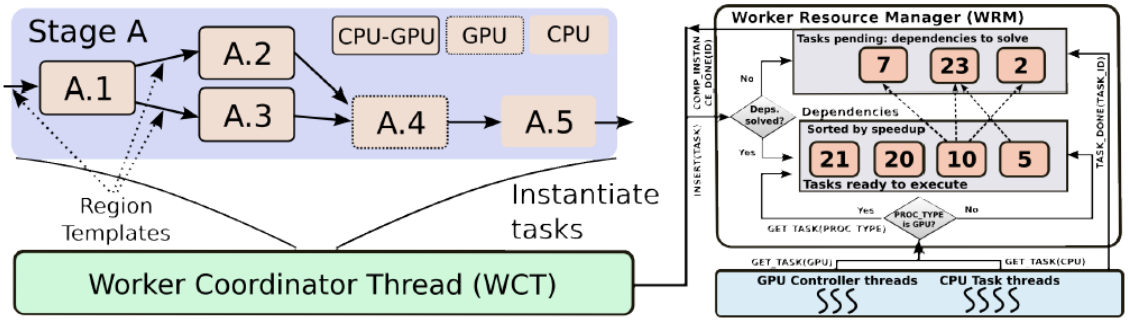


Figura 2.4: The execution of a stage instance from the perspective of a node, showing the fine-grain tasks scheduling. Image extracted from [35].

arrays (e.g., images or 3D volumes), segmented and annotated objects and regions, all of which are implemented using the OpenCV [5] library interfaces to simplify their use. A RT data instance represents a container for a region defined by a spatial and temporal bounding box. A data region object is a storage materialization of data types and stores the data elements in the region contained by a RT instance. A RT instance may have multiple data regions.

Access to the data elements in data regions is performed through a lightweight class that encapsulates the data layout, provided by the RT library. Each data region of one or multiple RT instances can be associated with different data storage implementations, defined by the application designer. With this design the decisions regarding data movement and placement are delegated to the runtime environment, which may use different layers of a system memory to place the data according to the workflow requirements.

The runtime system is implemented through a Manager-Worker execution model that combines a bag-of-tasks execution with workflows. The application Manager creates instances of coarse-grain stages, and exports the dependencies among them. These dependencies are represented as data regions to be consumed/produced by the stages. The assignment of work from the Manager to Worker nodes is performed at the granularity of a stage instance using a demand-driven mechanism, on which each Worker node independently requests stages instances from the Manager whenever it has idle resources. Each node is then responsible for fine-grain task scheduling of the received stage(s) to its local resources.

To create an application for the RTF the developer needs to provide a library of domain specific data analysis operations (in this case, microscopy image analysis) and implement a simple startup component that generates the desired workflow and starts the execution. The application developer also needs to specify a partitioning strategy for data regions encapsulated by the region templates to support parallel computation of said data regions associated with the respective region templates.

Stages of a workflow consume and produce Region Template (RT) objects, which are handled by the RTF, instead of having to read/write data directly from/to stages or disk. While the interactions between coarse-grain stages are handled by the RTF, the task of writing more complex, fine-grained, stages containing several external, domain specific, fine-grain API calls is significantly harder for application experts. This occurs since the RTF works only with one type of task objects as its runnable interface, not providing an easy way to compose stages using fine-grain tasks. The RTF also supports efficient execution on hybrid systems equipped with CPU and accelerators (e.g, GPUs).

2.4 Related Work on Computation Reuse

The idea of work reuse, also known as value locality [7, 23], has been employed on both the hardware and software fronts with diverse techniques, such as value prediction [28], dynamic instruction reuse [32] and *memoization* [30], with the goal of accelerating applications through the removal of duplicated computational tasks. This concept has been used for runtime optimizations on embedded systems [38], low-level encryption value generation [26] and even stadium designing [33]. In order to further analyze these existing approaches as to find desirable features that solve the problem approached in this work a qualitative analysis was performed, which is summarized on Table 2.2. This analysis uses taxonomic terms defined here to classify computation reuse approaches. The proposed taxonomic terminology is explained on the next section in addition to a brief analysis of each of the studied computation reuse approaches.

2.4.1 Computation Reuse Taxonomy

Implementation Level (IL)

Computation reuse can be enforced on either **Software** (S) or **Hardware** (H) levels. By **Software-Level** it is meant a hardware-independent approach that can either be executed as a static analysis before the execution of any computational task, or as a runtime service that performs computation reuse as the application is executed. Also, it is possible for computation reuse to be searched on compilation-time by a customized compiler, which is also defined as **Software-Level**. It is also possible for these techniques to be combined.

Software-Level approaches provide better flexibility since the algorithm may be compiled on any architecture. However, if there is the need for a runtime service to be executed (e.g., caching), these can be more efficiently implemented on **Hardware-Level**. Implementing reuse on **Hardware-Level** also enables finer-grain tasks to be used. By using a custom compiler on **Software-Level** it is possible to have access to both finer-grain and

Reference	IL	AF	Reuse Strat.	Tasks Granularity	RTM	Reuse Eval.	T	RES
[28]	H	Flexible	Predictive	Instruction-Level	Same	Dynamic	No	L
[32]	H	Flexible	Memoization	Instruction-Level	Same	Dynamic	No	L
[30]	H	Flexible	Memoization	Instruction-Level Trivial Operations	Same	Dynamic	No	L
[38]	S	Flexible	Analytic + Memoization	Fine-Grain Regions of Code	Same	Static	Yes	L
[26]	S	Partial	Memoization	Coarse-Grain	Same	Dynamic	No	D
[17]	S	Flexible	Memoization	Full Application	Same	Dynamic	No	D
[18]	S	DS	Memoization	Coarse-Grain	Same	Dynamic	No	L
[42]	H+S	DS	Memoization	Instruction-Level Complex Operations	Similar	Dynamic	Yes	L
[7]	H+S	Partial	Memoization	Instruction-Level + Fine-Grain Regions of Code	Same	Static + Dynamic	Yes	L
[2]	H	Partial	Memoization	Instruction-Level Floating-Point Operations	Similar	Dynamic	No	L
[41]	S	DS	Analytic	Coarse-Grain	Same	Dynamic	No	L
[25]	H	Partial	Memoization	Instruction-Level Floating-Point Operations	Similar	Dynamic	No	L
[23]	H	Flexible	Memoization	Instruction-Level	Same	Static	No	L
This Work	S	Partial	Analytic	Coarse-Grain and Fine-Grain	Same	Static	No	D

Tabela 2.2: Taxonomic evaluation of computation reuse approaches. Implementation Level (IL): Hardware (H) or Software (S). Application Flexibility (AF): Flexible, Partial or Domain Specific (DS). Reuse Strategy: Predictive, Memoization or Analytic. Task Granularity: Instruction-Level, Fine-Grain, Coarse-grain or Full Application. Reusable Tasks Matching: Same or Similar. Reuse Evaluation: Static or Dynamic. Needs Training Step (T). Reusability Environment Scale (RES): Local (L) or Distributed(D).

coarser-grain reusable tasks. However, compilers tend to share the lack of flexibility for distinct applications of **Hardware-Level** approaches.

Application Flexibility

Here we define the **Application Flexibility**(AF) of an approach as either **General**, **Partial** or **Domain Specific** (DS). A **General** approach is any that do not have domain-specific restrictions that limits or prevents its use on different domains. The flexibility of an approach can also be **Partial**, meaning that either some non-trivial adaptations need to be employed or that anything outside its application domain will execute rather poorly.

Reuse Strategy

One of the most important computation reuse characteristics is how computation reuse opportunities are found and explored. Computation reuse can even be said to be attainable

through the speculative technique of **Value Prediction** [32, 7] since its implementations rely on a buffer that contains the results of previous instructions executions.

The most common technique for computation reuse is through **Memoization**, which is a cache-based approach on which reusable tasks results are stored on a buffer for later reuse. This approach has the drawback of needing a buffer structure, being able to be implemented on either **Hardware-Level** or **Software-Level**. On **Hardware-Level** this buffer, or cache, is usually small since caching is expensive (both footprint and financial wise), and cannot have a flexible size. These limitations are inexistent with **Software-Level** buffers, however, at the cost of performance. For a buffer implementation on this level a runtime system is necessary in order to keep track of all reusable results.

The alternative to **Memoization** is to find all reuse opportunities in an **Analytic** manner. This means that the reused tasks were found *a priori*, instead of searching the results in a buffer as with the **Memoization** scheme. While this approach is considered to be the one with the least overhead, such analysis is more difficult to be achieved.

Tasks Granularity

Still another rather important aspect of computation reuse is the **Granularity** of the reusable tasks. On this work we break **Task Granularity** in four categories: **Instruction-Level** (i.e., CPU instruction), **Fine-Grain Subroutines**, **Coarse-Grain Routines** and **Full Application**. We differentiate **Fine-Grain** from **Coarse-Grain** tasks by their semantical meaning, and as a consequence, their overall cost. If a task is big enough to have a broader meaning (e.g., a segmentation operation, a hash key calculation) we call it a **Coarse-Grain Routine** or task. If the task is bigger than a CPU instruction but also not big enough to have a more abstract meaning (e.g., the preparation of a matrix on memory, or a set of loops on an algorithm) we define them as **Fine-Grain Subroutines** or tasks. Finally, some approaches may only be able to work with a **Full Application** or execution.

The importance of the granularity for computation reuse is that it limits the maximum amount of reuse of any application. As an example we have a segmentation algorithm. If we were to break it in CPU instructions and then perform a complete search for reusability (i.e., search for all available reuse) we would attain the maximum possible reuse. However, in order to do such we would either have to search a large number of task (CPU instructions on this example) if implemented on **Software-Level**, or need a rather large cache to hold all the reusable values. By grouping this low-level operations into subroutines we reduce the number of tasks, making the search for reuse more feasible. This grouping would also hide some reuse opportunities, effectively reducing the reuse potential of the application. Notwithstanding, this grouping into coarser-grained tasks would also make

the reusable tasks more complex, meaning that they can have more possible outcomes and, as a consequence, making bigger buffers necessary for **Memoization**-based approaches.

Reusable Tasks Matching (RTM)

An easy way to improve the reuse degree of an application is by relaxating the matching constraint for reuse. By doing this, reuse is possible even if not all tasks' parameters match (**Same**). The obvious consequence of doing this relaxation is that the tasks' results will be different. However, some applications have fault-tolerance for small imprecisions of its tasks (e.g., neural networks, multimedia applications, floating-point operations). As such, given that these partial (or **Similar**) matchings respect the precision necessary for these fault-tolerant applications, this strategy can improve the amount of reuse available.

Reuse Evaluation

Computation reuse can be analyzed either **Dynamically**, at runtime, of **Statically** before the execution of any task. Performing **Static** analysis incur in no runtime overhead, as opposed to a **Dynamic** approach. However, dynamism on computation reuse analysis is interesting for load balancing reasons.

Training Required (T)

Approaches that rely on domain-specific characteristics of applications (e.g., neural networks) usually require a **Training** step before the reuse analysis. For these approaches it is important to be mindful of the **Training** cost, as it is possible for it to dominate the full analysis makespan.

Reusability Environment Scale (RES)

The reusable tasks scope is defined here as the **Reusability Environment Scale**. The tasks can be reusable among a **Distributed** (D) environment of computing nodes or reused only **Locally** (L). For tasks to be able to only be reused **Locally** is very restrictive, since the maximum scale of the application becomes limited rather quickly.

2.4.2 Related Work Analysis

Sodani and Sohi [32] motivate their work by drawing a parallel of a computation *reuse buffer* used to optimize instruction execution with memory cache used to optimize memory access instructions. Their approach aims to reduce computational cost through reuse by (i) ending the instruction pipeline earlier, thus also reducing resources conflicts, and (ii)

by breaking dependencies of later instructions, which can be executed earlier since the necessary inputs are already present. They initially proposed their *reuse buffer* as a way to reduce branch misprediction penalties. However, the effectiveness of this approach proved itself much more powerful since the reuse frequency of other, more generic, types of instructions also proved to be high. Their implementation focus on adding a *reuse buffer* to any generic dynamically-scheduled superscalar processor, using one of the three instruction reuse schemes proposed by them.

The approach on [32] can be used for any application domain while also being exposed to the largest possible amount of reuse opportunities. Their incorporation of the *reuse buffer* in a superscalar processor is done without impacting the pipeline critical path, thus having no negative impact on non-reused instructions. Nevertheless, the efficiency of all instruction reuse schemes are heavily reliant on the buffer size. Although the used buffer sizes tested by them are small, this dependency is a limiting factor for the approach since smaller buffers means less reuse opportunities. Finally, the use of a hardware-based approach limits its use even further given the difficulty to design a processor for this sole purpose.

The work on [30], similarly to [32], also uses hardware-level *memoization*, but this time with a subset of operations called *trivial computation*. These are potentially complex operations that are trivialized by simple operands (e.g., integer division by two). This strategy greatly simplifies the reuse protocol (i.e., whether an instruction is reused, insertion and replacement policies) at the cost of reuse opportunities. The speedups achieved by this approach were only significant when the application was favorable to the reuse strategy (e.g., Ackerman-like applications with huge amounts of trivial operations, or floating-point-intensive programs, which have naturally long-latency instructions). The same limitations of [32] were present here as well.

Wang and Raghunathan [38] attempt to reduce the energetic cost of embedded software on battery-powered devices through a profiling-based reuse technique with a *memoization* structure. Some interesting discussions risen in their work regard reusable tasks granularity and the limitations of hardware-based reuse. Hardware implementations of computation reuse are usually complex, and the use of overly fine-grained operations for reuse may yield little or negative speedups given the overhead of *memoization* caches.

The methodology of [38] consists on profiling an application, generating *computation reuse regions*, setting the software cache configuration, evaluating the energy expenditure and then doing it all over again until a good enough solution is found. Only then, the optimized application is sent to production. The concept of flexible *computation reuse regions* is very powerful since it makes the application more domain-independent while also optimizing the granularity of the reuse for any application instance. Their automated

software cache configuration is also interesting since any *memoization*-based technique is heavily reliant on its size and performance.

Unfortunately [38] do not specify the cost of profiling (since for the test environment the typical input traces of the selected benchmarks were already available), nor the cost of configuring the *computation reuse regions* and the software cache. Regardless, this approach, while presenting the concepts of flexible granularity and automatic software cache configuration / optimization, cannot be recommended for large-scale workflow execution given its unknown-cost training step. Also, in order to distribute the computation reuse, the software cache used by it needs to be re-thought to be compatible with this paradigm.

It is brought to our attention on [26] the cost of two-party secure-function evaluation (SFE) and the tendency to offload these operations from resource-constrained devices to outside servers. In order to reduce the computational cost of these SFE operations as well as bandwidth requirements, a system on which state is retained as to later be reused was implemented. The reusable encrypted values can be used by a number of clients on a distributed setting, originating from a centralized server node that implements a *memoization* buffer.

Although [26] is the first approach to enable computation reuse to be done in a distributed environment, the encrypted values buffer is a bottleneck for the approach scalability. In order to remove this bottleneck, the buffer can be distributed among server nodes, which has as a consequence either (i) the buffers are coherent, and as such the servers need to keep trading messages to enforce it, or (ii) the buffers are not coherent and thus the reuse potential is reduced. Finally, this approach is only partially applicable for different application domains since the granularity of the reused tasks must be rather coarse in order to achieve good speedups. This happens because the of the big overhead of reusing encrypted values.

Approach [17] also works with distributable reusable values, but this time with bioinformatics applications, which are known to be computationally expensive. The granularity of reusable tasks is even coarser, being able to perform full end-to-end reuse of workflows. When comparing with [26], [17] has the same limitations given its *memoization*-based approach.

