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 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 [23]. This process is executed in many phases of scientific research and can be used to lower the effective computational cost of analisys on such researches, or even improve the quality of the results through parameter optimizaiton.

The main motivation of this work is the use of image analysis workflows for whole slide tissue images analysis [14], 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 [19], [30], [3], [11], 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) [19] may require hundreds of runs (sample size) of the image analysis workflow. 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 [27]. 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 some parameters with the same values. Since the workflows used on this work are hierarchical, and, as such, can be broken down in routines, it would be wasteful if one of these routines were to be executed on two or more evaluaitons, generated by the SA method, with the same parameters values and other inputs.

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, thus being unnecessary the re-execution of such task. On hierarchical workflows, the routines of which they are composed by can be reused fully, or partially, by breaking these routines in smaller subroutines. 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 fine-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 coarse-grain tasks reuse, and that the new fine-grain merging algorithms are proposed in this work in order to improve the performance of SA applications alongside the existing coarse-grain merging algorithm.

Other works perform computation reuse as a means to reduce overall computational cost in different ways [20, 24, 22, 29, 18, 25, 12, 13]. 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 [24, 22], which are not generic or flexible enough for the given problem. Some apply reuse by profiling the applicaiton [29], 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 applicaitons [18, 25, 12, 13], 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 comparison would be at least unfair.

This work proposes two ways of accomplishing computation reuse in SA, (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 the 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 computation reuse.

The specific contributions of this work are presented below with a reference to the section in which we describe them:

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 solves unbalancement issues on 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.

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. 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.

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 [9, 21, 16, 6, 7, 4, 5, 17] 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.

The microscopy image analysis workflow used on this work is presented in Figure 2.1. This workflow consists of normalization, segmentation, feature computation and final

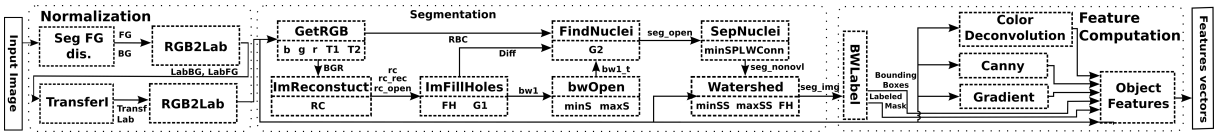
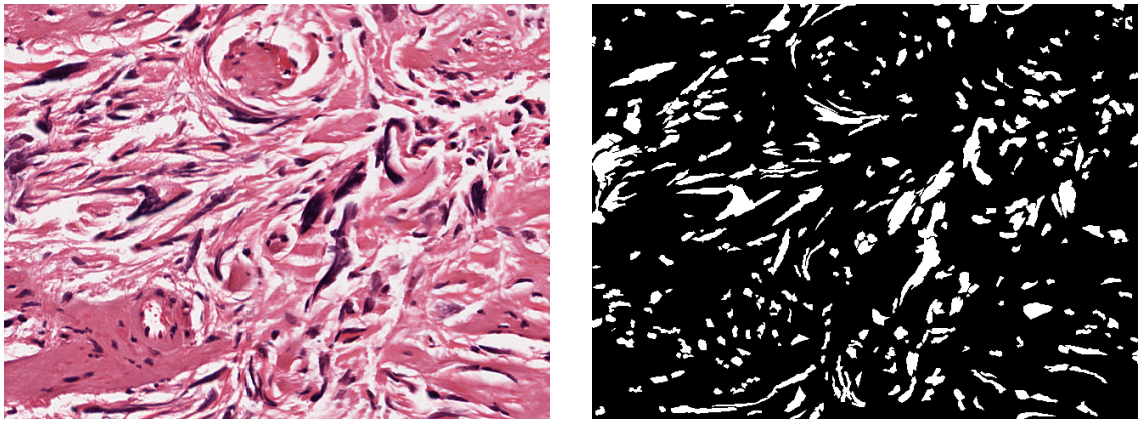


Figura 2.1: An example microscopy image analysis workflow performed before image classification. Image extracted from [1].



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

Figura 2.2: An example of tissue image segmentation.

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 of 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 parameters values, 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 will be detailed on the next 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 [23]. This process is applied on several phases of scientific research including, but not limited to model validation, parameter studies and optimization, and error estimation.

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 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) [27, 28], 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.

The light SA method, Morris One-At-A-Time (MOAT) [19], 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 value ranges are uniformly partitioned in p levels, thus resulting in a p^k grid of parameter sets to be evaluated. Each evaluation output x_i of the application creates a parameter elementary effect (EE), calculated as $EE_i = \frac{y(x_1,...,x_i+\Delta_i,...,x_k)-y(x)}{\Delta_i}$, with $y(x)$ being the application output before the parameter perturbation. In order to account for global SA the RTF uses $\Delta_i = \frac{p}{2(p-1)}$ [28]. The MOAT method requires $r(k+1)$ evaluations, with r in the range of 5 to 15 [10].

The second SA method, Variance-Based Decomposition (VBD) is preferably performed after a lighter SA method, as the MOAT method, since it requires $n(k+2)$ evaluations for k parameters and n samples, and n can lie in the order of thousands of executions [30]. Thus, a reduced number of parameters is used for feasibility reasons. VBD, unlike MOAT, discriminates the the output uncertainty effects among individual parameters (first-order) and high-order effects.

The combination 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 [27]. It supports hierarchical workflows that are composed of coarse-grain stages, which in turn are composed by fine-grain tasks, with all the dependencies being 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.

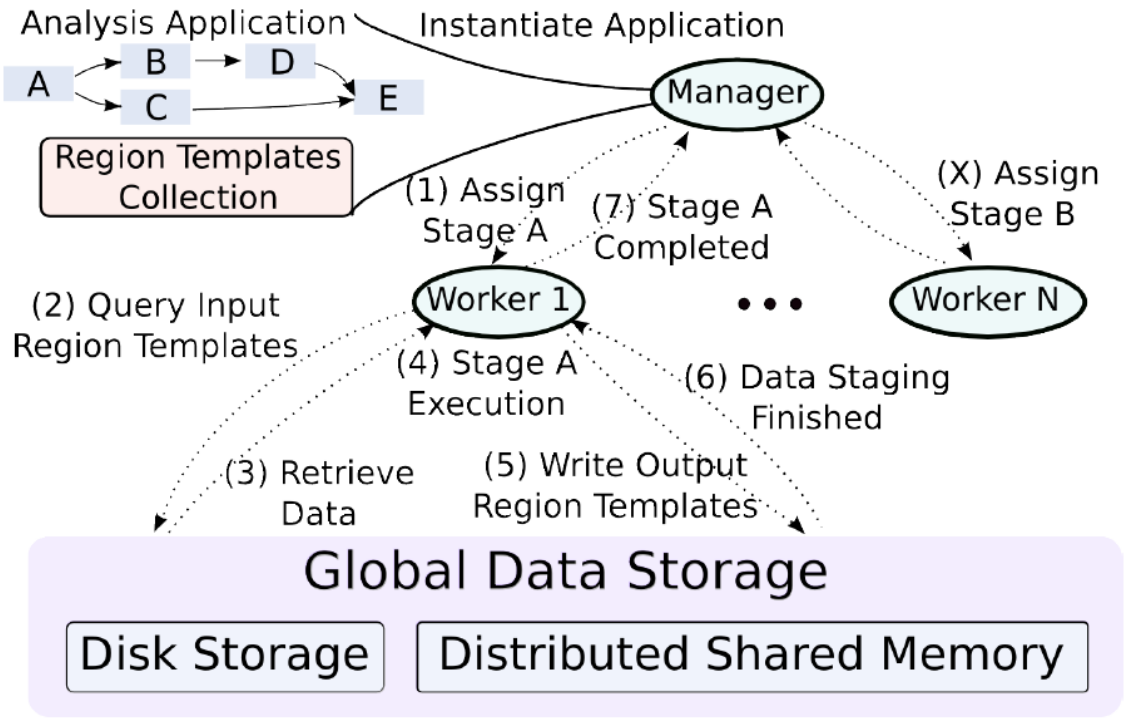


Figura 2.3: The main components of the Region Templates Framework, highlighting the steps of a coarse-grain stage instance execution. Image extracted from [27].

