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

Otimização de reúso computacional através de um escalonador sensível ao gasto de memória

Eduardo Scartezini C. Carvalho

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Orientador
Prof. Dr. George Luiz Medeiros Teodoro

Brasília
2019

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

Otimização de reúso computacional através de um escalonador sensível ao gasto de memória

Eduardo Scartezini C. Carvalho

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

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

Prof. Dr. Alba Cristina M. A. de Melo
CIC/UnB

Prof. Dr. Marcos F. Caetano
CIC/UnB

Prof. Dr. José Edil Guimarães de Medeiros
Coordenador do Curso de Engenharia da Computação

Brasília, 7 de Março de 2019

Dedicatória

Àquelas que me visitaram diversas vezes durante a produção deste projeto...

Agradecimentos

Ao meu orientador, professor George Luiz Medeiros Teodoro, pelos conhecimentos passados, pela paciência e por ter me apoiado em muitos momentos. Sou grato pela chance de ter participado deste projeto.

Aos meus amigos e colegas de pesquisa, Aurora, Jeremias, Pedro e Willian. Aurora e Pedro, obrigado por serem meus melhores amigos, obrigado por todo o apoio, pelos conselhos, este trabalho seria impossível sem o apoio de vocês. Jeremias, obrigado por todos os concelhos por todas nossas conversas. Willian, obrigado pela disponibilidade de me ajudar e pelo apoio.

Aos meus pais e familiares, Leonardo Carvalho e Lice Beatriz. Pai, obrigado por entender os motivos de ter demorado tanto para finalizar este trabalho. Mãe, obrigado por todo o apoio e por entender meus momentos em que precisava ficar recluso.

À Kassia Sayonara, obrigado por todo os conselhos durante este curso, obrigado por ter me apoiado em momentos muito difícil, cresci muito ao seu lado, boa parte da minha caminhada na Unb foi mais fácil por sua conta.

À Déborah de Sousa. Meu amor, não sei nem como te agradecer, obrigado por me apoiar a terminar este trabalho, obrigado por revisar meus textos, sem você não teria terminado este projeto. Obrigado por fazer meus dias mais felizes e me motivar a ser alguém melhor, me inspiro muito em você e tenho muito orgulho de ti, te amo.

Resumo

Com a crescente disponibilidade de equipamentos de imagens microscópicas médicas existe uma demanda para execução eficiente de aplicações de processamento de imagens digitais em alta resolução (*Whole Slide Tissue Image*) – o conteúdo completo de uma lâmina de tecido biológico. Pelo processo de análise de sensibilidade de parâmetros 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 reúso computacional. Pela realização de reúso computacional otimiza-se o tempo de execução das aplicações de análise de sensibilidade de parâmetros. Este trabalho propõem um escalonador que gerencia a execução limitando o uso de memória, de modo que possibilita que algoritmos de reuso computacional não se preocupe com a memória disponível no ambiente, ao fazer suas otimizações, se tornando eficientes em ambientes com pouca memória. Isso se mostrou eficiente na otimização da análise de sensibilidade de parâmetros, gerando uma redução, em alguns casos, de mais de 60% no tempo de execução.

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

Abstract

Due to an increasing availability of image scanning microscopy equipment, arise a demand for efficient processing of aplications that analize Whole slide Tissue Images (WSIs). These images are high resolution representations of biological tissue contents. The sensitivity analysis of parameters can improve the quality of the results of such applications. Subsequently, it can improve the quality of the analysis performed from them. Given the high computational cost and the recurring nature of the tasks performed by methods of sensitivity analysis, opportunities for computational reuse emerge. By performing computational reuse, the execution time of the sensitivity analysis applications is optimized. However, computational reuse methods can require high amounts of memory available. This poses as a limitation to computational reuse executions on limited memory environments. Therefore, this work focus is to study this limitation and to implement a solution to overcome it.

Keywords: Computational reuse, Sensitivity Analysis, Region Templates Framework

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

AS Análise de Sensibilidade.