On [18] Santos and Santos propose the use of a software-level runtime buffer system to cache and then reuse energy evaluations for predicting the native conformation of proteins. The domain-specific application relies on a genetic algorithm, and as such, their approach is tailored for this single application. A similar approach is the one of Yasoubi et al. [42], regarding the use of *memoization*-based computation reuse, optimized for a specific domain, which is neural networks on this case.

Yasoubi et al. [42] propose an offload hardware accelerator that uses clustered groups

of neurons that maximize the expected computation reuse when executing the application. It is worth noting that the clustering is done by a k-means algorithm on software level. The reusable tasks are hardware-level multiplication instructions that, given the multi-processing-unit (multi-PU) architecture, disable PUs that perform repetitive operations, thus reducing the power consumption.

The work of Connors and W.Hwu [7] exploit value locality through the combination of a hardware buffer, an instruction set for representing different-sized reusable computational tasks and a profile-guided compiler that groups instructions into reusable tasks as to optimize their granularity. This approach was implemented as a way to extend hardware-only-based reuse approaches while solving the limitation of instruction-level reusable tasks granularity. Again, the use of dynamically-sized reusable tasks makes the approach more flexible to different domains of applications while optimizing the reusable tasks granularity for each application instance. However, in order to implement this feature the approach on [7] limits itself by needing a complex hardware and compiler implementation and profiling information on the domain-specific application.

Álvarez et al. [2] focus on reducing the power consumption of low-end and/or mobile devices by applying computation reuse on multimedia applications. This is done by exploiting the fault-tolerance of multimedia floating-point operations at hardware-level to reuse tasks that are similar enough, thus increasing the amount of attainable computation reuse. Nonetheless, this "similar enough" strategy limits the usability of this approach to multimedia applications, or applications which have a large number of floating-point reusable operations. The same is true for approach [25], which in turn proposes a more generic implementation that was not tailored for multimedia applications.

The first analytic computation reuse method is presented by Xu et al., on [41]. On their work they propose a framework for Isogeometric Analysis (IGA) that reuse matrix calculations. The reuse operations were statically analyzed *a priori* and are specific of IGA, meaning that this approach, although having good speedups, cannot be applied for other application domains.

Lepak and Lipasti [23] propose reuse of memory load instructions. This is done through the characterization of value locality for memory instructions and the implementations of two reuse protocols for both uniprocessed and multiprocessed environments. For uniprocessed systems reuse can be attained by either analyzing the value locality of specific instructions (based on the program structure), or the locality of a particular memory address (message-passing locality). Furthermore, they define *silent stores* as stores operations that do not change the system state (i.e., the written value is the same as the one previously present on memory). Given some statistical analysis of how many *silent stores* are on selected benchmarks, they set an ideal maximum reuse possible to be achieved

and, through their proposed protocols, aim to get as close as possible to these values.

Since none of the previous applications is neither compatible or flexible enough to work on the large scale bioinformatics workflows application domain, this work proposes a novel approach to computation reuse. The proposed approach should work with software-level reuse, since it is being implemented on top of the RTF. Also, given that this application is supposed to be executed on a large-scale cluster environment, hardware-based approaches are impractical. Moreover, the runtime system must be light in order to execute on a large-scale distributed environment, thus making the use of *memoization* techniques impractical. Given that the application uses hierarchical workflows, any applications of other domains need to be converted to workflows in order to be executed by our approach, slightly impacting the application domain flexibility. Finally, computation reuse will be achieved by a static analytic analysis of reuse before the execution of any task, thus removing any distribution limitations as long as the reuse analysis can be performed quickly.

Capítulo 3

Multi-Level Computation Reuse

This work has as its main goal the development of Sensibility Analysis (SA) optimizations through multi-level computation reuse. This chapter analyzes computation reuse and then describes improvements made to the Region Templates Framework (RTF), which were implemented in order to enable the use of multi-level computation reuse. After that, the new computation reuse approaches are described, along with their advantages and disadvantages.

The SA studies and components that were developed and integrated into the RTF are illustrated in Figure 3.1. An SA study in this framework starts with the definition of a given workflow, the parameters to be studied, and the input data. The workflow is then instantiated and executed efficiently in RT using parameters values selected by the SA method. The output of the workflow is compared using a metric selected by the user to

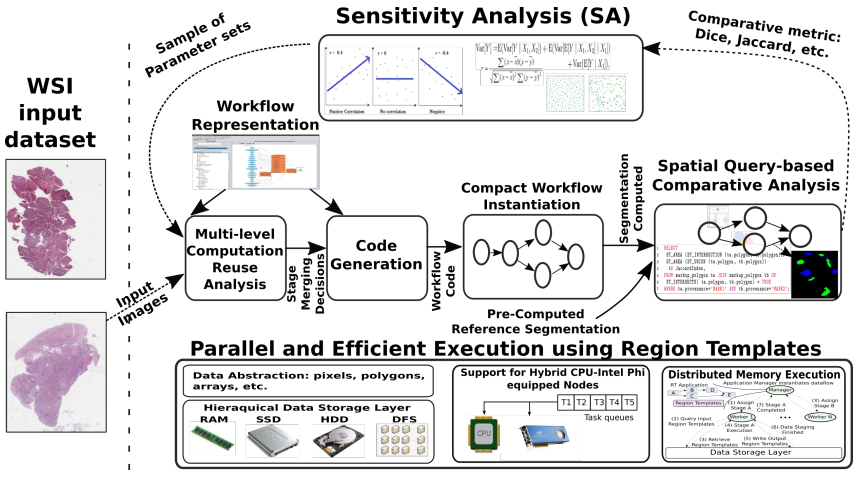


Figura 3.1: The parameter study framework. A SA method selects parameters of the analysis workflow, which is executed on a parallel machine. The workflow results are compared to a set of reference results to compute differences in the output. This process is repeated a set number of times (sample size) with varying input parameters' values.

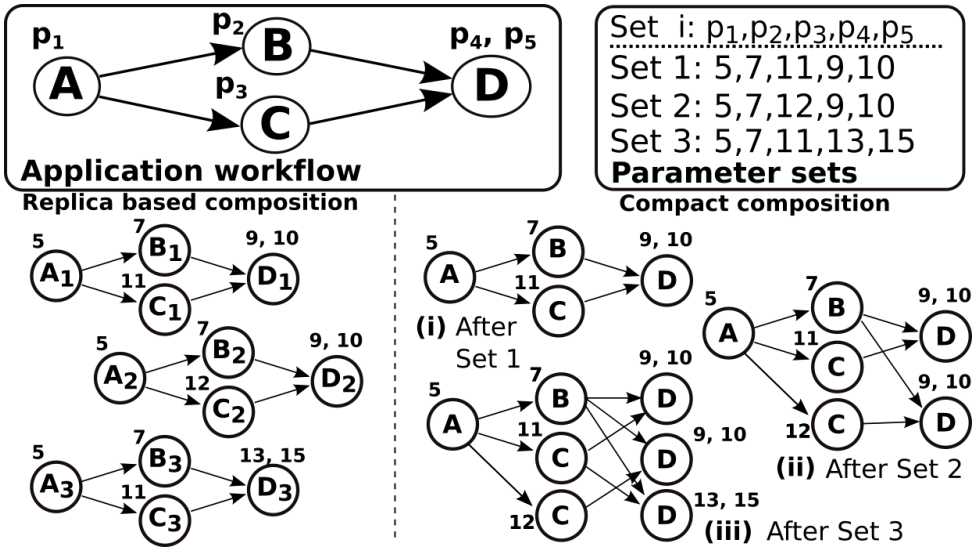


Figura 3.2: A comparison of a workflow generated with and without computation reuse. Image extracted from [36].

measure the difference between a reference segmentation result and the one computed by the workflow using the parameter set generated by the SA method. This process continues until the number of workflow runs does not achieve the sample size required by the SA. This sample size is effectively the number of times that the workflow will be instantiated and executed with different input parameters' values. The sample size is a way to limit the cost of the SA study while maintaining its significance and accuracy. This can be done by choosing a sample size that is big enough to have accurate results but so big that its cost is too high.

Computation reuse is achieved through the removal of repeated computation tasks. Figure 3.2 presents the comparison of a replica-based workflow generation, in which there is no reuse, and a compact composition, generated with maximal reuse. Given that we start generating a compact composition with no tasks on it, the first parameter set (Set 1, (i) in Figure 3.2) is added to the workflow in its entirety. The second parameter set (Set 2, (ii) in Figure 3.2), however, has the reuse opportunities of tasks A and B given they have the same input parameters values and input data. This results in the inclusion of only tasks C and D for parameter set 2 in the compact graph. With the current workflow state ((ii) on Figure 3.2), parameter set 3 presents reuse opportunities for tasks A, B and C, thus only needing to add task D with the parameter value 15 to the workflow. When comparing the workflow replica based composition with the compact composition we can notice a decrease on the number of executed tasks of approximately 41%, from 12 tasks to 7 tasks.

There are two computation reuse levels used on this work, (i) stage-level, on which coarse-grain computation tasks are reused, and (ii) task-level - with fine-grain tasks reused.

Coarse-grain computation reuse is significantly easier to implement than its fine-grained counterpart. However, the number of parameters that two coarse-grained merging candidates stages need to match for the reuse to take place is higher as when compared with fine-grain tasks.

3.1 Graphical User Interface and Code Generator

In this work a flexible task-based stage code generator was implemented to ease the process of developing RTF applications. This generator was created, together with a workflow generator graphical interface - with the purpose of making the RTF more accessible to domain-specific experts. Additionally, this code generator will simplify the application information gathering process, necessary for merging stages instances during the process of computation reuse.

The stage generator has as its input a stage descriptor file, formatted as XML, as shown in Figure 3.3. A stage is defined by its name, the external libraries it needs to call in order to execute the application domain transformations in each stage of the workflows, the necessary input arguments for its execution and the tasks it must execute. There are

```
{"name":"Segmentation",
"includes":"#include \"opencv2/opencv.hpp\"\n#include \"opencv2/gpu/gpu.hpp\"\n#include
\"HistologicalEntities.h\"",
"dr_args":[
        {"name":"normalized_rt", "type":"dr", "io":"input"},
        {"name":"segmented_rt", "type":"dr", "io":"output"}
    ],
"tasks":[
        {"call":"::nscale::HistologicalEntities::segmentNucleiStg1",
            "args":[
                {"name":"normalized_rt", "type":"dr", "io":"input"},
                {"name":"blue", "type":"uchar"},
                {"name":"green", "type":"uchar"},
                {"name":"red", "type":"uchar"},
                {"name":"T1", "type":"double"},
                {"name":"T2", "type":"double"}
            ],
            "intertask_args":[
                {"name":"bgr", "type":"mat_vect", "io":"output"},
                {"name":"rbc", "type":"mat", "io":"output"}
            ]
        },
        {"call":"::nscale::HistologicalEntities::segmentNucleiStg2",
            "args":[
                {"name":"reconConnectivity", "type":"int"}
            ],
            "intertask_args":[
                {"name":"bgr", "type":"mat_vect", "io":"input"},
                {"name":"rbc", "type":"mat", "io":"forward"},
                {"name":"rc", "type":"mat", "io":"output"},
                {"name":"rc_recon", "type":"mat", "io":"output"},
                {"name":"rc_open", "type":"mat", "io":"output"}
            ]
        },

        [...]

        {"call":"::nscale::HistologicalEntities::segmentNucleiStg7",
            "args":[
                {"name":"segmented_rt", "type":"dr", "io":"output"},
                {"name":"minSizeSeg", "type":"int"},
                {"name":"maxSizeSeg", "type":"int"},
                {"name":"fillHolesConnectivity", "type":"int"}
            ],
            "intertask_args":[
                {"name":"seg_nonoverlap", "type":"mat", "io":"input"}
            ]
        }
    ]
}
```

Figura 3.3: An example stage descriptor XML file.

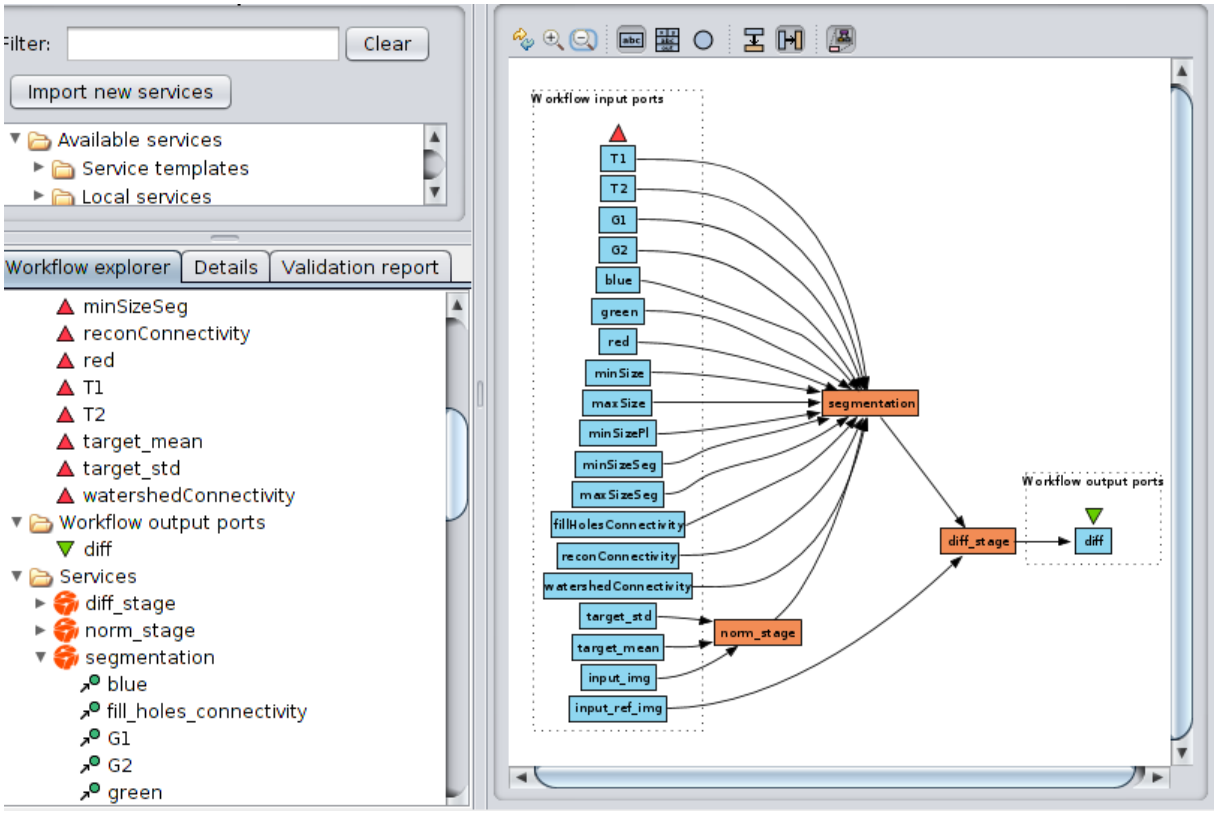


Figura 3.4: The example workflow described with the Taverna Workbench.

two kinds of inputs, the arguments and the Region Templates (RT). The arguments are constant inputs, which are varied by the given SA method and represent the application input parameter values. The RT is the data structure provided by the RTF for inter-stage and inter-task communication. As seen on the example descriptor file, only the RT inputs are explicitly written, while the remaining arguments can be inferred from the tasks descriptions.

Every stage is comprised of tasks, described by (i) the external call to the library of operations implemented by the user and (ii) its arguments. On Figure 3.3 the call for the first task is *segmentNucleiStg1* from the external library *nscale*. The arguments can be one of two types, (i) constant input arguments (args), defined by the SA application or (ii) intertask arguments (intertask_args), which are produced/consumed for/by a fine-grain task.