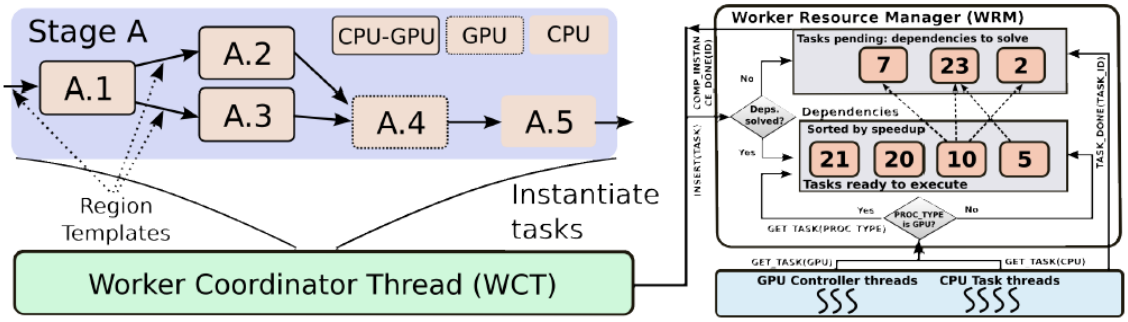


Figura 2.4: The execution of a stage instance from the perspective of a node, showing the fine-grain tasks scheduling. Image extracted from [27].

The main components of the RTF are: the data abstraction, the runtime system, and the hierarchical data storage layer [27]. The runtime system consists of core functions for scheduling of application stages, transparent data movement and management via the storage layers, as shown in Figure 2.3. 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 (stage-level and task-level). Fine-grain scheduling is possible at the second level to 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 undelying infrastructure. Being a stage composed of tasks, these are scheduled locally by the Worker executing them. These tasks differ in terms of data access paterns and computation intensity, thus, ataining 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 [27]. 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, arrays (e.g., images or 3D volumes), segmented and annotated objects and regions, all of which are implemented using the OpenCV [2] library interfaces to simplify their use. A region template 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 region template instance. A region template 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 region template 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 has been employed on both the hardware and software fronts with diverse techniques, such as value prediction [20], dynamic instruction reuse [24] and *memoization* [22], with the goal of accelerating applications through the removal of duplicated computational tasks. This concept has been used for runtime optimizations on embedded systems [29], low-level encryption value generation [18] and even stadium designing [25]. All of those approaches can be placed into one of two categories, hardware-level or software-level reuse. This work focus on the second type since hardware-level reuse requires specific hardware designing, which is not the target of this work.

On software-level reuse two main techniques were noticed, (i) reuse based on pre-execution profiling and (ii) *memoization* of recurrent tasks. The first method [29], although efficient, is rather application dependent and incurs in a pre-execution training step, which could be unfeasible to implement for SA applications given the size of the parameters set domain.

The other common approach is through caching of results of selected executions [22, 12, 13] named *memoization*. This method can virtually be used for any application domain, however, increases in the number of reuse opportunities available are achieved through either more domain-specific selection methods or by increasing the amount of memorized results. Also, in order to reduce the overhead of searching for reusable tasks results, higher-level operations ought to be used, effectively reducing the potential of reuse. Taking into account the granularity of the fine-grain task of the workflow used on this work and the scale of the number of tasks, the storage of this many results would be impractical. Finally, while the use of *memoization* can easily be brought to multiprocessed environments through the addition of parallel control structures (e.g., locks), when we attempt to bring this approach to a distributed environment there is an additional overhead for coherence enforcement, which can be hard to scale.

On [24] Sodani and Sohi implement a hardware-level *reuse buffer* for some instructions, which if an operation under execution is found, its execution can then be finished and the buffered result be returned. By doing this, Sodani and Sohi claim to have cut down the resources required to execute some instructions while also reducing the length of critical path of some operations. A similar but distinct hardware-level buffer is proposed on [22] for arithmetic operations.

On software level Wang and Raghunathan propose on [29] the use of computation reuse in order to make embedded applications more energy efficient. This is done by profiling the target application, grouping operations into reuse regions, generating a software-level cache and then doing this process over with new execution data in a feedback loop. This

approach draws some similarities with [18], in the sense that low-level software operations are reused through the caching of some results.

Still on software level, but on a higher level of abstraction, [12, 13] proposes the application of computation reuse on high-level complex operations or functions. On [12], complete analysis applications for genetic research can be reused if all input parameters match. This is done with the goal of reproducibility. Finally, [13] analyses reusing fitness calculation results for genetic algorithms applied to protein folding applications. Both approaches use a software-level cache to store reusable results.

When analyzing the above approaches for computation reuse and trying to apply their concepts to our domain of SA, any hardware-level proposal cannot be used. Their use would unnecessarily increase the complexity of the application while making any type of parallelism hard to achieve. Regarding [29] software-level approach, even a simple segmentation analysis application is too expensive for profiling. Moreover, while the flexibility of runtime operation grouping is desirable, the image segmentation analysis workflow used in this work is simple enough to be efficiently analyzed manually. Finally, regardless the level on which computation reuse is applied, caching approaches add an operations layer which needs to be optimized to the specific hardware environment on which it is being executed (e.g., the number of cache levels, cache size and number of blocks, replacement policy). This layer, while being easily implemented for serialized applications can become rather difficult very quickly for distributed applications, on which there needs to be a sense of global coherence among computing nodes. Given these problems, simpler and lighter approaches are required for the proposed computation reuse on SA applications on multiprocessed distributed environments.

Given that any approach for computation reuse is also defined by the level of abstraction of the analyzed tasks, it is worth noting that higher-level task reuse (or coarse-grain task reuse) is considerably easier to implement since the granularity of coarse-grain tasks incur in more expensive tasks, and thus resulting in less inner-task communication overhead. Also, working on the coarse-grain granularity level prevents the seizing of partial reuse opportunities.

Finer-grain task reuse can hence explore the missed opportunity of partial reusable coarse-grain tasks, since only the common portion can be reused. This approach is however much more difficult to cope with since the communication overhead between merged tasks is more significant given that fine-grain tasks are computationally less expensive.

Also, for the RTF environment performing stage merging indiscriminately can have a toll on parallelism. This is most prominently seen when analyzing a set of workflows which all of its stages instances have some minor degree of reuse opportunities among themselves. By attempting to maximize the reuse of such stages instances we would at

the same time minimize the overall computational cost of the analysis and serialize the execution of the stages to a single node.

With these characteristics for fine-grain reuse none of the previous analyzed works can be efficiently employed to solve this problem. With that, this work proposes a static reuse analysis, done before the execution of any task. With this tactic there is no need for any sort of caching systems. Moreover, the results of most of the tasks are images, what makes the storage of such data impracticable since the cost of storaging these images is generally greater than the cost of executing a fine-grain task. At last, the a static reuse analysis tactic is compatible with distributed environments since it requires no additional communication between worker nodes.