MOAT Morris One-At-A-Time.

RMSR Runtime Memory-Aware Scheduler for Reuse.

RT Region Template.

RTF Region Templates Framework.

RTMA Reuse-Tree Merging Algorithm.

VBD Variance-Based Decomposition.

WSI Whole Slide Tissue Images.

Capítulo 1

Introdução

1.1 Motivação

Com a evolução da tecnologia surgiram *scanners* de *whole tissue image* cada vez mais rápidos e com maiores resoluções [8]. Junto a isso cresceu a demanda por análise dessas imagens. Diversas aplicações foram propostas para suprir essa demanda, muitas com um nível de sofisticação e/ou complexidade bastante elevado [5, 9, 10, 11]. Além disso surgiram novas ferramentas, plataformas para executar essa grande quantidade de dados de forma eficiente [6, 12, 13, 14, 15]. Contudo, é preciso avaliar essas aplicações de forma quantitativa para propor otimizações e melhorias, de forma a mostrar o quanto cada parte influencia na qualidade da análise feita. Deste modo conseguimos entender melhor essas aplicações. Para isso são utilizados metodologias já consagradas, que são as análises de sensibilidade de parâmetros [16].

Métodos de análise de sensibilidade de parâmetros [7] são aqueles que quantificam, correlacionam e comparam os resultados de determinados *workflows* baseados nas variações dos seus parâmetros de entrada. Este processo é feito várias vezes durante uma pesquisa científica e é capaz de identificar o quanto um parâmetro de entrada impacta no resultado de um *workflow* ou aplicação. Este processo é essencial para que possamos reduzir incertezas e variações durante uma análise e para que possamos otimizar determinados algoritmos, por exemplo, através da eliminação de partes que não causam impacto nos resultados.

Nossa principal motivação neste trabalho é otimizar a utilização de recursos na análise de sensibilidade de parâmetros em aplicações que fazem análise histopatológica [8] em *whole slide tissue images* (WSI[1]) [17]. Uma análise histopatológica típica irá extrair informações da imagem, segmentando e identificando suas formas e texturas. As características das imagens computadas por essas aplicações contém informações que podem ser

[1]Whole Slide Tissue Images (WSI) - Imagens digitais em alta resolução de lâminas de tecidos biológicos.

usadas para desenvolver modelos morfológicos dos espécimes em estudo para obter novas ideias sobre mecanismos de uma doença e avaliar sua evolução [8].

Apesar dos benefícios que advém da análise de sensibilidade de parâmetros, o seu uso na prática é limitado, dados os desafios computacionais associados a ele. Por exemplo, um estudo usando um método clássico como Variância baseado em Decomposição (VDB) [18] pode exigir de centenas até milhares de execuções por parâmetro. O processamento de uma única *whole slide tissue image* vai extrair por volta de 400.000 núcleos e pode demorar horas em um único nó computacional. Um estudo de análise de sensibilidade de parâmetros irá considerar centenas destas imagens e calculará milhões de núcleos por execução. Assim, é inviável fazer uma análise deste porte com execução sequencial.

O *Region Template Framework (Region Templates Framework (RTF))* [6] é um ambiente de execução de alto desempenho para sistemas distribuídos em ambientes híbridos, plataforma a qual foi utilizada para a execução da aplicação utilizada neste trabalho. Nele as aplicações são representadas a partir de uma estrutura hierárquica que descreve os *workflows* de suas tarefas em dois níveis: o primeiro, grão grosso (que chamaremos de estágios), que por sua vez são compostos de tarefas de grão fino (ou simplesmente chamadas de tarefas). Isso é de suma importância, pois é a partir desta estrutura hierárquica que são avaliadas algumas oportunidades de otimizações, via reúso computacional.