With task-based stages generated, the user can instantiate workflows using the newly generated stages. As with tasks, the RTF did not support a flexible, non-compiled solution for generating workflows, being these workflows hardcoded into the RTF. The solution implemented on this work was to use the Taverna Workbench tool [40] as a graphical interface for producing workflows and implement a parser for the generated Taverna file. An example workflow on the Taverna Workbench is displayed on Figure 3.4.

3.2 Stage-Level Merging

The stage level merging needs to identify and remove common stage instances and build a compact representation of the workflow, as presented in Algorithm 1. The algorithm receives the application directed workflow graph (appGraph) and parameter sets to be tested as input (parSets) and outputs the compact graph (comGraph). It iterates over each parameter set (lines 3-5) to instantiate a replica of the application workflow graph with parameters from *set*. It then calls MergeGraph to merge the replica to the compact representation.

The MergeGraph procedure walks simultaneously in an application workflow graph instance and in the compact representation. If a path in the application workflow graph instance is not found in the latter, it is added to the compact graph. The MergeGraph

Algorithm 1 Compact Graph Construction

```
 1: Input: appGraph; parSets;
 2: Output: comGraph;
 3: for each set ∈ parSets do
 4:     appGraphInst = instantiateAppGraph(set);
 5:     MergeGraph(appGraphInst.root, comGraph.root);
 6: end for
 7: procedure MergeGraph(appVer, comVer)
 8:     for each v ∈ appVer.children do
 9:         if (v' ← find(v, comVer.children)) then
10:             MergeGraph(v, v');
11:         else
12:             if ((v' ← PendingVer.find(v))==∅) then
13:                 v' ← clone(v)
14:                 v'.depsSolved ← 1
15:                 comVer.children.add(v')
16:                 if v'.deps ≥ 1 then
17:                     PendingVer.insert(v')
18:                 end if
19:                 MergeGraph(v, v');
20:             else
21:                 comVer.children.add(v')
22:                 v'.depsSolved ← v'.depsSolved+1
23:                 if v'.depsSolved == v'.deps then
24:                     PendingVer.remove(v')
25:                 end if
26:                 MergeGraph(v, v')
27:             end if
28:         end if
29:     end for
30: end procedure
```

procedure receives the current set of vertices in the application workflow ($appVer$) and in the compact graph ($comVer$) as a parameter and, for each child vertex of the $appVer$, finds a corresponding vertex in the children of $comVer$. Each vertex in the graph has a property called *deps*, which refers to its number of dependencies. The find step considers the name of a stage and the parameters used by the stage. If a vertex is found, the path already exists, and the same procedure is called recursively to merge sub-graphs starting with the matched vertices (lines 9-10). When a corresponding vertex is not found in the compact graph, there are two cases to be considered (lines 11-26). In the first one, the searched node does not exist in *comGraph*. The node is created and added to the compact graph (lines 12-18). To check if this is the case, the algorithm verifies if the node (v) has not been already created and added to *comGraph* as a result of processing another path of the application workflow that leads to v. This occurs for nodes with multiple dependencies, e.g., D in Figure 3.2. If the path (A,B,D) is first merged to the compact graph, when C is processed, it should not create another instance of D. Instead, the existing one should be added to the children list as the algorithm does in the second case (lines 21-25). The $PendingVer$ data structure is used as a look-up table to store such nodes with multiple dependencies during graph merging. This algorithm makes k calls to MergeGraph for each *appGraphInst* to be merged, where k is the number of stages of the workflow. The cost of each call is dominated by the $find$ operation in the $comVer.children$. The *children* will have a size of up to n or $|parSets|$ in the worst case. By using a hash table to implement children, the find is $\mathcal{O}(1)$. Thus, the insertion of n instances of the workflow in the compact graph is $\mathcal{O}(kn)$.

3.3 Task-Level Merging

On the previous section coarse-grain reuse was implemented through a stage-level merging algorithm. This approach can by itself attain good speedups for the workflow used on this work. However, due to the granularity of the stages there is still many reuse opportunities which are wasted since they are not visible or even achievable on stage-level. These opportunities are visible though on task-level, through what we define as fine-grain reuse. This reuse can be achieved by merging stages together and removing the repeated tasks, through what we call task-level merging. Merging at task-level, unlike stage-level, has some limitations due to the way stages and tasks are implemented on the RTF. Tasks are a finer-grain computational job, intended to be small activities. Although stages can be executed on distinct computing nodes, tasks cannot, since it would not make sense to distribute such small tasks which communication overhead over the nodes network would most likely outweigh the task cost itself.

With these peculiarities in mind, before we implement any fine-grain merging algorithm we must first address some limitations on excessive fine-grain reuse. When excessive task-level merging is performed the joint number of parameters and variables of a merged stage, containing a large number of tasks, may not fit on the system memory These variables are most of the times intermediate data that is passed between tasks, also including intermediate images, which are rather large for the purpose of this work. Also, it is possible for all stages to be merged in a number smaller than the number of available nodes, hence making some of the available resources idle. Both these problems can be solved by limiting the maximum number of stages that can be merged (bucket size). This limit is defined here as $MaxBucketSize$. Another way to enforce memory restriction is to limit the maximum number of tasks per group of merged stages (buckets). This limit is the $MaxBuckets$.

3.3.1 Naïve Algorithm

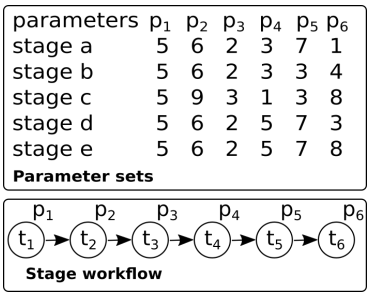
In the interest of better understanding the task-level merging problem, a naïve algorithm was implemented to serve as a baseline for our analysis. This simplified algorithm groups $MaxBucketSize$ stages in buckets and attempt to merge all stages of each bucket among themselves. This was achieved by sequentially grouping the first $MaxBucketSize$ stages into buckets, until there are no more stages to be merged.

Although this simple solution was quickly implemented and has a linear algorithmic complexity its reuse efficiency is, however, highly dependent on the stages ordering. For instance, if similar stages were to be generated close together a greater amount of reusable computation is more likely to exist.

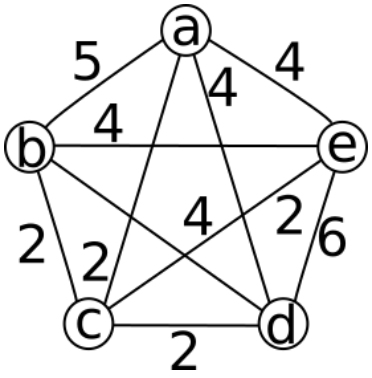
3.3.2 Smart Cut Algorithm (SCA)

Another strategy to create buckets of stages to be merged that was investigated is through the use of a graph based representation (see Figure 3.5). A representation for this could be done using fully-connected undirected graphs on which the stage instances are the nodes and each edge is the degree of reuse between two stage instances (Figure 3.5b). By degree of reuse we mean the number of tasks that would be reused if the two stages are merged. With this perspective we would need only to partition this graph in subgraphs, maximizing the reuse degree of all subgraphs. This is a well-known problem, called min-cut [34].

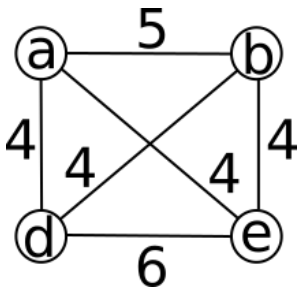
Although there are many variations for the min-cut problem [34, 13], we define here a min-cut algorithm as one that takes an undirected graph and performs a 2-cut (i.e., cut the graph in two subgraphs) operation, minimizing the sum of the cut edges weight.



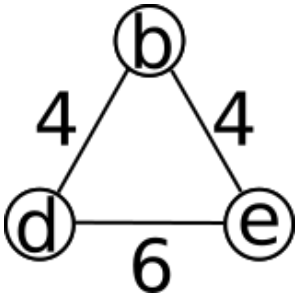
(a) Example application.



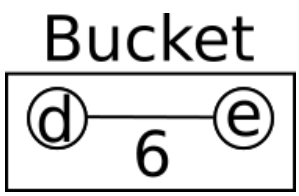
(b) Initial graph of instance example.



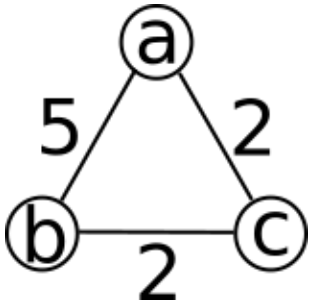
(c) First cut is performed, removing node c.



(d) After the next cut node a is removed.



(e) After final cut of node b $MaxBucketSize$ sized subgraph is found.



(f) The cutting starts over with the remaining nodes.

Figura 3.5: An example on which SCA executes on 5 instances of a workflow application of 6 tasks, with $MaxBucketSize = 2$.

This 2-cut operation was selected because of its flexibility and computational complexity. First, the recursive use of 2-cuts can break a graph in any number of subgraphs. Moreover, k-cut algorithms are not only more computationally intensive than 2-cut algorithms, but also have no guarantees for the balancement of the subgraphs (e.g., for $k = 5$ on a graph with 10 nodes one possible solution is 4 subgraphs with 1 node each and 1 subgraph with 6 nodes). As such, we can implement a simple k-cut balanced algorithm by performing 2-cut operations on the most expensive graph/subgraph until a stopping condition is reached (e.g., number of subgraphs is reached, number of nodes per subgraph is reached). With all these considerations only 2-cut operations are used on the proposed algorithm.

Figure 3.5 demonstrate a way to group stages into buckets using 2-cut operations. First, the fully-connected graph in Figure 3.5b is generated given the stage instances of Figure 3.5a. Figure 3.5c shows the result of the first 2-cut operation, on which the subgraph containing only the node c is found to be the one least related to the subgraph with the remaining nodes. This is similar to the state that c is the "least reusable" stage among all other stages (i.e., the stage which, if selected for merging, would have highest

Algorithm 2 Smart Cut Algorithm

```
1: Input: stages; MaxBucketSize;
2: Output: bucketList;
3: while |stages| > 0 do
4:     {s1,s2} ← 2CUT(stages)
5:     while |s1| > MaxBucketSize do
6:         {s1,s2} ← 2CUT(s1)
7:     end while
8:     bucketList.add(s1)
9:     for each s ∈ s1 do
10:        stages.remove(s)
11:    end for
12: end while
```

computational cost). Next, nodes a and b are removed until a bucket of size 2 is reached (see Figures 3.5c and 3.5d). The previously removed nodes (a, b and c) are then put together (Figure 3.5f) and the same cutting algorithm starts over. This process is then repeated until all stages are grouped into buckets.

With this procedure in mind Algorithm 2 was designed. This algorithm performs successive 2-cut operations on the graphs to divide it into unconnected subgraphs that fit in a bucket. The cuts are performed such that the amount of reuse lost with a cut is minimized. In more detail, the partition process starts by dividing the graph into 2 subgraphs (s1 and s2) using a minimum cut algorithm [34] (line 4). Still, after the cut, both subgraphs may have more than $MaxBucketSize$ vertices. In this case, another cut is applied in the subgraph with the largest number of stages (lines 5-7), and this is repeated until a viable subgraph (number of stages $\leq MaxBucketSize$) is found. When this occurs, the viable subgraph is removed from the original graph (lines 8-11), and the full process is repeated until the graphs with stage instances yet not assigned to a bucket can fit in one.

The number of cuts necessary to compute a single viable subgraph of n stages is $\mathcal{O}(n)$ in the worst case. This occurs when each cut returns a subgraph with only one stage and another subgraph with the remaining nodes. The cut then needs to be recomputed – about $n - MaxBucketSize$) times – on the largest subgraph until a viable subgraph is found. Also, in the worst case, all viable subgraphs would have have $MaxBucketSize$ stages and, as such, up to $n/MaxBucketSize$ buckets could be created. Therefore, the algorithm will perform $\mathcal{O}(n^2)$ cuts in the worst case to create all buckets. In our implementation, the min-cut is computed using a Fibonacci heap [34] to speed up the algorithm, making each cut $\mathcal{O}(E + V\ log\ V)$. Since the graph used is fully connected, the complexity of a single cut in our case is $\mathcal{O}(n^2)$ and, as consequence, the full SCA is $\mathcal{O}(n^4)$. Although the SCA computes good reuse solutions, its use in practice is limited because of the computational

complexity. This motivated the proposal of the strategy described in Section 3.3.3.

3.3.3 Reuse Tree Merging Algorithm (RTMA)

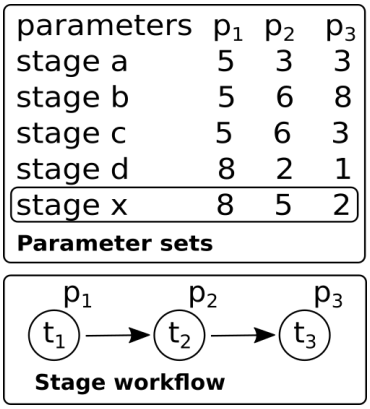
Still on graphs, a natural way to display hierarchical structures is with trees. Using tasks as nodes on this tree, subtrees with the same parent node indicates that all child task nodes of said parent node use its output. As such, if we constructed a tree with several stages, we are able to easily see the reuse opportunities, lying in the nodes with more than one child node. Moreover, each level of the tree would represent a given task, which can be instantiated with different parameters sets.

Detailing this structure, each level of the tree represents a task, and if a stage s shares a parent node on level k with s', this implies that all tasks from 1 to k are the same for both stages, an thus reusable among themselves (i.e., same computational task with the same inputs). This structure is defined as a Reuse Tree, with every node being defined by its level (or height), its parent, its children and a reference to the stage responsible for its generation.

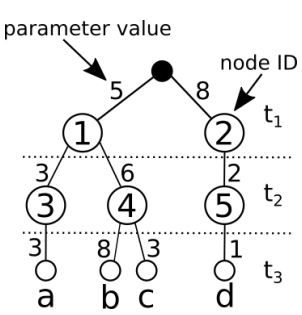
Reuse-Tree Generation

On a SA example we have a workflow w that is instantiated n times with different parameters (w_1, w_2, ..., w_n). Each workflow w_i is composed by m stages s_{ij} with $i \in [1, n]$ and $j \in [1, m]$. A reuse-tree is then generated for each j-th stage level. The reuse-tree for a given stage level can be generated by iteratively inserting one stage instance after the other on the reuse-tree. Initially, a stage is represented as a tree on which every node has a single child and each node represents a task instantiation for that stage. Furthermore, any given node has as its parent a task that it is dependent on. Every stage is inserted one task node at a time. If, for a given task node, there already exists on the tree another node representing the same task with the same parameter inputs, said task node is not created, but instead the insertion process carry on from the equivalent node, characterizing task reuse.

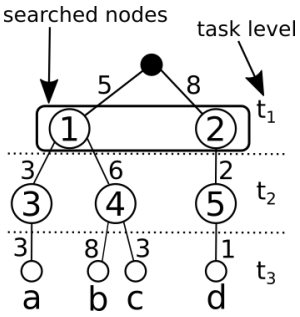
As an example, Figure 3.6 demonstrates the insertion of a stage (stage x) with the stage workflow and the parameters of each stage instance defined in Figure 3.6a, and the starting reuse tree in Figure 3.6b. Starting at the root node, its children (1 and 2) are searched for reuse opportunities for the first task (Figure 3.6c). Since node 2 represents all stages whose task 1 has as its input $p_1 = 8$ the first task of x can be reused through it. The search for reuse of the second task is then performed on the children of node 2 (Figure 3.6d). Since node's 2 only child, node 5, cannot be reused for stage x's second task (values for p_2 of stages d and x are different), a new node representing this non-reusable



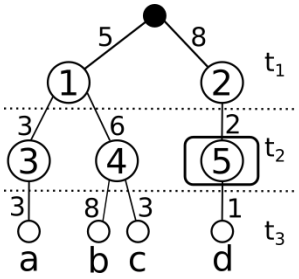
(a) Example application.