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

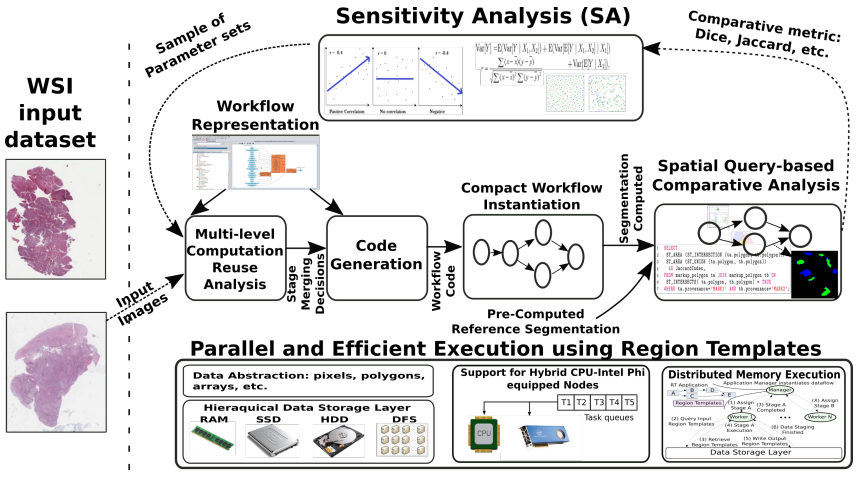


Figura 3.1: The parameter study framework. A SA method selects parameters of the analysis workflow, which is executed on a parallel machine. The workflow results are compared to a set of reference results to compute differences in the output. This process is repeated a set number of times (sample size)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

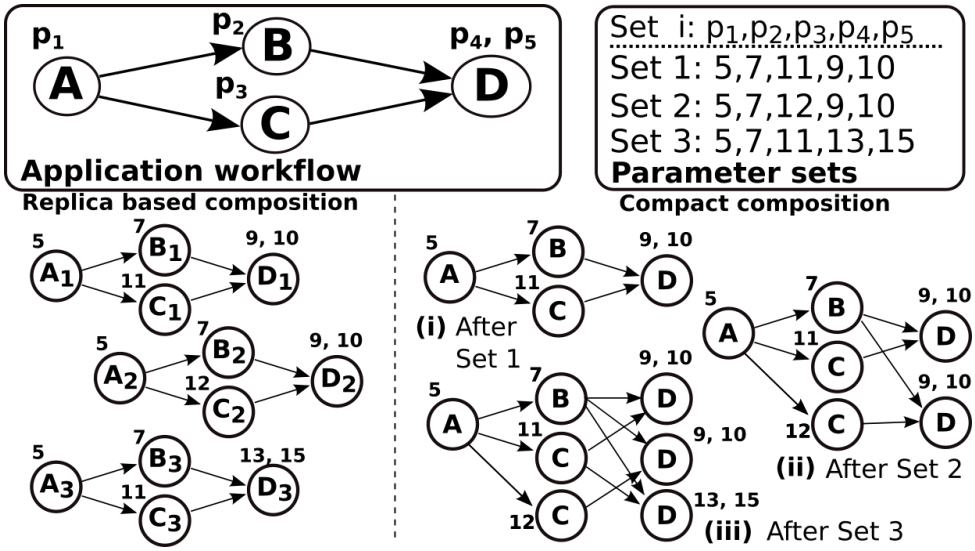


Figura 3.2: A comparison of a workflow generated with and without computation reuse. Image extracted from [28].

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.

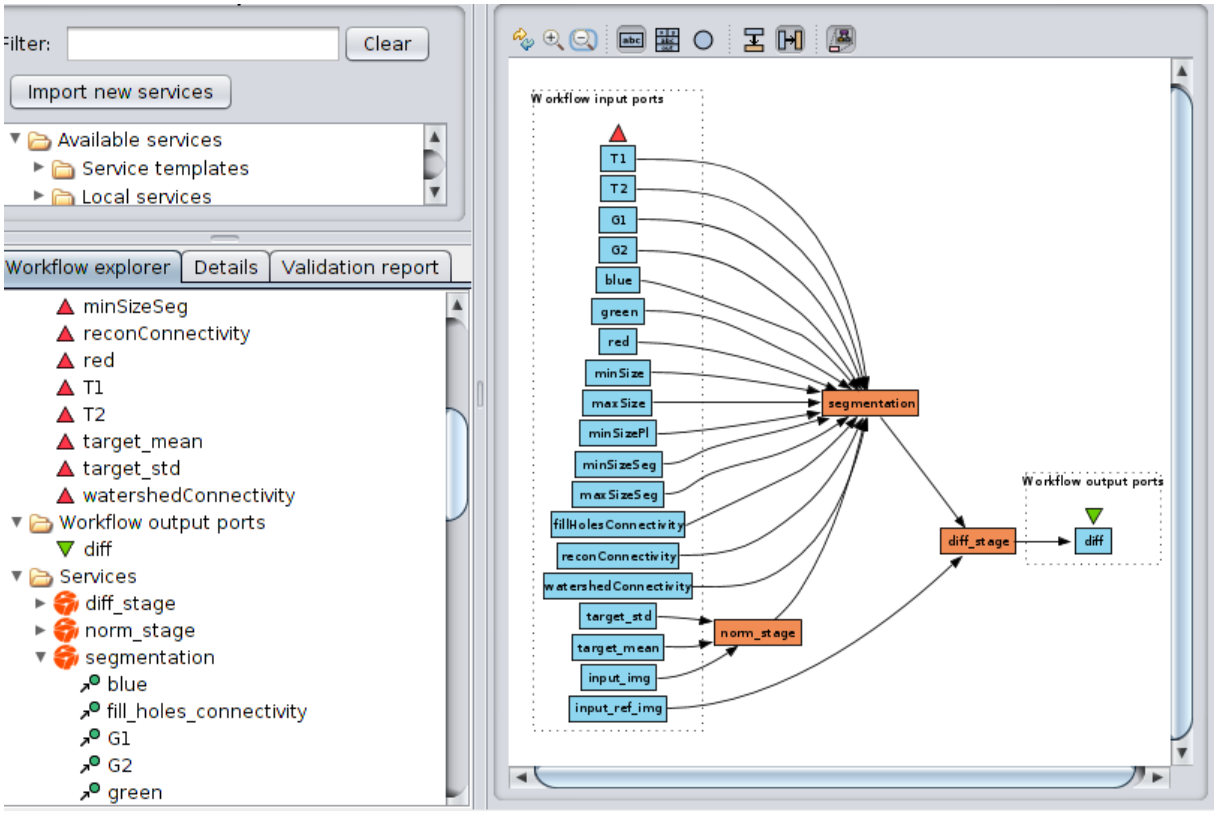


Figura 3.4: The example workflow described with the Taverna Workbench.

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 [31] 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 thorugh 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

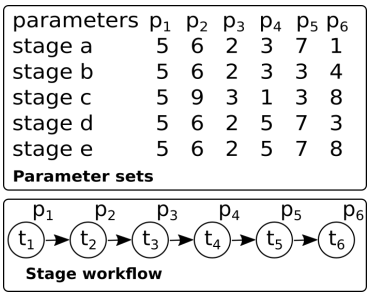
In the interest of better understanding the task-level merging problem, a naïve algorithm was implemented to serve as a baseline for our analysis. This simplified algorithm groups $MaxBucketSize$ stages in buckets and attempt to merge all stages of each bucket among themselves. This was achieved by sequentially grouping the first $MaxBucketSize$ stages into buckets, untill 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.