As informações geradas com análise de sensibilidade de parâmetros são computadas pela execução e avaliação do mesmo *workflow* variando sistematicamente os parâmetros de entrada, tal que, são vários os conjuntos ou subconjuntos de parâmetros com valores similares. Como tarefas com os mesmos parâmetros geram os mesmos resultado, é possível selecionar dois ou mais *workflows* que tenham tarefas em comum, uni-los e executar estas tarefas apenas uma vez, reusando-as, e por consequência evitando desperdício de processamento. Assim, gera-se um novo *workflow* com menos tarefas a serem executalas. Este tipo de otimização pode ser feito tanto com estágios quanto com tarefas, como mostrado por Barreiros em [2], que propôs diversos algoritmos que avaliam, as oportunidades de reúso nas estruturas representadas no *Region Template Framework*. Isso se mostrou bastante eficaz para melhorar o desempenho da análise de sensibilidade de parâmetros.

1.2 Definição do problema

Algoritmos para a otimização da análise de sensibilidade de parâmetros [2, 19, 20] através reúso computacional, ao unir estágios para o reaproveitamento de tarefas geram *workflows* que requer mais memória na sua execução. Assim, se limita a otimização pela quantidade de memória disponível.

1.3 Objetivo

O objetivo deste trabalho é permitir que otimizações através de reúso computacional sejam feitas, sem causar grande impacto no uso de memória do ambiente. Dessa forma, possibilitando que otimizações que antes só seria possíveis em ambiente com uma grande quantidade de memória, agora seja feita em ambientes com memória limitada.

1.4 Contribuição

As principal contribuição deste trabalho de graduação é:

1. Propõe um escalonador por restrição de memória, o qual possibilita que algoritmos que fazem reúso computacional a grão fino tenham melhores resultados em ambientes com memória limitada (Seção3.1);

A contribuição acima têm como objetivo tornar viável a utilização de otimizações por reúso computacional em ambiente com memória limitada. O algoritmo proposto atua como um escalonador nas tarefas de um *workflow* gerado pela união entre *workflows* no processo do reúso computacional. Controlando a ordem e o momento da execução de cada tarefa, torna-se possível a execução de todas as tarefas do *workflow*, sem ultrapassar o limite da memória disponível. Desse modo, essa organização na execução consegue limitar o uso de memória máximo do *workflow* em detrimento do paralelismo, e consequentemente, o processo de avaliação do reúso computacional pode fazer algumas escolhas de otimizações sem necessariamente se limitar a memória disponível.

1.5 Organização do trabalho

A organização deste trabalho se dá da seguinte forma. No Capítulo 2, Referencial Teórico, são apresentados conceitos sobre análise histopatológica com imagens, análise de sensibilidade de parâmetros, assim como o funcionamento e características do *Region Template Framework*; em seguida, é mostrada a definição de reúso computacional, e por fim é apresentado o algoritmo que será base para este trabalho, e suas limitações. No Capítulo 3, é apresentado o escalonador proposto para suprir esta limitação, e finalmente é mostrado como foi feita a integração deste escalonador ao *Region Template Framework*, a contribuição prática desse projeto. No Capítulo 4, são discutidos os resultados obtidos, comparando-se os ganhos das implementações deste trabalho em relação ao algoritmo original, que chegaram a ser de 65% no tempo de execução. Por fim, no Capítulo 5 são apresentadas as conclusões, as contribuições, e as perspectivas para a continuação do trabalho desenvolvido.

Capítulo 2

Referencial Teórico

Este capítulo descreve conceitos fundamentais, assim como as ferramentas utilizadas nesse trabalho. A primeira seção deste capítulo apresenta o que é *whole slide tissue images*, mostrando os avanços tecnológico na área. A Seção 2.2 explica o conceito de análise hipopatológica com imagens, mostrando trabalhos feitos na área e seus benefícios e desafios. A Seção 2.3 descreve o que é a análise de sensibilidade de parâmetros. Na Seção 2.4, descreve-se o *Region Templates Framework*, uma plataforma de execução distribuída na qual foram executadas as aplicações e suas análises de sensibilidade. Na Seção 2.5 referencia diversos trabalhos que o usa reúso computacional em áreas distintas. Por fim, a Seção 2.6, define o que é chamado de reúso computacional, bem como explica-se o seu princípio de funcionamento; além disso, é introduzido um algoritmo que foi utilizado como base do desenvolvimento deste trabalho.