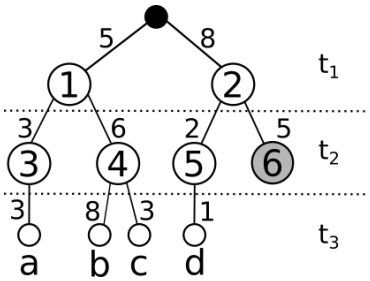
(b) Initial reuse tree for the instance example.



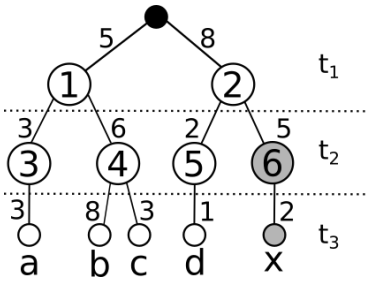
(c) Searching for reuse on the first task.



(d) Searching for reuse on the second task.



(e) Inserting a new node, 6.



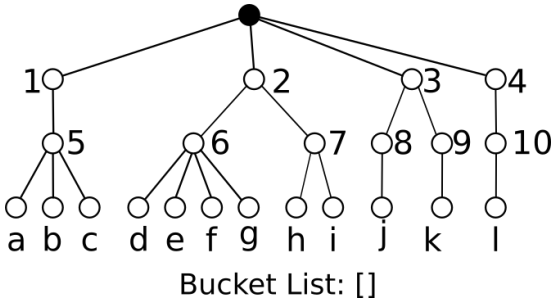
(f) Inserting the leaf node x.

Figura 3.6: An example where node x is inserted on the existing reuse tree. Figure 3.6a defines the tasks of which each stage is composed by and presents the parameters' values for each stage instance.

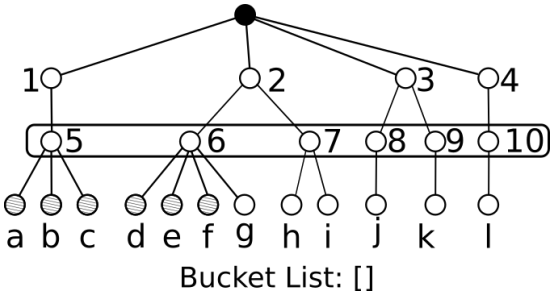
task is created (node 6) as shown in Figure 3.6e. Finally, since node 6 is new, there are no more reuse opportunities from it, thereby, a single child node must be created for each of the remaining non-reusable tasks (Figure 3.6f).

The Merging Implementation

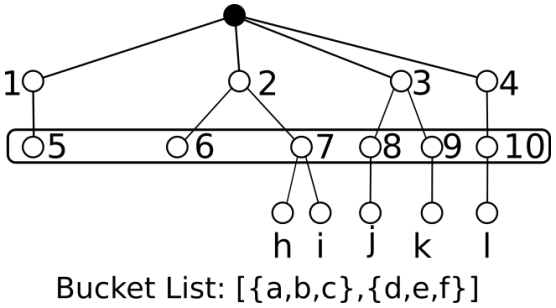
In order for a merging algorithm to be implemented on top of the Reuse Tree structure we must take advantage of its hierarchical characteristics. Given that we want to bundle together buckets of stages of exactly $MaxBucketSize$ stages we must start with the deepest stages and move up. Figure 3.7a shows an example of a Reuse Tree with 12 stages and 3 tasks each. Stages a, b and c have two out of three reusable tasks, and as such, given a $MaxBucketSize = 3$, should be put together in the same bucket. Meanwhile, stages d through i have one out of three reusable tasks. To maximize the reuse, stages d, e, f and g should be together, as should stages h and i. Since $MaxBucketSize = 3$, only 3 stages out of d, e, f and g can be put together, not mattering which 3 stages. This merger is seen in Figure 3.7c. After the merger of d, e and f, stage g is left alone, having



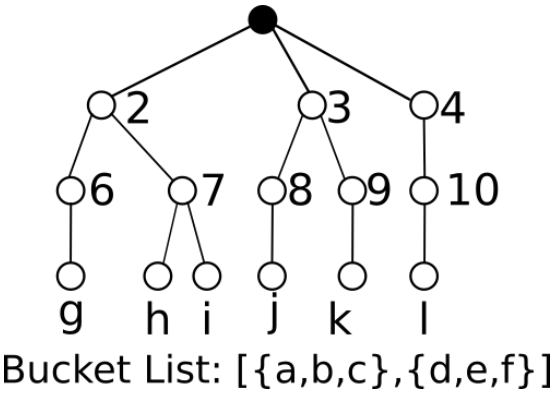
(a) Initial reuse tree.



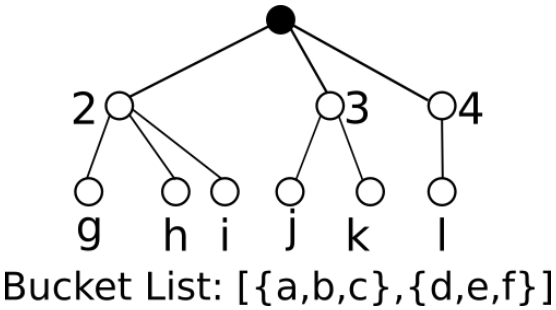
(b) Reuse Tree after select procedure.



(c) Reuse Tree after the selected mergeable leaf nodes are pruned and added to the bucket list.



(d) Reuse tree after the childless parents are recursively removed.



(e) Reuse tree after move-up procedure.

Figura 3.7: An example of Reuse Tree based merging with MaxBucketSize 3. The merged stages of each step are shown below the tree on the bucket list.

as its best option, reuse-wise, to be put together with h and i. As such, it is visible that the merging should happen in a bottom-up fashion.

The Reuse Tree Merging Algorithm (RTMA), listed on Algorithm 3, was implemented in three steps, (i) bucket candidates selection (line 6), (ii) tree pruning (line 7) and (iii) move-up operation (line 9), which are performed iteratively until the whole tree is consumed. If at the end of the main loop (line 5) there are still any non mergeable stages, those will be converted to one-stage buckets (lines 12-13) and then inserted on the final solution (line 14).

Algorithm 3 Reuse-Tree Merging Algorithm (RTMA)

```
1: Input: stages; maxBucketSize;
2: Output: bucketList;
3: bucketList ← ∅;
4: rTree ← GenerateReuseTree(stages)
5: while rTree.height > 2 do
6:     leafsPList ← GenerateLeafsParentList(rTree)
7:     newBuckets ← PruneLeafLevel(rTree, leafsPList, maxBucketSize)
8:     bucketList ← bucketList ∪ newBuckets
9:     MoveReuseTreeUp(reuseTree, leafsPList)
10: end while
11: while rTree.root.children ≠ ∅ do
12:     newBucket ← ∅
13:     newBucket.add(removeFirstChildren(rTree.root.children));
14:     bucketList ← bucketList ∪ newBucket
15: end while
16: return bucketList
```

The first step of the algorithm (Algorithm 3, line 6) is to get a list of all parents of leaf nodes. In Figure 3.7c we can see the selected parents (5-10). With the leaf's parents list the pruning step makes as many $MaxBucketSize$ sized buckets as possible and then remove them from the reuse tree. The procedure $PruneLeafLevel$ (line 7) attempts to make buckets for each leaf parent node. As stated before, the new buckets must have an exact size of $MaxBucketSize$, thereby, if the parent node does not have at least $MaxBucketSize$ children will not create a bucket with them. Given that the parent has enough children, a number of $MaxBucketSize$ children will be bundled together as a new bucket to later be added to the solution pool. On Figure 3.7c the two formed buckets are shown: $\{a, b, c\}$ and $\{d, e, f\}$. Each time a leaf node is added to the current new bucket, it is then removed from the parent children list, and as a consequence, removed from the tree, as seen on Figure 3.7d.

If a parent node ends up grouping all of its children in buckets, it must be removed from the tree (node 5 on Figure 3.7c). This process is performed recursively by removing the given childless parent node and then checking if the removal of the current parent also makes its parent childless. If this is the case the parent node removal must continue on its parent (node 1 of Figure 3.7c is also removed, as seen on Figure 3.7d).

The final step of merging is to move the leaf nodes up one level in order to enable the creation of new buckets. The operation $MoveReuseTreeUp$ (Algorithm 3, line 9) is done by taking each of the previously selected parent nodes and moving all of its children to its parent's children list (e.g., nodes g, h and i of Figure 3.7d are moved to parent node 2, as seen on Figure 3.7e). After that, the current node is remove from its parent (e.g., nodes 6 and 7 of Figure 3.7d are removed from parent node 2, as seen on Figure 3.7e). After all nodes from the parent list are removed and its children are moved up the tree

height is updated (line 6).

Algorithmic Complexity

Assuming an empty tree, the GenerateReuseTree performs the insertion of n stage instances with k tasks each. In the worst case of a stage insertion there is no reuse whatsoever, resulting in the creation of the maximum number of nodes. In this case, given that $m < n$ stage instances were already added, the next stage will perform m comparisons, looking for a reuse opportunity. After no opportunities are found k nodes will be created. This results in kn new nodes generated and $n(n+1)/2$ nodes traversed in total, and as a consequence, GenerateReuseTree is $\mathcal{O}(n^2)$.

The $n(n+1)/2$ nodes traversed is due to a linear search for reusable tasks on a given level with m stages instances. It is possible to further reduce this cost by performing this reuse check on a hash table on which the key is a combination of all parameters' values. By doing this hash table search the cost of each insertion will be $\mathcal{O}(1)$, thus resulting in the overall time complexity of $\mathcal{O}(kn)$.

The analysis of the actual merging algorithm can be split in the three operations performed on the reuse tree. Starting with the select operation, on the worst case, there will be one child per stage (i.e., no reuse on the first task), resulting in n nodes visited. On this case, the number of children of each node beyond the first level will be one, resulting in $k-2$ extra nodes visited. As a result we have that that $GenerateLeafsParentList$ runs in $\mathcal{O}(nk)$ per iteration, or $\mathcal{O}(nk^2)$, for there are exactly $k-1$ iterations.

For the pruning step the most expensive operation is the one of adding a stage to a solution bucket. Knowing that the exact number of bucket insertions must be at most n for the whole merging algorithm, we get the complexity $O(n)$ for all iterations of the pruning step.

At last, the complexity of the move-up step will be calculated by the amount of times a leaf node is moved from the current node child list to its parent. Independently to the structure of the tree, given that it has n leaf nodes, all of them will be moved once per move-up operation. Given that there are exactly $k-1$ iterations, we have $\mathcal{O}(nk)$.

The RTMA complexity is then dominated by tree generation algorithm since it is $\mathcal{O}(n^2)$, versus the joint complexity of the other three steps, $\mathcal{O}(nk^2 + nk + n)$. This happens because $n \gg k$ by the order of hundreds to thousands times greater. With such time complexity, the RTMA is expected to be scalable enough in order to be a viable solution. Furthermore, if the hash table optimization of GenerateReuseTree is implemented, then the time complexity becomes $\mathcal{O}(nk^2)$.

3.3.4 Task-Balanced Reuse-Tree Merging Algorithm (TRMA)

Given the nature of the chosen SA application and its scale (both computational cost and used resources wise), if the scale of resources is high enough, or the chosen SA method require a sample size small enough, the ratio of buckets per computing node (or core) may become low. This may lead to unbalancement, and thereafter performance degradation. This happens since the RTMA naturally reduces the parallelism of the application due to its grouping of stages. To solve this problem, a task-wise balanced version of RTMA was implemented, the Task-Balanced Reuse-Tree Merging Algorithm (TRMA). The TRMA will be presented in five parts. First a general idea of the unbalancement problem and how to solve it is presented. Then, algorithmic details are presented, followed by the complexity analysis of the TRMA. Finally, some optimizations are described along with a qualitative analysis of the achievable results of the TRMA.

General Idea

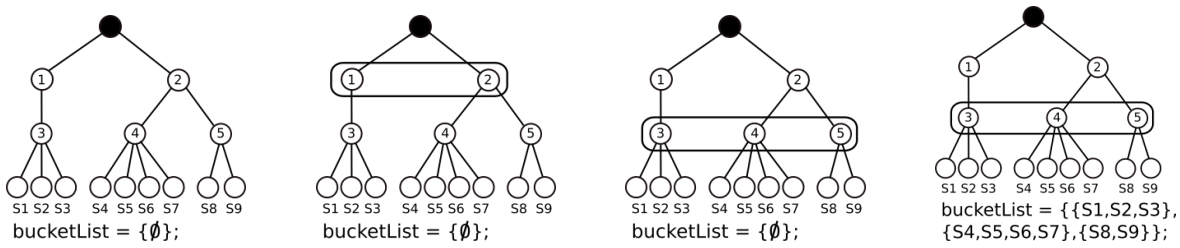
In more details, the RTMA balances its buckets stage-wise. This means that the buckets generated by it have similar (or most of the times, the same) number of stages. As such, buckets unbalancement comes from the difference on the number of tasks that two buckets with the same number of stages can have. This difference is a consequence of distinct reuse patterns on a reuse-tree structure, which in turn leads to different numbers of tasks for buckets with the same number of stages.

Given this unbalancement of stage-wise balanced buckets, the TRMA can be seen as an improvement of the RTMA on which task-wise balancement is enforced. In order to do so, the TRMA also uses the reuse-tree structure, while trying to achieve the best balancement for $MaxBuckets$ buckets. The change of the $MaxBucketSize$ parameter to $MaxBuckets$ helps the usability of the algorithm since the maximum number of buckets is a higher-level concept than the maximum number of stages per buckets.

The TRMA is implemented in three steps. On the first two, the $MaxBuckets$ number of buckets is achieved to then be balanced task-wise on a third step. The tree steps are defined as: Full-Merge, Fold-Merge and Balance.

Full Merge is the first attempt at achieving $MaxBuckets$ buckets. It is done by traversing the reuse-tree on a top-down fashion, attempting to find a task-level on which there are at least $MaxBuckets$ nodes. The full process can be seen on Figure 3.8, on which $MaxBuckets = 3$ is used. As seen, Figure 3.8a shows a simple initial reuse-tree. On the first level of tasks there are only two nodes (1 and 2), meaning that the next level should be searched (see Figure 3.8b). The next level has the exact number of tasks (nodes

2, 4 and 5) and, as such, the buckets can be generated by the leaf-nodes of the nodes on branches at this level (see Figure 3.8c). Finally, the buckets are generated on Figure 3.8d.



(a) Initial reuse-tree.

(b) Attempt of *Full-Merge* from root node results in two buckets.

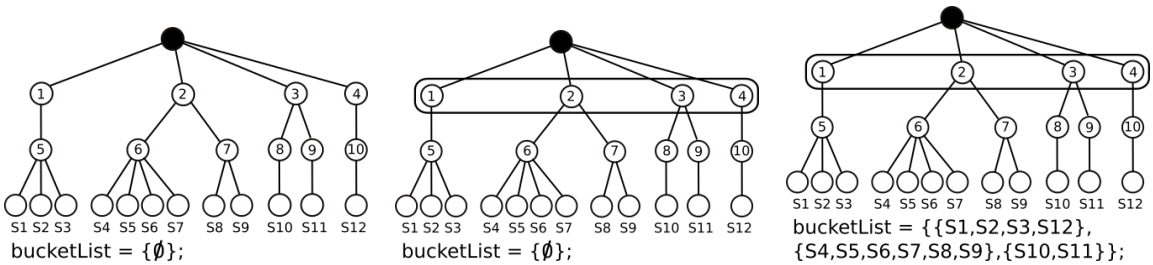
(c) Attempt of *Full-Merge* on the children of previous attempt results in three buckets.