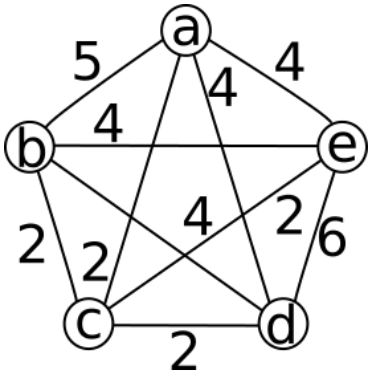
3.3.2 Smart Cut Algorithm (SCA)

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 [26].

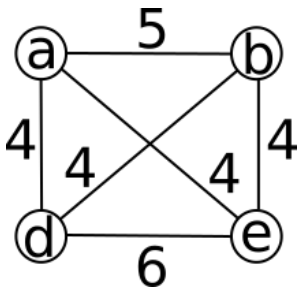
Although there are many variations for the min-cut problem [26, 8], we define here a min-cut algorithm as one that takes an undirected graph and performs a 2-cut (i.e., cut the graph in two subgraphs) operation, minimizing the sum of the cut edges weight. This



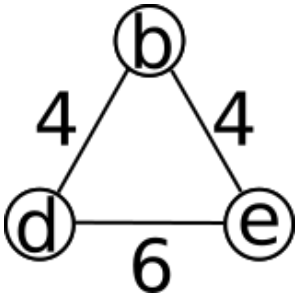
(a) Example application.



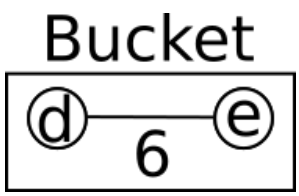
(b) Initial graph of instance example.



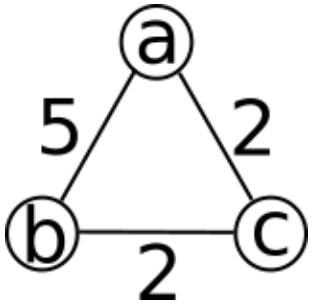
(c) First cut is performed, removing node c.



(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 $MaxBucketSize = 2$.

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. Next, nodes a and b are removed until a bucket of size 2 is

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
```

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 [26] (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 [26] 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)

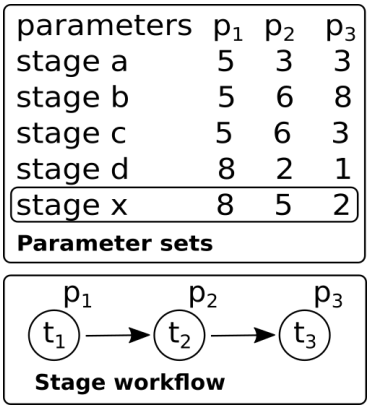
Still on graphs, a natural way to display hierarchical structures is with trees. Using tasks as nodes on this tree, subtrees with the same parent node indicates that all child task nodes of said parent node use its output. As such, if we constructed a tree with several stages, we are able to easily see the reuse opportuninies, 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 arese same for both stages, an thus reusable among themselves. 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.

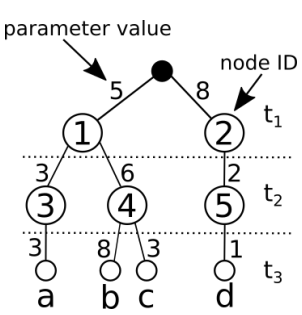
On a SA example we have a workflow w that is instantiated n times with different parameters (w_1, w_2, ..., w_n). Each workflow w_i is composed by m stages s_{ij} with $i \in [1, n]$ and $j \in [1, m]$. A reuse-tree is then generated for each j-th stage level. The reuse-tree for a given stage level can 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 intantiation 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 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).

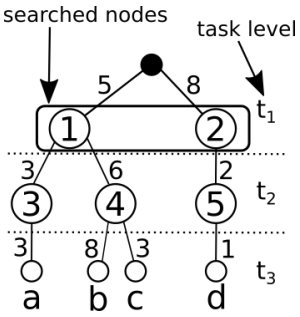
In order for a merging algorithm to be implemented on top of the Reuse Tree structure we must take advantage of its hierarchical characteristics. Given that we want to bundle together buckets of stages of at most $MaxBucketSize$ stages we can start with the deepest



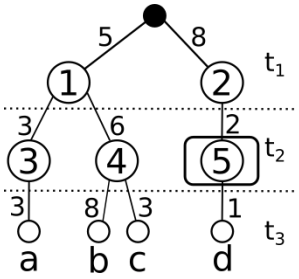
(a) Example application.



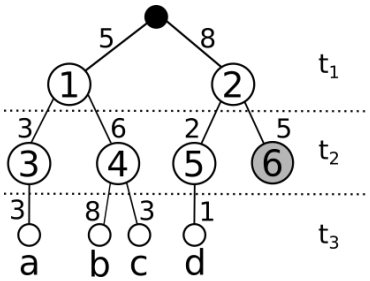
(b) Initial reuse tree for the instance example.



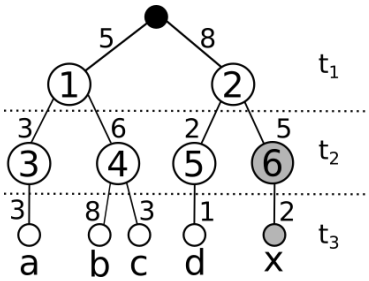
(c) Searching for reuse on the first task.



(d) Searching for reuse on the second task.



(e) Inserting a new node, 6.



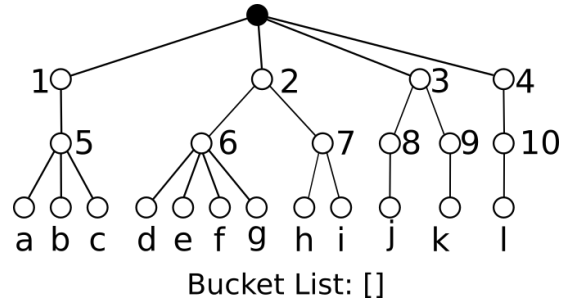
(f) Inserting the leaf node x.

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.

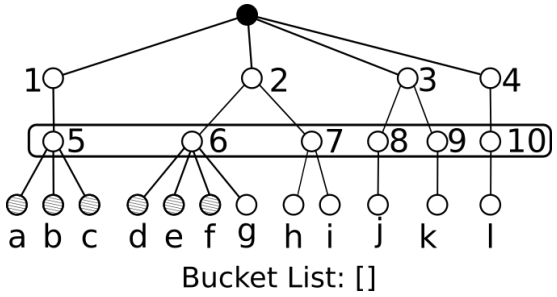
stages and move up. Figure 3.7a shows an example of a Reuse Tree with 12 stages and 3 tasks each. Stages a, b and c have two out of three reusable tasks, and as such, given a $MaxBucketSize = 3$, should be put together in the same bucket. Meanwhile, stages d through i have one out of three reusable tasks. However, it is better to place three of the stages d, e, f and g together, with the remaining stage on a bucket with h and i. As we can see, 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).

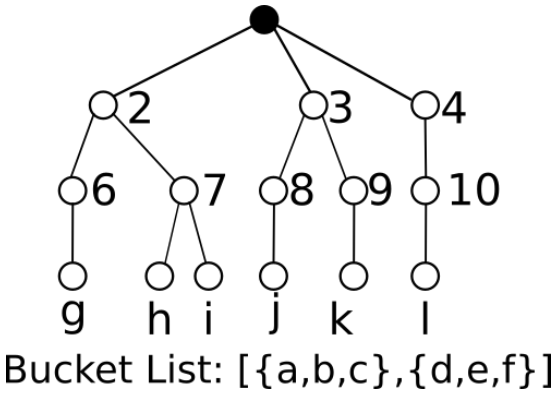
The first step of the algorithm (Algorithm 3, line 6) is to get a list of all parents of leaf nodes. In Figure 3.7b we can see the selected parents (5-10). With the leaf's parents list the pruning step makes as many $MaxBucketSize$ sized buckets as possible and then remove them from the reuse tree. The procedure $PruneLeafLevel$ (line 7) attempts to



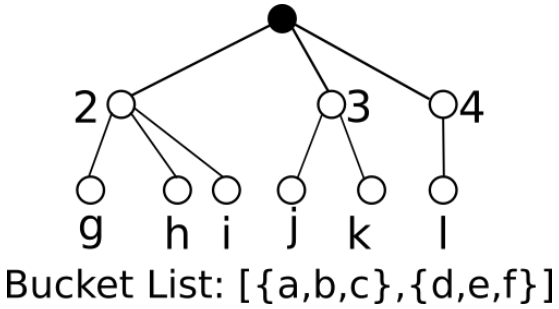
(a) Initial reuse tree.



(b) Reuse Tree after select procedure.



(c) Reuse tree after pruning procedure.



(d) 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.

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 3.7b 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 3.7c.

If a parent node ends up grouping all of its children in buckets, it must be removed from the tree (node 5 on Figure 3.7b). 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 3.7b is also removed, as seen on Figure 3.7c).

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 3.7c are moved to parent node

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
```