(d) Merger is performed since the correct number of buckets was achieved.

Figura 3.8: Simple example of *Full-Merge* on which $MaxBuckets$ is 3 and the exact division os stages is reached.

However, there are cases on which a perfect number of $MaxBuckets$ cannot be achieved (see Figure 3.9). On this cases, the Full-Merge step brakes stages in a number of buckets greater than $MaxBuckets$ (see Figure 3.9b). The $MaxBuckets$ number of buckets is then achieved by the merging of $b - Mb$ buckets, with b being the current number of buckets and Mb the $MaxBuckets$ goal on the next step (see Figure 3.10).

Fold-Merge, as demonstrated on Figure 3.10, merges the buckets with the smallest cost in a fold-like operation. Given that the buckets are sorted by decreasing order, according to the cost (number of tasks), we can imagine that a line of these buckets is folded on a *folding pivot*, between Mb and $Mb + 1$ (see Figure 3.10), with Mb being the $MaxBuckets$ value. By doing this we are reducing the maximum bucket cost of the merged buckets, and thus reducing the unbalancement. It is important to notice that although the folding-fashion on which the buckets are merged is not necessary, its use



(a) Initial reuse-tree.

(b) Attempt of *Full-Merge* from root node results in four buckets.

(c) After *Fold-Merge* the correct number of buckets is reached.

Figura 3.9: Another example of *Full-Merge* and *Fold-Merge* on which $MaxBuckets$ is 3 and the exact division of stages cannot be reached by *Full-Merge*.

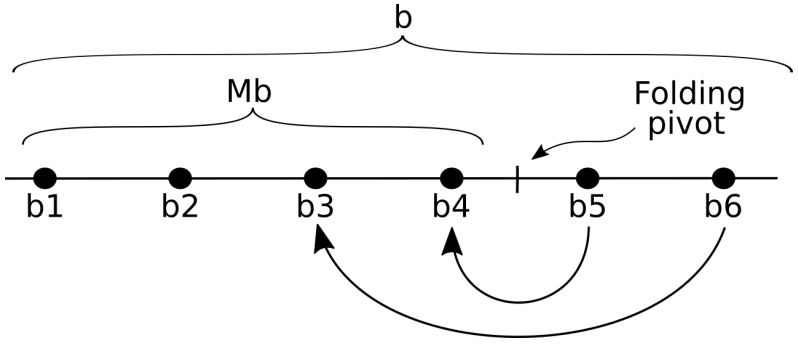


Figura 3.10: An example of a Fold-Merging of buckets b1-b6. Initially we start with $b = 6$ buckets, trying to achieve $Mb = 4$ buckets. In order to do so $b - Mb$ merger operations are performed. The task cost of the buckets follows the ordering $b1 \geq b2 \geq b3 \geq b4 \geq b5 \geq b6$.

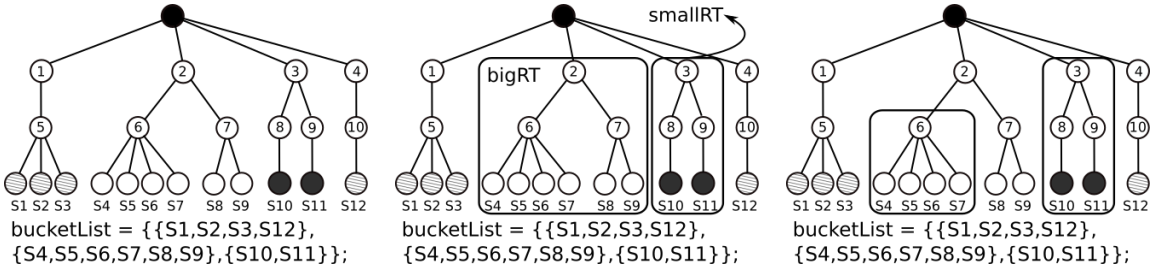
reduces the initial unbalance of the $MaxBuckets$ buckets, therefore reducing the cost of balancing these buckets.

On the example of Figure 3.9b four buckets are achieved through the Full-Merge procedure. As such, the Fold-Merge would then take the two last buckets and merge them together, resulting in the buckets of Figure 3.9c.

Balance is the last step of the TRMA. The balancement of buckets is done by searching for improvement operations on the reuse-tree grouped in the initial buckets outputted by the previous steps. An improvement operation is defined as a node of the reuse-tree (imp), which leaf nodes (or stages) can be sent from an original big reuse-tree node ($bigRT$) to a small one ($smallRT$), resulting in $TaskCost(smallRT \cup imp) - TaskCost(bigRT \setminus imp) \leq TaskCost(bigRT) - TaskCost(smallRT)$. $TaskCost(rt)$ is defined as the number of unique, not-reused, tasks present in the bucket (represented by a set of leaf nodes) of an arbitrary reuse-tree node rt.

However, there are cases on which an improvement is found but it cannot positively impact the overall solution of buckets. For example, the cost $TaskCost(smallRT \cup imp)$ may be the same as $TaskCost(bigRT)$, meaning that imp had some degree of reuse with $bigRT$ and thus, the cost of imp is greater on $smallRT$ than on $bigRT$. On this case this improvement will reduce the unbalancement but will not reduce the makespan of the application (i.e., the maximum number of tasks of all buckets). This is defined as a false improvement, or false balance operation, and since these can only increase the overall application cost, they will never be applied. As an example, two buckets $b1$ and $b2$ have initial costs 4 and 7, and thus 3 of unbalance. After a given improvement their costs are 7 and 5 respectively. The unbalance is now 2, which is "better", but the makespan is still the same.

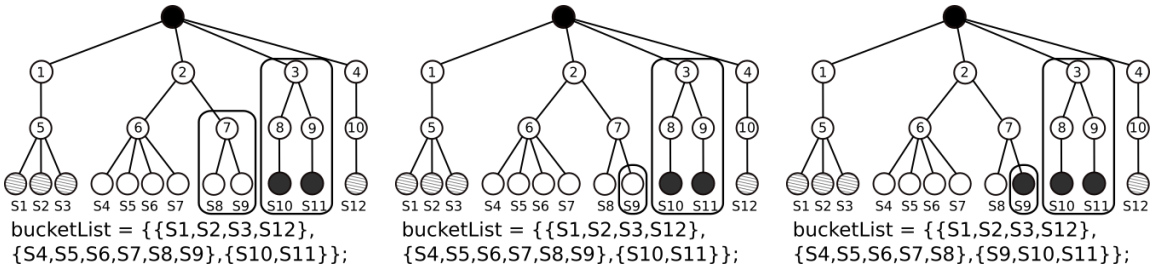
The full balance of a pair of buckets is achieved by attempting to find and applying valid improvements until there are no more improvements. Each improvement-search



(a) Initial reuse-tree with 3 buckets of costs 8, 9 and 5, respectively.

(b) Buckets of greatest and smallest cost values are selected for balance, with current unbalancement of 4 and max cost 9.

(c) The balance operation of sending node 6 children to smallRT is attempted, resulting in an unbalancement of 7.



(d) After the bad selection of node 6, balancement with node 7 is also done, resulting in an unbalancement of 3 but still having a max cost 9.

(e) As a final attempt, by sending node S9 to smallRT we have an unbalancement of 0 and max cost 8, making it a viable balancement.

(f) After the balancement operation of sending node S9 to smallRT we have the buckets with updated costs 8, 8 and 8.

Figura 3.11: An example of the *Balance* step on which there are 3 buckets to be balanced.

iteration is executed for a pair of the two ends of the current *bucketList*, which should be sorted in a non-ascending order with respect to cost (number of unique tasks). These are defined as *bigRT* and *smallRT*, and are the buckets with the greatest and smallest task costs respectively. If a single improvement attempt fails, then the balance step finishes. This single improvement attempt operation is defined as *SingleBalance*.

A *SingleBalance* operation consists in traversing the *bigRT* subtree in a breadth-first, bottom-up fashion in the search for a node that can be sent from *bigRT* to *smallRT*, characterizing an improvement. The reason for this traversal order is that lower nodes on the reuse-tree will have at most the same number of leaf nodes of its parent and thus, balancement on a finer-grain is performed earlier.

The full *Balance* process is exemplified on Figure 3.11. Starting with an initial reuse-tree of Figure 3.11a, the *bigRT* and *smallRT* buckets are selected, with task costs 9 and 5, respectively, and thus unbalancement of 4 (see Figure 3.11b). Several nodes are searched as improvement candidates. When trying an improvement operation of node 6, the resulting buckets $\{S8, S9\}$ and $\{S4, S5, S6, S7, S10, S11\}$ would have costs 4 and 11,

resulting in a new unbalance of 7, making this operation impracticable (see Figure 3.11c). Another alternative is the improvement of node 7. This results in buckets $\{S4, S5, S6, S7\}$ and $\{S8, S9, S10, S11\}$, and costs 6 and 9. This improvement operation unbalancement of 3 is better than the previous unbalancement of 4, but the maximum task cost, still remains at 9, meaning this is a false improvement and hence, an invalid balance operation (see Figure 3.11d). Finally, by applying the improvement of leaf-node $S9$ (which could be any of the nodes in the interval $[S4, S9]$) the resulting buckets would be $\{S4, S5, S6, S7, S8\}$ and $\{S9, S10, S11\}$, both with cost 8 and thus, 0 of unbalancement (see Figure 3.11e). Given that this last improvement operation was the best found, it is applied (see Figure 3.11f). Since it is impossible to improve an unbalance of 0, the next *SingleBalance* attempt will not find any valid improvement, consequently ending the *Balance* step.

Algorithmic Implementation Details

On this section the *Balance* and *SingleBalance* algorithms will be detailed since they are the most complex of all previously presented algorithms. Going through in a bottom-up fashion, *SingleBalance* is detailed in Algorithm 4.

The *SingleBalance* algorithm (see Algorithm 4) is divided into two parts, the recursion loop (lines 9-22) and the current level search loop (lines 23-29). Since the nodes are searched on a bottom-up breadth-first fashion, the first loop is responsible for recurring the *SingleBalance* operation on each of *bigRT* child nodes (lines 9-10). The stop-condition for this recursion is when an empty bigRT is passed to *SingleBalance*, thus returning an empty improvement.

If an improvement is found (lines 11-13) it is then set as the new current best improvement (lines 13-16). Finally, the second loop (lines 23-29) goes through the current level children attempting to find improvements (lines 24-28), after which, the best improvement is returned (line 30) if any was found, or an empty improvement if no solution was found (line 8).

The *Balance* algorithm (see Algorithm 5) is implemented by repeatedly attempting to find an improvement from *SingleBalance* (line 8) until either an invalid improvement is returned (false balancement) or an empty improvement is returned (line 10). If any of those conditions apply then the *Balance* algorithm ends and return the current state of *bucketList* (line 21). It is worth noting that the *bucketList* input must be a non-ascending ordered data structure, being this algorithm implemented with a C++ *multiset* container with the task cost of each bucket as their keys. The reasons for choosing this structure is threefold: (i) it is sorted on insertions (with insertion $\mathcal{O}(log(n))$), (ii) it has a direct access operation on the beginning and end of the list ($\mathcal{O}(1)$), and (iii) the keys are not unique.

Algorithm 4 Balancement algorithm for two tree nodes (*SingleBalance*)

```
 1: Input: currChildren; bigRT; smallRT; unbal;
 2: Output: improvement;
 3: while |currChildren| = 1 and | currChildren.first().children| > 0 do
 4:     currChildren ← currChildren.first()
 5: end while
 6: uniqueChildren ← ∅
 7: uniqueChildrenCosts ← ∅
 8: improvement ← ∅
 9: for each children c ∈ currChildren do
10:     recSol ← SingleBalance(c, smallRT, unbal)
11:     if recSol ≠ ∅ then
12:         recUnbal ← | TaskCost(bigRT \ recSol) - TaskCost(smallRT ∪ recSol) |
13:         if recUnbal < unbal then
14:             improvement ← recSol
15:             unbal ← recUnbal
16:         end if
17:     end if
18:     if TaskCost(c) ∉ uniqueChildrenCosts then
19:         uniqueChildrenCosts ← uniqueChildrenCosts ∪ TaskCost(c)
20:         uniqueChildren ← uniqueChildren ∪ c
21:     end if
22: end for
23: for each children c ∈ uniqueChildren do
24:     currUnbal ← | TaskCost(bigRT \ c) - TaskCost(smallRT ∪ c) |
25:     if currUnbal < unbal then
26:         unbal ← currUnbal
27:         improvement ← c
28:     end if
29: end for
30: return improvement
```

The first step of *Balance* is to select *bigRT* and *smallRT* (Algorithm 5, lines 5-6). The selection of *bigRT* is done by taking the first bucket of *bucketList*. For *smallRT* we can either select the last bucket of *bucketList* (i.e., one of the buckets with the smallest task cost) or search among the buckets of *bucketList* with the smallest cost for the one with the greatest reuse with *bigRT*. The later selection strategy can potentially result in more and better improvement opportunities available. The used approach was the last-bucket-selection with a full analysis on the impact of different *smallRT* selection methods discussed further ahead.

After selecting *bigRT* and *smallRT*, both are used on *SingleBalance* in order to search for an improvement on the current state of *bucketList* (Algorithm 5, line 8). The returned improvement, if returned, is then validated (lines 9-10). If no improvement is returned, or if the returned improvement is a false balancement then the algorithm finishes its execution and returns the *bucketList* as it currently is (lines 17-21). Otherwise, the improvement is applied to *bucketList* (lines 11-16) and the algorithm searches for another improvement. Given the necessity for the ordering of *bucketList*, *bigRT* and *smallRT*

Algorithm 5 The *Balance* step of the TRMA

```
 1: Input: bucketList;
 2: Output: bucketList;
 3: bucketList is a sorted data structure by descending cost (e.g., multiset)
 4: while true do
 5:     bigRT ← bucketList.first()
 6:     smallRT ← selectSmallRT(bucketList)
 7:     unbal ← TaskCost(bigRT) - TaskCost(smallRT)
 8:     improvement ← SingleBalance(bigRT.children, bigRT, smallRT, unbal)
 9:     newMksp ← Max(TaskCost(bigRT \ improvement), TaskCost(smallRT ∪ improvement))
10:     if improvement ≠ ∅ and newMksp < TaskCost(bigRT) then
11:         bucketList ← bucketList \ smallRT
12:         bucketList ← bucketList \ bigRT
13:         smallRT ← smallRT ∪ improvement
14:         bigRT ← bigRT \ improvement
15:         bucketList ← bucketList ∪ smallRT
16:         bucketList ← bucketList ∪ bigRT
17:     else
18:         break
19:     end if
20: end while
21: return bucketList
```

are removed from *bucketList* (lines 11-12), updated (lines 13-14) and then re-inserted on *bucketList* on the right position (lines 15-16).

Algorithmic Complexity

In order to calculate the computational complexity of the TRMA we must first define a worst-case scenario on which the number of improvement attempts is maximum. One of this cases is defined in Figure 3.12 with $\mathcal{O}(n)$ maximum improvement operations. This is the case on which $n/2-1$ of the $n/2$ buckets start with exactly one stage, and a single remaining bucket starts with $n-b+1$, with $b = MaxBuckets$. On this situation $n-b-1$ stages of the last bucket will be sent to another bucket on *SingleBalance* operations. Assuming that *SingleBalance* balances every pair of buckets with the minimum impact (improvements of exactly one stage), it will take $n-b-1$ balancement operation for all buckets to reach the final stable state of two stages per bucket. Thus, $\mathcal{O}(n)$ improvement operations.

For each improvement operation there is a selection step for $bigRT$ and $smallRT$, their update, and a *SingleBalance* call. The selection is done in $\mathcal{O}(1)$ for both subtrees since we are accessing the first and last elements of *bucketList* (Algorithm 5 lines 5-6). The update, which is comprised of two removal operations and two insertion operations are done in $\mathcal{O}(log(n))$ since *bucketList* is an ordered data structure based on trees (Algorithm 5 lines 11-16). For the *SingleBalance* call, exactly one balancement attempt is done for each traversed node on the worst case. Since for a graph with height k and n leaf nodes the

number of nodes is bounded by $\mathcal{O}(kn)$, we have a final complexity of $\mathcal{O}(n\ log(n) + kn^2)$. Also, given that $n \gg k$ the time complexity will be dominated by $\mathcal{O}(n^2)$.

Optimizations

It is possible to reduce the cost of *SingleBalance* through two optimizations, already implemented on Algorithm 4: (i) single child pruning and (ii) unique sibling selection.

If a reuse-tree node *rtn* is being visited by *SingleBalance*, and *rtn* has only a single child node rtn', then the improvement operation for both *rtn* and rtn' are the same. As such, we can prune *rtn* from the search by moving down the subtree until either a leaf node is reached or a reuse-tree node with more than one child is found. This is implemented on Algorithm 4, lines 3-5.

Furthermore, it is noticeable that any leaf node on the interval of S4-S9 of Figure 3.11e would result in the same balancement outcome (an unbalancement of 0 with all buckets with cost 8). As such, it would be interesting if we pruned all nodes that would result in the same outcome. This can be, and is, achieved by verifying both the number of children and the cost of two nodes. If both values are the same than we have similar (or non-unique) nodes, meaning that only one of the nodes must be searched. This strategy is currently implemented locally, meaning that only sibling nodes are verified, which can

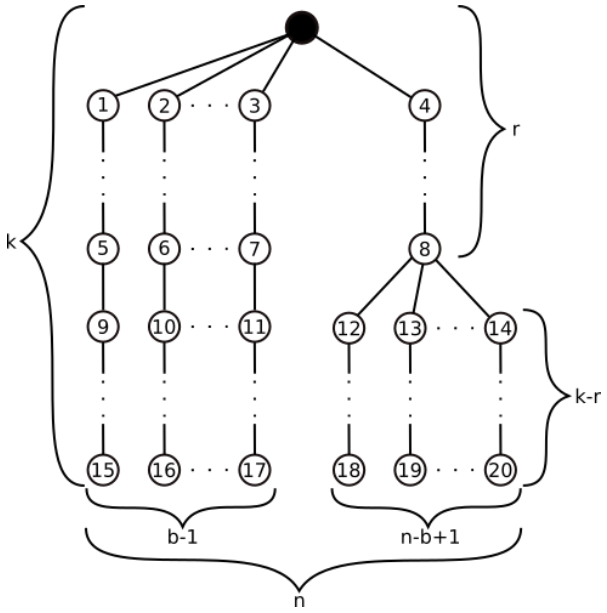


Figura 3.12: A general worst-case reuse-tree representation on which we have all n stages divided into b buckets. On this case we have $b-1$ buckets with exactly one stage, and thus cost k. Hence, the last bucket has $n-b+1$ stages. For this last bucket we assume the single and uniform reuse of the first r task, having no reuse for the remaining $k-r$ tasks. This is the worst-case for balancement applications.

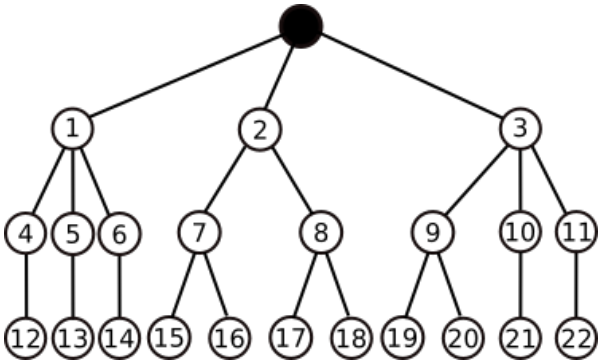
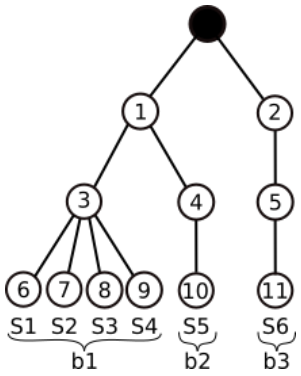


Figura 3.13: An example reuse-tree that can be used to illustrate possible prunable nodes. E.g., the use of nodes 4, 5, 6, or 10, 11 as an improvement attempt results in the same outcome (cost 3), making them interchangeable, as with nodes 7, 8 or 9 (cost 4), or nodes 12-22 (cost 3).

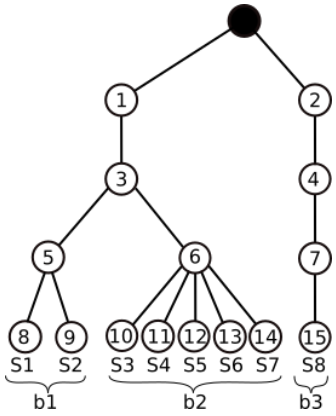
be seen using Figure 3.13. This implementation is present on Algorithm 5 on lines 18-21 and 23. For each child node traversed on *SingleBalance*, its task cost is calculated and, if it is unique (line 18), the matching child is added to a list of unique children (lines 19-20) to later be consumed (line 23).

By verifying prunable nodes locally it is meant that a node can only be pruned if the equivalent (repeated) search node is a sibling. On Figure 3.13 this means that when searching the children of node 1, only node 4 would be further searched, being node 12 searched afterwards, ignoring nodes 5 and 6. As the search progresses, on the search of the children of node 2, only the nodes 7 and 15 would also be searched. Finally, nodes 9 and 10, and their children would be searched as well. However, by keeping a list of searched nodes, uniquely ordered by their children count and overall cost, it is possible to extend this strategy to a global scope, thus removing the sibling-only prunable node restriction. While using local prune on the reuse-tree of Figure 3.13 would result in the search of 11 nodes (1, 4, 12, 2, 7, 15, 3, 9, 19, 10 and 21), a global prune scheme would result in 7 nodes searched (1, 4, 12, 2, 7, 15, 3).

In order to implement a global scope prune algorithm there is the need for both children count and overall cost metrics. Assuming that the reuse-tree of Figure 3.13 does not have the subtree of node 3, both subtrees of nodes 1 and 2 would have the same overall cost (6). Thus, by considering only the overall cost, subtree 2 would not be searched, resulting in the missed opportunity of balancing with subtree 7 which has a cost of 4 (from the root node), an impossible value to achieve with only subtree 1 (which can achieve a costs 3 with nodes 1, 4 and 12, or 5 with nodes 1, 4, 12, 5 and 13). Likewise, by only verifying the children count on a reuse-tree with only the subtrees of nodes 1 and 3 we would come to the same fallacy of pruning a necessary subtree (this time, subtree of node 3), hence, making it necessary the use of both metrics.



(a) Choosing the bucket with S5 results in a the premature finish of the TRMA since there is not a single improvement between buckets b1 and b3.



(b) Choosing to balance buckets b2 and b3 results in an unbalancement of 1 with max cost 8 (imp = {S7}), while balancing b2 and b1 results in an unbalancement of 0 with max cost 7 (imp = {S7}).

Figura 3.14: Two examples of bad selection of *smallRT* using the last-bucket strategy.

Discussion on Additional Optimizations and Limitations

A limiting factor of the TRMA is the *smallRT* selection strategy. By trying an improvement with only a single *smallRT* we may miss some better improvement opportunities, which may lead to better makespan values or even more balanced final results. The first possibility of improvement in the selection strategy arises from when two buckets of the same task cost can have different balancement outcomes when balancing with a given *bigRT*. This is exemplified on Figure 3.14a, where we have three buckets: $b1 = \{S1, S2, S3\}$, $b2 = \{S4\}$ and $b3 = \{S5\}$, and either buckets $b2$ or $b3$ can be selected as *smallRT* since they have the same cost, 3. If bucket $b3$ is selected then the TRMA would finish prematurely since it does not exist an improvement between $b1$ and $b3$ that reduces the existing unbalance of 3 with max cost 6. However, for buckets $b1$ and $b3$ we have $imp = S3$ which results in $b1 = \{S1, S2\}$ and $b2 = \{S3, S4\}$ with costs 5 and 5, thus showing a missed improvement opportunity.

This problem can be solved by selecting *smallRT* as the bucket with the lowest task cost and also the highest reuse with *bigRT*. This solution was implemented and, across all tests, had negligible impact on the reuse attained by the TRMA. Moreover, having to compare all *smallRT* candidates with *bigRT* has the execution time complexity $\mathcal{O}(n)$, since on the worst-case scenario we have $n/2 - 1$ buckets with one stage each (see Figure 3.12). Although the time complexity for TRMA would not be changed, we would be increasing the reuse analysis execution cost to not achieve any benefits.

The second kind of missed improvements is shown in Figure 3.14b, on which the selection of *smallRT* as one of the buckets with the smallest task cost (i.e., $b3$) results

in missing the balancement of $smallRT = b1$, both with $bigRT = b2$. By attempting to balance $b2$ and $b3$ there exists no valid improvement. However, with $b2$ and $b3$ we have $imp = S7$, which results in buckets $b2$ and $b3$ with new cost 7 for both, improving the previous maximum task cost of 8.

In order to solve this problem the reuse between a single $bigRT$ and all remaining buckets would need to be calculated, which is basically an exhaustive search for all valid balancements and would have a combinatory-like time complexity. Preliminary testing has shown that the last-bucket selection strategy already achieves reuse degrees of close to 95% of the reuse achieved by the RTMA for $MaxBucketSize = n$, for n stages. As such, neither of these extra-reuse problems are worth being solved.

Capítulo 4

Experimental Results

This chapter presents the experimental results of all proposed algorithms, regarding scalability, bucket cost balancement, the impact of different Sensitivity Analysis methods on reuse and the impact of the bucket size on run time.

4.1 Experimental Environment

We evaluated the proposed algorithms using a set of tissue images from brain cancer studies [21]. The images were divided into 4K×4K tiles for concurrent execution. The image analysis workflow consisted of normalization, segmentation and comparison stages. The comparison stage computes the difference between masks generated and a reference mask set, created using the application default parameters. The experimental evaluations were conducted on two distributed memory machine environments. The first is the TACC Stampede cluster, with each node having dual socket Intel Xeon E5-2680 processors, an Intel Xeon Phi SE10P co-processor and 32GB RAM. The nodes are inter-connected via Mellanox FDR Infiniband switches. Stampede uses a Lustre file system accessible from all nodes. The second environment is the PSC Bridges cluster. Each node has a dual socket Intel E5-2695 and 128 GB RAM. Bridges uses a Pylon file system accessible from all nodes. The application and middleware codes were compiled using Intel Compiler 13.1 with "-O3" flag in both cases.

4.2 Impact of Multi-level Computation Reuse for Multiple SA Methods

This section presents the impact of the computation reuse to the performance of the MOAT and VBD SA methods. We first compute MOAT on all the application parameters,

because it demands a smaller per parameter sampling to exclude those parameters that are non-influential to the output from the VBD. Most of the experiments in this section were executed using a small number of machines, because this section intended to detail the gains with the reuse optimizations. However, Sections 4.4 and 4.5 present experimental results for runs with large numbers of nodes.

4.2.1 Impact of Multi-level Computation Reuse for MOAT

Figure 4.1 presents the execution times of MOAT studies with parameter sample sizes varying from 160 to 640, which were executed using only 6 Stampede nodes to demonstrate the impact of the optimizations. The parameters were generated with a quasi-Monte Carlo sampling using a Halton sequence, which is known to provide a good coverage of the parameter space. These experiments use $MaxBucketSize$ set to 7, and the execution times refer to the makespan and also include the cost to perform the computation reuse analysis and I/O. For the task level merging approaches, the time spent by the merging algorithm is shown in the upper part of the graph bars. Additionally, five application versions were executed: the "No reuse" that employs the replica based composition, the "Stage level" performs reuse only of stage instances, and the "Task Level" that reuses fine-grain tasks and is executed with the Naïve, SCA, and RTMA algorithms. The TRMA was not included on this analysis since for this scale it has the same performance as RTMA.

The results presented in Figure 4.1 show that all application versions that reused computation significantly outperformed the baseline "No reuse" version. The "Stage Level" reached a speedup of up to 1.85× on top of the "No reuse", while the application versions with "Task Level" reuse have higher gains. The "Task Level - Naïve" is only

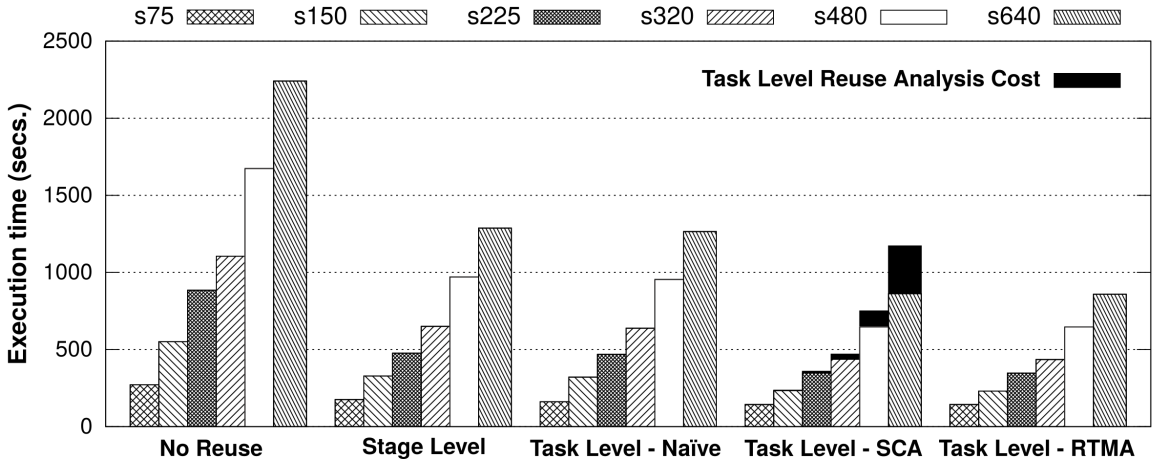


Figura 4.1: Impact of the computation reuse for different strategies as the sample size of the MOAT analysis is varied.

slightly better than the "Stage Level" (1.08× faster in the best case). This result is attributed to the highly order dependent nature of the naïve approach. The "Task Level" with SCA and RTMA, on the other hand, have remarkable speedups of up to, respectively, 1.39× and 1.5× on top of the "Stage Level" reuse only.

It is also noticeable from Figure 4.1 that the performance gains with RTMA increase as the sample size grows and, as a consequence, more reuse opportunities are available. In the SCA algorithm, however, the opposite behavior is observed. This is a result of the higher costs of executing SCA to compute the stages to be merged, which offsets the gains with the actual execution of the application after the merging. The time taken by Naïve, SCA, and RTMA to compute the reuse are shown on the top of their bars on Figure 4.1. For a sample of size 640, the time taken by SCA is about 26% of the entire execution. It is also interesting to see that although the RTMA takes a much shorter time to compute the merging choices, it provides solutions as good as the ones returned by the SCA. In the best case, RTMA attained a speedup of up to 2.61× on top of the "No reuse" version.