2, as seen on Figure 3.7d). After that, the current node is remove from its parent (e.g., nodes 6 and 7 of Figure 3.7c are removed from parent node 2, as seen on Figure 3.7d). After all nodes from the parent list are removed and its children are moved up the tree height is updated (line 6).

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 a number $m < n$ of 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 analysis of the actual merging algorithm will be split by 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's $\mathcal{O}(n^2)$, 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.

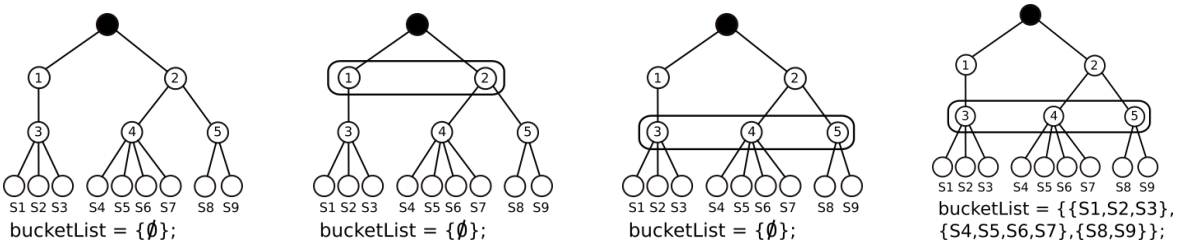
3.3.4 Task-Balanced Reuse Tree Merging Algorithm (TRMA)

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 the unbalanced grouping of stages. To solve this problem, a balancement-wise version of RTMA was implemented, the Task-Balanced Reuse Tree Merging Algorithm (TRMA).

The TRMA can be explained as an improved version of the RTMA, on which the balancement of the buckets' cost is taken into consideration. This is achieved by breaking all stages into an user-defined number of buckets, which are balanced afterwards. This balancements is task-wise, meaning that the difference of task number of any two buckets at the end of the TRMA is at most k, being k the number of tasks that a stage is composed. Although we cannot ensure that the computational cost of all types of tasks are equivalent, being this another source of unbalancement, the granularity of unbalancement is smaller, resulting in a better result, when comparing with the previous approaches covered by this work.

The TRMA is performed in three steps, using the same reuse-tree structure as the one used by the RTMA. At first, all stages are divided into $MaxBuckets$ disjoint buckets that maximizes the overall reuse, in an operation defined as *Full-Merge*. This means that there cannot exist another combination of disjoint buckets which has an overall cost (total number of non-repeated tasks) smaller that the set of buckets outputted by the *Full Merge* operation. Figure 3.8 shows an example of the *Full-Merge* process on which the stages are perfectly divided into the three buckets. As seen in Figure 3.8d, after the stopping condition of *Full-Merge* the stages are arranged in buckets, while preserving the reuse-tree for further use (i.e., balancing).

Still, there may be a case on which the *Full-Merge* will not be able to exactly divide the stages in $MaxBuckets$. On this particular case, as shown in Figure 3.9, the *Full-Merge* operation will stop then the number of buckets is greater than $MaxBuckets$, as seen



(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.

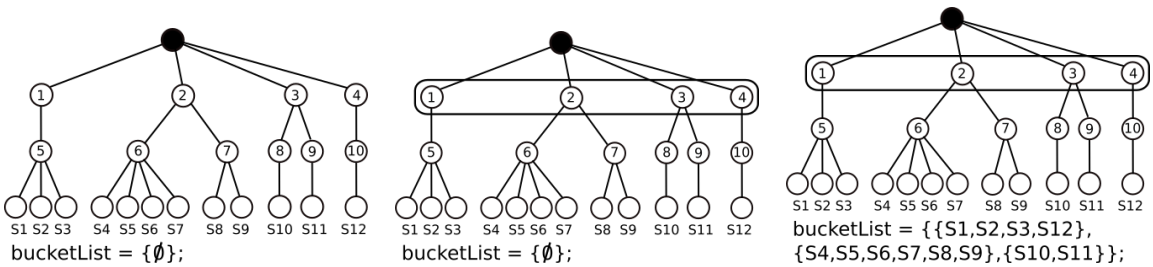
(d) Merger is performed since the correct number of buckets was achieved.

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

in Figure 3.9b. From this number of buckets greater that $MaxBuckets$ another merger process is applied, the *Fold-Merge*.

It is simple to notice that for a set of ordered numbers (sized n, being n even) which we want to find set of $n/2$ disjoint pairs of distinct numbers that minimizes the maximum sum of each pair, we can simply pair the opposite ends of the initial set. E.g., for a set {1,2,3,4,5,6} we can have the result set {{1,6},{2,5},{3,4}}, in a procedure that resembles a visual "folding" of the initial set, pivoted on the center of the list (between 3 and 4).

As such, we want to transform a set of n buckets in another set of size $MaxBuckets$ ($n > MaxBuckets$), which can be achieved by performing a "folding" operation on $k = 2(n - MaxBuckets)$ buckets. The k "folded" buckets are selected in order to minimize the maximum cost of the $MaxBuckets$ output buckets, being the cost defined by the number of different tasks of a buckets. Thus, the k buckets with the smallest cost are chosen to be "folded". This process of folding k buckets in order to achieve $MaxBuckets$ buckets is defined as *Fold-Merge*, with an example of its application shown in Figure 3.9c

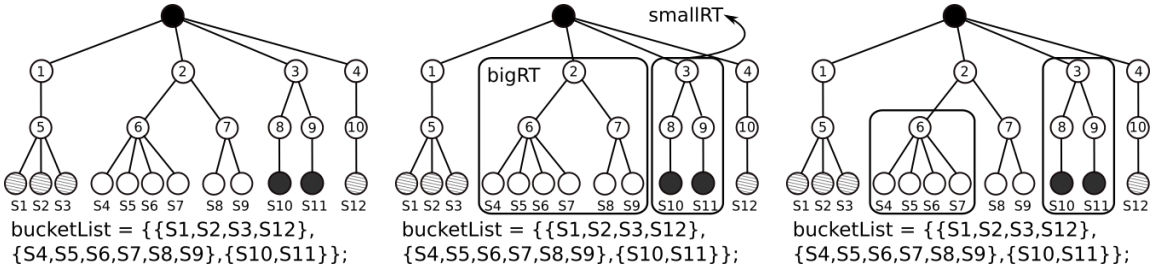


(a) Initial reuse-tree.