Regarding the atained reuse on the tested algorithms, both SCA and RTMA achieved values around 33% of reuse. This value is the raw value of tasks that were not executed due to a merging algorithm. As such, the speedup of 1.5× of RTMA on top of "Stage Level" reuse, which is greater than the 33% of reuse, is justified by the variable cost of each task. This means that the of 33% of tasks that were not executed, or reused, were comprised of expensive tasks. A further analysis on the costs of tasks and the impact this variance has on the implemented approaches is present on Section 4.5.1

4.2.2 Impact of Multi-level Computation Reuse for VBD

The performance of the proposed optimizations for the VBD are presented in Figure 4.2. The VBD was executed using the 8 remaining parameters (the original parameter set contains 15 parameters) that were not discarded in the MOAT analysis. VBD requirements are of the order of hundreds to thousands runs per parameter. As such, the sample size in this experiment is higher and was varied from 2000 to 10000 runs, whereas the same application versions used with MOAT were evaluated. In order to accelerate this analysis, we have increased the number of nodes to 16 Stampede nodes.

As presented in Figure 4.2, the relative performance of the application versions is similar to that observed with MOAT, except for the task level merging using SCA. Given that the sample size used in VBD is much higher, the SCA was not even able to finish computing the reuse to start up the actual execution of the workflow in 14000 secs. The RTMA had speedups of at most 2.9× against the "No Reuse" approach, and 1.51× on top of "Stage Level". These speedups were consistent with the ones found in the MOAT

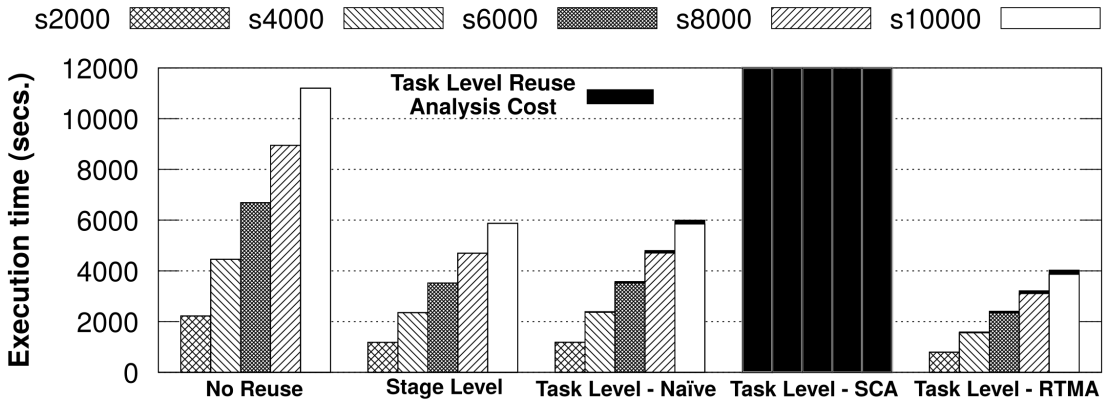


Figura 4.2: Impact of the computation reuse strategies for the VBD SA method.

analysis. Similarly, the reuse for the VBD experiments was of at most 35% for 10000 executions for the RTMA.

4.3 SA Methods Reuse Analysis

For all previous computation reuse tests which used the VBD method, the experiments were generated with the Latin Hypercube Sampler (LHS). Since the computation reuse on this work is highly reliant on the generated experiments, some sensitivity analysis methods were analyzed regarding their maximum reuse potential. Among them, in addition to LHS, the Monte-Carlo (MC) and Quasi-Monte-Carlo (QMC) methods were analyzed. The results are presented in Table 4.1. This analysis is only performed for VBD given its continuous ranges of parameter values, which would present itself with less potential reuse when compared to MOAT methods and their discrete parameter value ranges.

Sample Size	200	600	1000
MC	36.35%	36.46%	36.40%
LHS	36.62%	36.44%	36.44%
QMC	35.10%	34.44%	33.48%

Tabela 4.1: Maximum computation reuse potential for MC, LHS and QMC methods with different sample sizes. For VBD, the number of experiments is $10 \times SampleSize$. The reuse percentages represent fine-grain reuse after coarse-grain reuse, meaning that only fine-grain reuse is being shown.

4.4 Impact of Max Bucket Size

This section presents the impact of varying the $MaxBucketSize$ parameter on the execution times. As shown in Figure 4.3, an increase in $MaxBucketSize$ leads to smaller execution times because of the larger number of merging opportunities. This increase has, however, a threshold, after which the maximum reuse for the experiment is achieved (usually arround 33% of reuse, which results in speedups close to 1.5×).

However, it interesting to notice that the variation in execution times as a result of the bucket size changes, when comparing the two ends ($MaxBucketSize$ 2 and 8), is up to 12%, which shows that "Task Level" reuse can achieve significant gains even with small bucket sizes. This is result shows the viability of fine-grain reuse for execution environments on which there is a limited amount of memory available.

A large-scale SA experiment using with sample size of 240, 4,276 4K×4K image tiles, and 128 Stampede computing nodes, using all optimizations and the "No reuse", "Stage Level", and "Task Level RTMA" versions of the workflow attained execution times of, 15,681s, 12,544s and 6,173s, respectively.

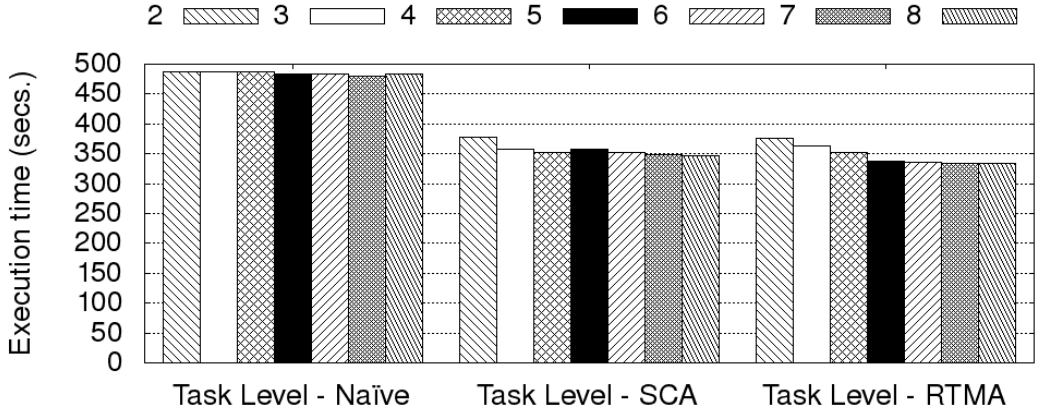


Figura 4.3: Impact of varying $MaxBucketSize$ from 2 to 8.

It is important to highlight that the task level merging reduces the number of stage instances up to $MaxBucketSize$ times, and the parallelism as a consequence. This could affect the application scalability if the number of stage instances after the merging was not sufficient to completely use the parallel environment.

4.5 The Effect of the Merging on Scalability

This section evaluates the case on which performing merging operations may lead to poor scalability due to loss of parallelism. This problem is caused by the load imbalance of executing a different number of buckets on each node and can be triggered by either

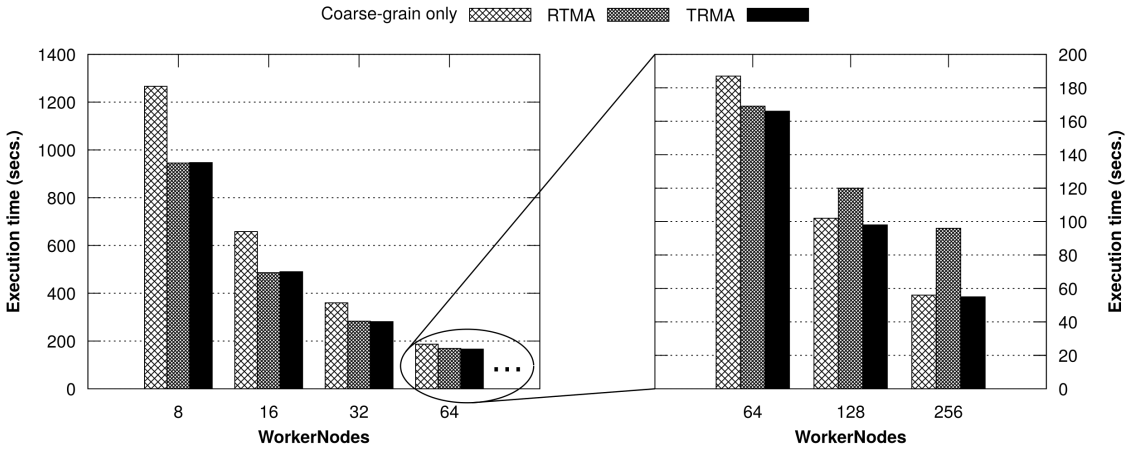


Figura 4.4: Comparison of the "no fine-grain reuse" (NR) approach with the RTMA and TRMA. RTMA uses $MaxBucketSize$ 10, while TRMA uses $MaxBuckets$ 3× the number Worker Processes (WP). The execution times for WP > 32 were zoomed in in a separated figure for the purpose of better visualization.

increasing the amount of merging performed or by increasing the number of nodes used. The later case was reproduced in Figure 4.4 with the MOAT SA method and a sample size of 1000, with up to 256 Worker Processes/nodes (WP).

This performance degradation caused by excessive merging, as seen with the parallel efficiency of the RTMA on Table 4.2, is aggravated by the variable cost of different buckets generated by the RTMA. The workflow used on this work had its stages broken into finer-grain tasks in order to mitigate this variance on the costs. Since the RTMA generate buckets that are balanced stage-wise, but not task-wise, this difference in the number of tasks per bucket may lead to unbalancement on environments with a low stages-per-worker ratio. This unbalancement leads to a reduction of parallelism and, thereafter, degradation on the performance of the application due to load imbalance among nodes. On these cases the Task-Balanced Reuse-Tree Merging Algorithm (TRMA) could be employed to extenuate this problem.

Still on Table 4.2 it is visible that if the stages-per-worker ratio becomes low enough, the RTMA parallel efficiency drops to an extent on which it performs worse than not performing any fine-grain reuse at all. The values of stages-per-worker ratio (S/W), parallel efficiency and TRMA reuse, compiled on Table 4.2, show that regardless the reuse algorithm employed, for the highest WP values the S/W ratio becomes low enough to impact the parallelism. This is true not only to RTMA, which becomes worse than "No Reuse" (NR) after WP 64, but also for NR itself. This loss of parallelism in NR is an indication of the unbalancement between stages without reuse caused by the variance

on the cost of tasks of the same level, but different inputs. Given that, the NR parallel efficiency values can be seen as the upper bound for any approach, since the reuse degree cannot increase for bigger WP values, nor can the parallel efficiency.

The TRMA approach manages to improve on the RTMA parallel efficiency through bucket balancement, resulting in it not becoming worse than NR (see Table 4.2 and Figure 4.4). The speedups that TRMA achieves on top of NR lowers as WP increases, becoming negligeble for WP values of at least 128 (see Table 4.3). Given that for WP 256 the TRMA attained 10.73% of reuse, the speedup should either match this value o come close to it. This phenomenon of lack of performance is cause by another source of unbalancement on buckets.

4.5.1 The Impact of Variable Task Cost

By taking another look at Table 4.2 we can notice that the loss of parallelism due to unbalancement starts at WP 32 for the RTMA and TRMA approaches. This indicates that there exists another source of unbalancement, for merging algorithms only, that affects RTMA harder than TRMA and that is unaffected by TRMA balancement techniques. It was found that this imbalance comes from the difference in the cost of tasks of different levels.

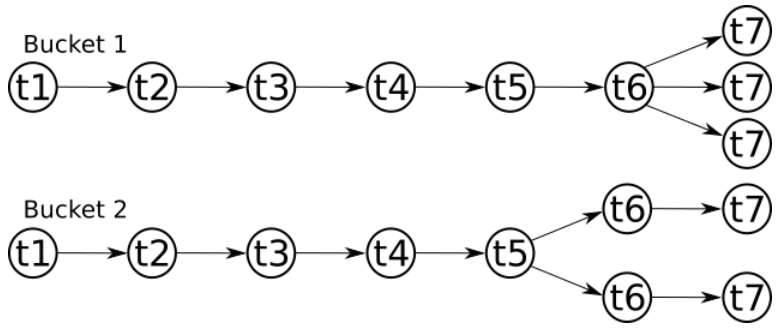
As shown in Table 4.4, the costs of the task which compose a stage are not constant. As such, buckets which are balanced task-count-wise may still be susceptible to imbalance. An example of such case is presented in Figure 4.5. There, we have two buckets with the same number of tasks, but with different topologies. The first bucket was generated with three stages that attained maximum reuse, while the second had only two stages with less reuse. By using the TRMA, the difference of execution cost between them of

Worker Processes (WP)	8	16	32	64	128	256
No Reuse S/W	126	63	31.5	15.75	7.85	3.93
RTMA S/W	47.25	23.62	11.81	5.9	2.95	1.47
No Reuse efficiency	-	96.21%	91.26%	93.03%	91.43%	90.95
RTMA efficiency	-	97.20%	85.72%	83.64%	70.08%	62.44
TRMA efficiency	-	96.68%	86.94%	84.65%	84.38%	89.04

Tabela 4.2: Combination of Stages per Worker Processes (S/W) and parallelism efficiency values. The S/W ratio for TRMA was fixed as 3 for all WP values. The parallelism efficiency was calculated based on the previous execution (e.g. for WP 64, it is the execution time for WP 32 vs WP 64).

Worker Processes (WP)	8	16	32	64	128	256
Speedup TRMA vs NR	1.33	1.34	1.27	1.12	1.04	1.01
TRMA reuse	32.96%	32.96%	32.11%	30.58%	28.23%	10.73%

Tabela 4.3: Speedup of the TRMA vs the "No Reuse" (NR) approach.



(a) Two example buckets and their reuse trees. Bucket 1 was the result of the merger of three stages, while Bucket 2 had two stages initially.

Task	t1	t2	t3	t4	t5	t6	t7	Total
Bucket 1	0.12	0.20	0.06	0.03	0.08	0.39	0.27	1.18
Bucket 2	0.12	0.20	0.06	0.03	0.08	0.79	0.18	1.48

(b) Sum of relative costs of tasks for each bucket. For a bucket containing only a single stage, and thus 7 tasks, the total cost would be 1.

Figura 4.5: An example case on which two buckets with the same number of tasks have different execution costs. This is due to the difference in the cost of different tasks. In this example Bucket 1 should execute 1.25× faster than Bucket 2.

around 25% would go unnoticed. This unbalancement is enough to impact the parallel efficiency of an application through load imbalance. Effectively, this problem just makes the unbalancement of buckets by tasks visible on an earlier S/W ratio.

Altogether, three sources of unbalancement affects the maximum achievable parallel efficiency: (i) differently sized buckets (same stage count but different task count), (ii) buckets with the same size (task count) but different topologies, and (iii) same tasks

Task	t1	t2	t3	t4	t5	t6	t7	Total
Avg Exec Time (s)	1.14	1.99	0.65	0.33	0.76	3.76	0.86	9.51
Percentual	12.03%	20.90%	6.92%	3.49%	8.02%	39.59%	9.05%	100%

Tabela 4.4: An empirical evaluation on the costs of each task of which a stage is composed of. This approximation was generated with the purpose of showing the relative cost of the tasks, not being suitable as a absolute cost approximation.

having variant execution costs, which happens if two stages with the same topology and task count can have significantly different costs. The (i) problem is already solved by the TRMA, while (ii) and (iii) can only be solved if we have an approximation of the costs of each task *a priori*.