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

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

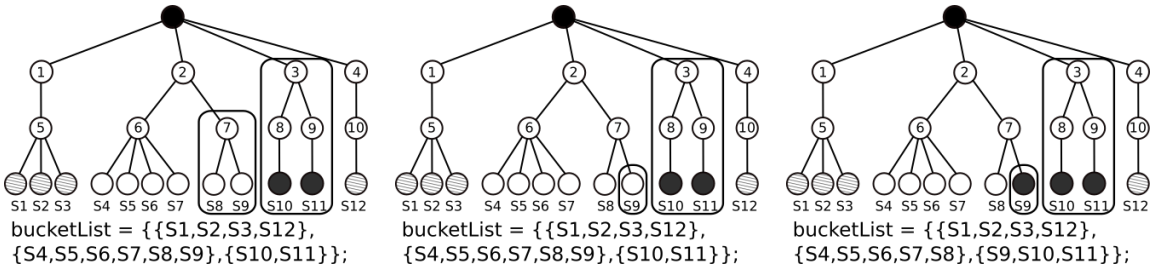
Figura 3.9: Another example of *Full-Merge* and *Fold-Merge* on which $MaxBuckets$ is 3 and the exact division of stages cannot be reached by *Full-Merge*.



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

With the correct number of buckets, these are then balanced among each other, on a *Balance* step, defined by Algorithm 4. An example of this operation is seen on Figure 3.10. At first, we have a reuse-tree which had its stages divided into three buckets (Figure 3.10a) as a result of the *Full-Merge* and *Fold-Merge* operations. As before, the cost of a bucket is defined by the number of different tasks it has. As stated on Algorithm 4, we attempt to balance the buckets represented by the reuse-tree nodes with greatest ($bigRTN$) and smallest ($smallRTN$) costs of all input buckets ($bucketList$). It is worth noting that although the input data structure of *Balance* is a list of buckets, each one of its stages are also leaf nodes of the reuse-tree generated by the previous steps of the TRMA.

An attempt of balancement is defined as a reuse-tree node $improvement \in bigRTN$, which leaf nodes can be subtracted from subtree $bigRTN$ and added to subtree $smallRTN$, resulting in a smaller unbalancement value, i.e., being $unbal = TaskCost(bigRTN) - TaskCost(smallRTN)$ the old unbalancement and $nUnbal = Max(TaskCost(bigRTN \setminus improvement) - TaskCost(smallRTN \cup improvement))$ the new unbalancement value,

Algorithm 4 The *Balance* step of the TRMA

```
 1: Input: bucketList;
 2: Output: bucketList;
 3: while true do
 4:     sort bucketList by descending cost
 5:     bigRTN ← bucketList.first()
 6:     smallRTN ← bucketList.last()
 7:     unbal ← TaskCost(bigRTN) - TaskCost(smallRTN)
 8:     improvement ← SingleBalance(bigRTN, smallRTN, unbal)
 9:     newMksp ← Max(TaskCost(bigRTN \ improvement), TaskCost(smallRTN ∪ improvement))
10:     if improvement ≠ ∅ and newMksp < TaskCost(bigRTN) then
11:         smallRTN ← smallRTN ∪ improvement
12:         bigRTN ← bigRTN \ improvement
13:     else
14:         break
15:     end if
16: end while
17: return bucketList
```

$unbal < nUnbal$. Also, in order for a balancement improvement be valid the new maximum cost value, after the improvement, must be smaller than $bigRTN$ cost. This constraint prevents useless balancement attempts, i.e., the unbalancement value is improved but there is no difference in the application makespan. An example of useless improvement can be seen in Figure 3.10d, on which the improvement of sending the children of node 7 to smallRTN, represented by node 3, results in a better unbalancement (3 vs the previous 4) while not improving the final outcome of the balancement procedure (the maximum cost of the buckets under balancement is still 9). We call this an unnatural improvement, on which a better balancement value is achieved by simply increasing the cost of $smallRTN$, thus, not having any positive contribution to the application makespan. This check is performed at lines 9 and 10 of Algorithm 4.

The process of finding balancement attempts is demonstrated on Algorithm 5. It starts with two input subtrees, $bigRTN$ and $smallRTN$, being the first, the one with the greatest cost. As shown in lines 4 and 10 of Algorithm 5, the search is performed in a depth first search fashion, on which every visited node (c) is checked as an improvement (lines 5-9). If there is not any improvements on $bigRTN$, $\emptyset$ is returned (lines 3 and 16).

This search process is exemplified on Figure 3.10. Given the selected $bigRTN$ and $smallRTN$ on Figure 3.10b, the nodes of $bigRTN$ are searched for improvements. On Figure 3.10c, on which node 6 is visited, the attempt of sending stages S4-S7 to $smallRTN$ results in $bigRTN$ with cost 4 and $smallRTN$ with a new cost of 11. Given that this is not an acceptable balancement attempt the current best improvement value continues to be $\emptyset$. After that, node 7 is searched (Figure 3.10d). On this case we have a better unbalancement, validating it as an improvement (although, being an unnatural improvement), then becoming the new best solution (Algorithm 5, lines 7 and 8). Finally, one of the leaf

Algorithm 5 Balancement algorithm for two tree nodes (*SingleBalance*)

```
1: Input: bigRTN; smallRTN; unbal;
2: Output: improvement;
3: improvement ← ∅
4: for each unique children c ∈ bigRTN do
5:     newUnbal ← | TaskCost(bigRTN \ c) - TaskCost(smallRTN ∪ c) |
6:     if newUnbal < unbal then
7:         unbal ← newUnbal
8:         improvement ← c
9:     end if
10:    bestRec ← SingleBalance(c, smallRTN, unbal)
11:    if bestRec ≠ ∅ then
12:        unbal ← | TaskCost(bigRTN \ bestRec) - TaskCost(smallRTN ∪ bestRec) |
13:        improvement ← bestRec
14:    end if
15: end for
16: return improvement
```

nodes is searched, resulting in a task cost of 8 for both $bigRTN$ and $smallRTN$, and as such, an unbalancement of 0 (Figure 3.10e). After the search for balancement attempts between $bigRTN$ and $smallRTN$ is finished, the current best solution (leaf node S9) is tested by Algorithm 4 (lines 10-12) and accepted. Since there cannot be an improvement on the current unbalancement of 0, the next attempt of *SingleBalance* (line 8) will return an $improvement = \emptyset$, thus ending the *Balance* step.

Given the already known complexity for the reuse-tree generation, of $\mathcal{O}(n^2)$, the *Full-Merge*, *Fold-Merge* and *Balance* operations remain to be analyzed. On the worst case *Full-Merge* will visit every node of the reuse-tree. Since there cannot be more than kn nodes on a reuse-tree, being k the number of tasks of each one of the n stages, the complexity for *Full-Merge* is bounded by $\mathcal{O}(kn)$. Moreover, the necessary calculations for *Fold-Merge* can be performed in constant time, and that we have b buckets to look into, the computational cost for this operation is negligible. Given that these two operations are less expensive than the reuse-tree generation procedure, their cost are ignored.

In order to calculate the complexity for the remaining operation, *Balance*, we must first create a general model for a worst-case reuse-tree, which is seen in Figure 3.11. On this model, we have a reuse-tree divided into b buckets, of which $b-1$ have exactly one stage. This represents a worst-case scenario on which one big bucket is shrinked one stage at a time, until a perfect balance is achieved. E.g., given 3 buckets $B1$, $B2$, and $B3$, with respectively 1, 1 and 4 stages, and with all the stages of $B3$ having the first task reusable among all 4 stages, the costs for the 3 buckets will be 2, 1, 3 after the first balance iteration, and 2, 2, 2 after the second iteration.

Furthermore, *Balance* is divided into two main parts, the main loop of Algorithm 4, and the balancement attempts of Algorithm 5. The main loop of *Balance* runs while there

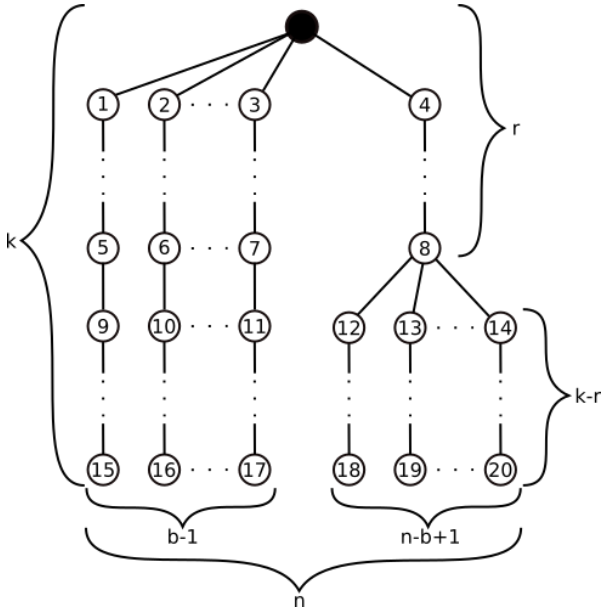


Figura 3.11: A general worst-case reuse-tree representation on which we have all n stages divided into b buckets. On this case we have $b-1$ buckets with exactly one stage, and thus cost k. Hence, the last bucket has $n-b+1$ stages. For this last bucket we assume the single and uniform reuse of the first r task, having no reuse for the remaining $k-r$ tasks. This is the worst-case for balancement applications.

are improvements, which value can be estimated using the general model of Figure 3.11. Since the last bucket (beginning on node 4) has $n-b+1$ stages and it will reach perfect balance once its size reaches n/b we have that $n-b+1-n/b$ improvement operations will be performed. Besides, for each improvement attempt there will be a call to the algorithm used for sorting the buckets by their cost (Algorithm 4, line 4) and a call for SingleBalance. Seeing that on the worst case SingleBalance will visit each node of the biggest bucket (i.e., the last bucket of Figure 3.11, represented by node 4), totalizing $r+(n-b+1)(k-r)$ nodes, and also that, generally, good sorting algorithms are $\mathcal{O}(n\ log(n))$, we arrive at the final complexity of $\mathcal{O}(kn^2+nb\ log(b))$. Similarly to the RTMA, $n \gg k$, ergo, the time complexity can be dominated by n^2. In the same manner, if the number of buckets is large enough it approaches $n^2\ log(n)$. This case is, however, unlikely, since it means that for n stages we want to separate them in n to n/b ($b \simeq n$ and $n > b$) buckets, resulting in a speedup of at most n/b, which does not justify the use of any fine-grain merging algorithm. This speedup is calculated based on the best case scenario, on which there is maximum reuse on every bucket, e.g., a bucket with 5 stages and k tasks per stage has $k+4$ tasks.

As a final observation, it is easy to notice that any leaf node on the interval of S4-S9 of Figure 3.10e 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

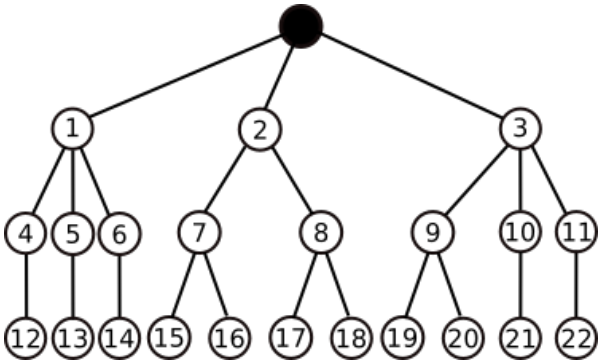


Figura 3.12: An example reuse-tree that can be used to illustrate possible prunable nodes. E.g., the use of nodes 4, 5, 6, or 10, 11 as an improvement attempt results in the same outcome (cost 3), making them interchangeable, as with nodes 7, 8 or 9 (cost 4), or nodes 12-22 (cost 3).

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 be seen using Figure 3.12.

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 3.12 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 3.12 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 3.12 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.

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 [15]. 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 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 slightly better than the "Stage Level" (1.08× faster in the best case). This result is

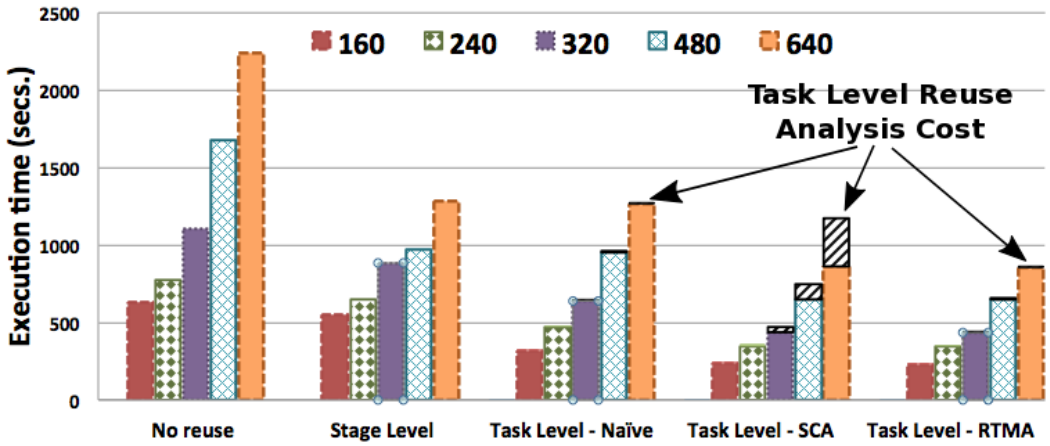


Figura 4.1: Impact of the computation reuse for different strategies as the sample size of the MOAT analysis is varied.

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.

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

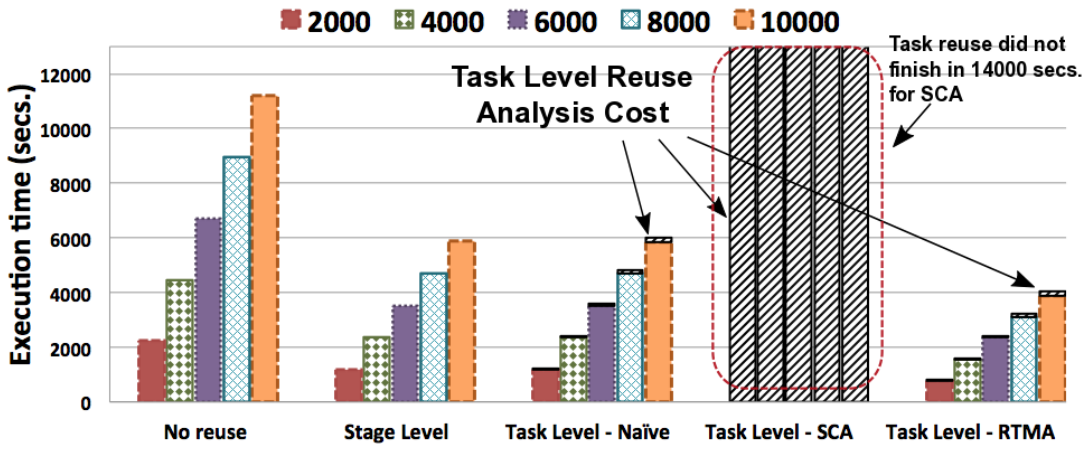


Figura 4.2: Impact of the computation reuse strategies for the VBD SA method.