Capítulo 5

Conclusion

This work has proposed new algorithms that optimize Sensitivity Analysis (SA) through multi-level computation reuse. These algorithms were employed to optimize SA on a medical imaging analysis workflow, executed on large scale computation environments. Three fine-grain computation reuse algorithms were implemented, along with optimizations in order to deal with balancement, level of parallelism available and memory constraints.

The application selected for evaluating the proposed optimizations was a microscopy image analysis workflow. This workflow was chosen given its relevance **??**, having a large sample space (around 21 trillion parameter combinations). The workflow is comprised of three stages, with the most expensive operation (segmentation) being composed of seven finer-grain tasks. On this workflow distinct SA methods were applied (MOAT and VBD) with several experiment generation methods (Section 4.3). Also, these analysis were tested on a large scale environment, running the Region Templates Framework (RTF) with at most 256 worker processes.

The RTF received two main improvements. The first is a way to easily generate workflows compatible with the RTF. This was achieved by using a descriptor file for the definition of each stage of the workflow, with a GUI to build and compose workflows based on this descriptor. These workflow compositions are performed with the assist of the Taverna Workbench [40], which provides an easy way to generate workflows for application experts.

Although computation reuse was an already studied strategy to reduce computational cost (Section 2.4), it was different from what was proposed by this work. The referenced approaches would either need a training step to be executed before the main application, which would be rather inefficient for a large scale workload such as the one used on this work; or perform computation reuse through caching methods, which would be too expensive to be employed on large scale computation environments. As such, the algorithms proposed on this work fill these limitations by performing computation reuse,

in a lightweight manner.

Computation reuse was implemented and evaluated in two levels, stage-level and task-level. Stage-level computation reuse, implemented with a coarse-grain merging algorithm, was already proposed on previous works [35, 36] and re-implemented in this work. Although it already reduced the overall runtime by a large factor, some other computation reuse opportunities were unachievable through coarse-grain merging. Therefore, task-level computation reuse, implemented with fine-grain merging algorithms, was employed. One important feature of the fine-grain merging algorithms was that they could be used on top of coarse-grain merging results, augmenting their performance.

Out of the three fine-grain merging algorithms proposed, implemented and evaluated the Reuse-Tree Merging Algorithm (RTMA, Section 3.3.3) stood out as an efficient approach. The RTMA achieved both high reuse factor (around 35%) and low execution cost, when compared with the remaining approaches.

It was identified that task balancement could be a problem if the ratio of tasks per core was low. In order to solve this problem a new approach based on the RTMA was implemented. This new approach, the Task-Balanced Reuse Tree Algorithm (TRMA), was implemented to behave as the RTMA if the raw number of tasks is large enough that maximum parallelism is achieved, while also not degrading its performance if the tasks-per-core ratio was low. Moreover, the TRMA was implemented with the intent to take only into consideration parallelism issues, by adjusting the $MaxBuckets$ parameter, which can be automatically chosen on runtime to optimize the application makespan while also taking the memory restrictions into consideration, thus reducing the dependency on the end user.

All algorithms were tested at first with the MOAT and VBD SA methods in order to assert their performance on real-world applications. It was shown that even though coarse-grain merging already had great speedups (from 1.85× to 1.9×), fine-grain reuse managed to improve this values, achieving aggregate speedups between 1.39X to 1.51X on top of coarse-grain merging results, amounting to speedups of up to 2.89X. However, it is worth noting that the Smart Cut Algorithm (SCA) execution cost did not scale well, making this approach unfeasible for large scale setups.

The impact of the $MaxBucketSize$ constraint on the performance of the application was also analyzed, proving that the RTMA can be employed on heavily memory-constrained environments while also achieving good speedups. Since the TRMA algorithm was equivalent to the RTMA on regular, large scale setups, only the worst-case scenario was tested. It was shown that even on this case, the TRMA would always follow the best-case behavior. Finally, in order to validate the existence of computation reuse opportunities in the use case applications, and therefore validate the use of the proposed

algorithms as a way to improve the makespan said applications, different SA experiment generators were tested in order to verify their maximum reuse degree. It was shown that across all cases the reuse degree was high enough to justify the use of computation reuse algorithms.

As a future work other aplication workflows would be to studied. For those applications the extensibility and ease of generating a new workflow from scratch would be observed. Then, it would be interesting to see the impact on reuse of differently structured workflows.

Another way to further optimize the workflow execution time through computation reuse is to perform balancement of buckets not by task count, but using the actual tasks costs. This approach would yield the best result, since there is not be any other source of loss of parallelism through unbalancement of buckets. However, cost analyzing is a difficult task which requires intrumentation and monitoring of such tasks [37, 11, 19], returning an estimation of these tasks costs. As such, the performance of this task-cost balancement can only be as good as the estimative of the tasks costs.

Furthermore, by balancing the buckets by task cost the bucket sizes could be limited only by parallelism and memory restrictions. The parallelism limitation is trivial to implement, being the number of buckets at least the number of worker processes. Again, the estimation is where the difficulty lies, being this memory consumption value rather hard to be found through static analysis [4]. However, a task-cost balanced, memory-limited algorithm would attain not only maximum reuse, but maximum theorical speedup since all limitations of computation would be solved.

Referências

[1] *Sensitivity analysis in practice: a guide to assessing scientific models*, ISBN 978-0-470-87093-8. 6, 7

[2] Alvarez, Carlos, Jesus Corbal e Mateo Valero: *Fuzzy memoization for floating-point multimedia applications*. 54(7):922–927. 12, 17

[3] Barreiros, Willian, George Teodoro, Tahsin Kurc, Jun Kong, Alba C. M. A. Melo e Joel Saltz: *Parallel and Efficient Sensitivity Analysis of Microscopy Image Segmentation Workflows in Hybrid Systems*. páginas 25–35. IEEE, setembro 2017, ISBN 978-1-5386-2326-8. `http://ieeexplore.ieee.org/document/8048914/`, acesso em 2017-11-10. x, 5

[4] Barthe, G., M. Pavlova e G. Schneider: *Precise analysis of memory consumption using program logics*. Software Engineering and Formal Methods, setembro 2005. 55

[5] Bradski, G.: *The opencv library*. Dr. Dobb's Journal of Software Tools, 2000. 10

[6] Campolongo, F., J. Cariboni e A. Saltelli: *An effective screening design for sensitivity analysis of large models*. Environmental Modelling & Software, 22(10):1509–1518, 2007. 1

[7] Connors, Daniel A. e Wen Mei W. Hwu: *Compiler-directed dynamic computation reuse: rationale and initial results*. Em *Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, páginas 158–169. IEEE Computer Society. 11, 12, 13, 17

[8] Cooper, L., D. Gutman, Q. Long, B. Johnson, S. Cholleti, T. Kurc, J. Saltz, D. Brat e C. Moreno: *The proneural molecular signature is enriched in oligodendrogliomas and predicts improved survival among diffuse gliomas*. PloS ONE, 5, 2010. 5

[9] Cooper, L. A., J. Kong, D. A. Gutman, F. Wang, J. Gao, C. Appin, S. Cholleti, T. Pan, A. Sharma, L. Scarpace, T. Mikkelsen, T. Kurc, C. S. Moreno, D. J. Brat e J. H. Saltz: *Integrated morphologic analysis for the identification and characterization of disease subtypes*. J Am Med Inform Assoc., 19(2):317–323, 2012. 5

[10] Filippi-Chiela, E. C., M. M. Oliveira, B. Jurkovski, S. M. Callegari-Jacques, V. D. da Silva e G. Lenz: *Nuclear morphometric analysis (nma): Screening of senescence, apoptosis and nuclear irregularities*. PloS ONE, 7, 2012. 5

[11] Geimer, M., S. Shende, A. D. Malony e F. Wolf: *A generic and configurable source-code instrumentation component*. G. Allen, J. Nabrzyski, E. Seidel, G. D. van Albada, J. Dongarra, P. M. A. Sloot (Eds.), ICCS, 5545(2):696–705, 2009. 55

[12] Gurcan, M. N., H. Shimada T. Pan e J. Saltz: *Image analysis for neuroblastoma classification: segmentation of cell nuclei*. Conf Proc IEEE Eng Med Biol Soc, páginas 4844—-4847, 2006. 5

[13] Guttmann-Beck, N. e R. Hassin: *Approximation algorithms for minimum k-cut*. Algoritmica, 27:198–207, 2000. 25

[14] Han, J., H. Chang, G. V. Fontenay, P. T. Spellman, A. Borowsky e B. Parvin: *Molecular bases of morphometric composition in glioblastoma multiforme*. 9th IEEE International Symposium on Biomedical Imaging, páginas 1631–1634, 2012. 5

[15] Iooss, B. e P. Lemaitre: *A review on global sensitivity analysis methods*. In G. Dellino and C. Meloni, editors, Uncertainty Management in Simulation-Optimization of Complex Systems, 59:101–122, 2015. 8

[16] Iooss, B. e P. Lemaitre: *A review on global sensitivity analysis methods in Uncertainty Management in Simulation-Optimization of Complex Systems*, volume 59. Springer US, 2015. 1

[17] J. Goecks, A. Nekrutenko, J. Taylor: *Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences*. Genome Biol., 11(8), 2010. 2, 3, 12, 16

[18] Jr., Eugene Santos e Eunice E. Santos: *Effective computational reuse for energy evaluations in protein folding*. International Journal on Artificial Intelligence Tools, 15(5):725–739, 2006. 2, 3, 12, 16

[19] Knüpfer, Andreas, Dieter Kranzlmüller, Bernd Mohr e Wolfgang Nagel: *M09 - program analysis tools for massively parallel applications: how to achieve highest performance.* página 223, janeiro 2006. 55

[20] Kong, J., L. A. D. Cooper, F. Wang, J. Gao, G. Teodoro, T. Mikkelsen, M. J. Schniederjan, C. S. Moreno, J. H. Saltz e D. J. Brat: *Machine-based morphologic analysis of glioblastoma using whole-slide pathology images uncovers clinically relevant molecular correlates*. PLoS ONE, 2013. 1

[21] Kong, Jun, Lee A. D. Cooper, Fusheng Wang, Jingjing Gao, George Teodoro, Tom Mikkelsen, Matthew Schniederjan J., Carlos S. Moreno, Joel H. Saltz e Daniel J. Brat: *Machine-based morphologic analysis of glioblastoma using whole-slide pathology images uncovers clinically relevant molecular correlates*. PLoS ONE, 2013. 6, 44

[22] Körbes, A., G. B. Vitor, R. de Alencar Lotufo e J. V. Ferreira: *Advances on watershed processing on gpu architecture*. In Proceedings of the 10th International Conference on Mathematical Morphology, 2011. 5

[23] Lepak, Kevin M. e Mikko H. Lipasti: *On the value locality of store instructions*, volume 28. ACM. 11, 12, 17

[24] Meredith, J. S., S. Ahern, D. Pugmire e R. Sisneros: *Eavl: The extreme-scale analysis and visualization library.* páginas 21–30, 2012. 5

[25] Modarressi, Mehdi, Seyyed Hossein Nikounia e Amir Hossein Jahangir: *Low-power arithmetic unit for DSP applications.* Em *System on Chip (SoC), 2011 International Symposium on*, páginas 68–71. IEEE. 12, 17

[26] Mood, Benjamin, Debayan Gupta, Kevin Butler e Joan Feigenbaum: *Reuse it or lose it: More efficient secure computation through reuse of encrypted values.* Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, páginas Pages 582–596, 2014. 2, 3, 11, 12, 16

[27] Morris, M. D.: *Factorial sampling plans for preliminary computational experiments.* Technometrics, 33(2):161–174, 1991. 1, 7

[28] Nakra, T., R. Gupta e M. Soffa: *Value prediction in vliw machines.* Int. Symp. Computer Architecture, páginas 258–269, 1999. 2, 11, 12

[29] Patlolla, D. R., E. A. Bright, J. E. Weaver e A. M. Cheriyadat: *Accelerating satellite image based large-scale settlement detection with gpu.* In Proceedings of the 1st ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data, páginas 43–51, 2012. 5

[30] Richardson, S. E.: *Caching function results: Faster arithmetic by avoiding unnecessary computation.* Technical Report, Sun Microsystems Laboratories, 92(1), 1992. 2, 11, 12, 15

[31] Saltelli, A. (editor): *Global sensitivity analysis: the primer.* John Wiley, Chichester, England ; Hoboken, NJ, 2008, ISBN 978-0-470-05997-5. OCLC: ocn180852094. 1, 6

[32] Sodani, A. e G. S.Sohi: *Dynamic instruction reuse.* Proc. Int. Symp. Computer Architecture, páginas 194–205, 1998. 2, 11, 12, 13, 14, 15

[33] Steen, J. Van der, J.L. Coenders, S. Pasterkamp, A. Rolvink e J. Van Steekelenburg: *Computational reuse optimisation for stadium design.* Proceedings of the International Association for Shell and Spatial Structures, 2015. 2, 3, 11

[34] Stoer, M. e F. Wagner: *A simple min-cut algorithm.* J. ACM, 44(4):585—-591, 1997. 25, 27

[35] Teodoro, G., T. Pan, T. Kurc, J. Kong, L. Cooper, S. Klasky e J. Saltz: *Region templates: Data representation and management for high-throughput image analysis.* Parallel Computing, 40(10):589–610, 2014. x, 2, 7, 8, 9, 10, 54

[36] Teodoro, George, Tahsin M. Kurc, Luís F. R. Taveira, Alba C. M. A. Melo, Yi Gao, Jun Kong e Joel H. Saltz: *Algorithm sensitivity analysis and parameter tuning for tissue image segmentation pipelines.* Bioinformatics, 2017. x, 7, 8, 20, 54

[37] Truong, H. L., P. Brunner e T. Fahringer V. Nae: *Dipas: A distributed performance analysis service for grid service-based workflows.* Future Generation Comp. Syst, 25(4):385–398, 2009. 55

[38] Wang, Weidong, A. Raghunathan e N.K. Jha: *Profiling driven computation reuse: An embedded software synthesis technique for energy and performance optimization*. Proceedings. 17th International Conference on VLSI Design, 2004. 2, 3, 11, 12, 15, 16

[39] Weirs, V. G., J. R. Kamm, L. P. Swiler, S. Tarantola, M. Ratto, B. M. Adams, W. J. Rider e M. S. Eldred: *Sensitivity analysis techniques applied to a system of hyperbolic conservation laws*. Reliability Engineering & System Safety, 107:157–170, 2012. 1, 8

[40] Wolstencroft, Katherine, Robert Haines, Donal Fellows, Alan Williams, David Withers, Stuart Owen, Stian Soiland-Reyes, Ian Dunlop, Aleksandra Nenadic, Paul Fisher, Jiten Bhagat, Khalid Belhajjame, Finn Bacall, Alex Hardisty, Abraham Nieva de la Hidalga, Maria P. Balcazar Vargas, Shoaib Sufi e Carole Goble: *The taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud*. Nucleic Acids Res, 2013. 22, 53

[41] Xu, Gang, Tsz Ho Kwok e Charlie C.L. Wang: *Isogeometric computation reuse method for complex objects with topology-consistent volumetric parameterization*. 91:1–13, ISSN 00104485. `http://linkinghub.elsevier.com/retrieve/pii/S0010448517300477`, acesso em 2017-12-12. 12, 17

[42] Yasoubi, Ali, Reza Hojabr e Mehdi Modarressi: *Power-efficient accelerator design for neural networks using computation reuse*. 16(1):72–75, ISSN 1556-6056. `http://ieeexplore.ieee.org/document/7393481/`, acesso em 2017-12-12. 12, 16