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".

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 with 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. However, it interesting to notice that the variation in execution times as a result of the bucket size changes is up to 12%, which shows that "Task Level" reuse can achieve significant gains even with small bucket sizes. 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.

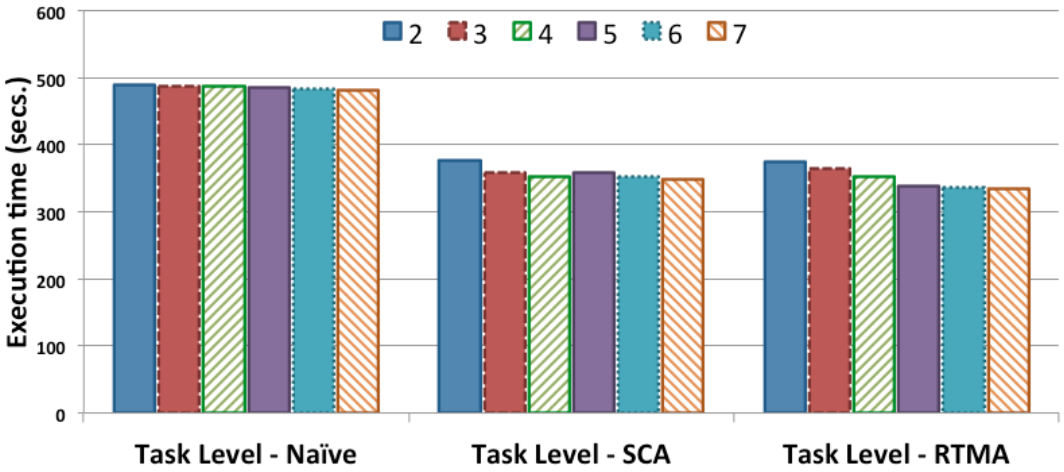


Figura 4.3: Impact of varying $MaxBucketSize$.

We want 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 opperations may lead to poor scalability due to unbalancement. This problem is rather common with workloads of variable cost. 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 inbalance among nodes. Although this problem may be mitigated by lowering the value of $MaxBucketSize$, by doing such we also reduce the amount of computation reuse. On these cases the Task-Balanced Reuse-Tree Merging Algorithm (TRMA) could be employed to extenuate this problem. One of this cases was found by using the MOAT SA method with a sample size of 1000 and up to 256 Worker processes/nodes.

Figure 4.4 shows that when the ratio of stages per worker is low enough (manipulated here by increasing the number of workers) the RTMA stops to scale. This problem is so severe that when comparing the RTMA (with $MaxBucketSize$ 10) and "No Reuse" approaches, RTMA needs 74% more time to execute 36% less tasks on 256 Worker Processes.

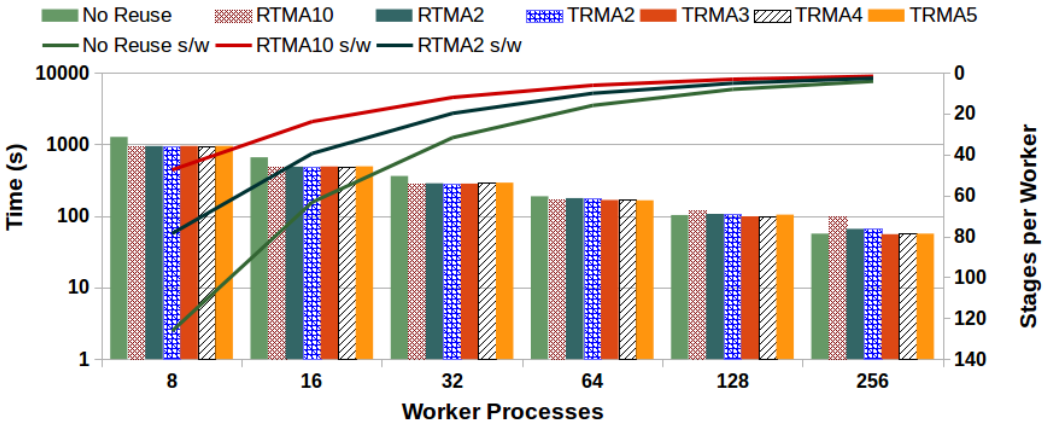


Figura 4.4: Comparison of the Coarse-grain-only approach with the RTMA and TRMA. RTMA uses $MaxBucketSize$ 10 and 2, while TRMA uses $MaxBuckets$ 2, 3, 4 and 5 $\times$ the number Worker Processes. The stages-per-worker ratio is also shown for the No Reuse and RTMA approaches only, since the ratios for TRMA are fixed (2, 3, 4 and 5, respectively).

The RTMA (with $MaxBucketSize$ 10) performance drops from 128 Worker Processes on, when the stages-per-worker ratio is at least 3. By setting the stages-per-worker ratio to 3 we can see that the TRMA outperforms all other approaches and configurations. This result is only achievable for two reasons, (i) the reuse achieved by using the TRMA with $MaxBuckets$ $3 \times WorkerProcesses$ is high enough to reduce the overall execution cost (see Table 4.2), (ii) while maintaining the buckets balanced task-wise.

MaxBuckets	768	512	256	128	64	32	16	8
Reuse	10.73%	26.06%	30.61%	31.01%	32.95%	32.98%	33.89%	33.91%

Tabela 4.2: Reuse of the TRMA with different $MaxBuckets$ values.

Figure 4.4 also shows how carefull we must be when chosing a stages-per-worker ratio value for TRMA. By going too high the amount of reuse will be reduced. However, if the value is too low the amount of achievable reuse increases while unbalancement becomes a problem still. This unbalancement, while not being as bad as the one present in RTMA, is a consequence of the variable cost of tasks and cannot be easily solved, since it would require the computational cost of the tasks *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 [31], 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 [27, 28] 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.

Referências

[1] 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

[2] Bradski, G.: *The opencv library*. Dr. Dobb's Journal of Software Tools, 2000. 9

[3] 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

[4] 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

[5] 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

[6] 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

[7] 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

[8] Guttmann-Beck, N. e R. Hassin: *Approximation algorithms for minimum k-cut*. Algoritmica, 27:198–207, 2000. 20

[9] 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

[10] 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. 7

[11] 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

[12] 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, 11, 12

[13] 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, 11, 12

[14] 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

[15] 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. 35

[16] 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

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

[18] 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, 11, 12

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

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

[21] 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

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

[23] 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

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

[25] 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, 11

[26] Stoer, M. e F. Wagner: *A simple min-cut algorithm.* J. ACM, 44(4):585—-591, 1997. 20, 22

[27] 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, 42

[28] 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, 15, 42

[29] 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, 11, 12

[30] 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, 7

[31] 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. 17, 41