2.1 Imagens digitais de lâminas de tecidos biológicos

Na última década, imagem digital em patologia foi significativamente impactada pelo desenvolvimento da tecnologia de *whole slide imaging* [21, 22]. Os scanners de *whole slide tissue images (WSI)* automatizados são microscópios capazes de digitalizar uma lâmina de vidro, produzindo uma imagem digital do tecido ali inserido em alta resolução (Figura 2.1). Os componentes críticos de um dispositivo de captura *Whole Slide Tissue Images (WSI)* automatizado incluem o hardware (scanner composto por um microscópio óptico e câmera digital conectada a um computador), software (responsável pela criação e gerenciamento de imagens, visualização de imagens e análise de imagens quando aplicável) e conectividade da rede. A tecnologia evoluiu ao ponto de que scanner são capazes de produzir imagens de alta resolução em um tempo relativamente curto [23]. A imagem digital pode representar uma lâmina de vidro inteira ou uma área selecionada pelo usuário da lâmina de vidro e é geralmente referida como uma imagem de *whole slide image*. Após a

recuperação do arquivo digital, a imagem capturada pode ser visualizada em um monitor de computador sem o uso de um microscópio real. A interface do software usada para visualizar as lâminas digitalizadas simula a operação da microscopia de luz. Vários tipos de scanners *WSI* foram desenvolvidos por fornecedores, todos capazes de produzir imagens digitais de lâminas inteiras automatizadas, em alta velocidade e de alta resolução [24, 25].

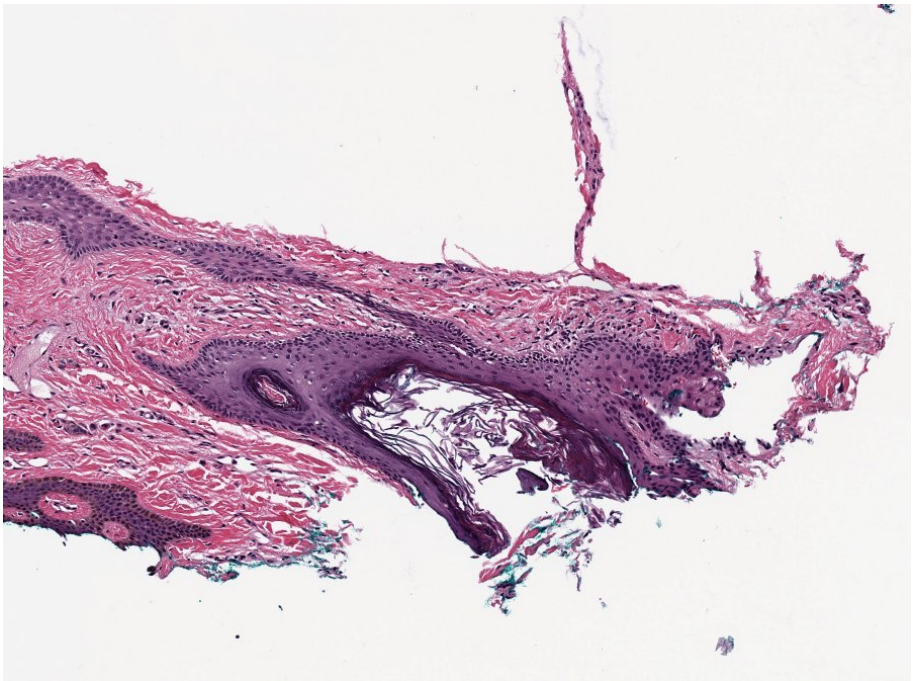

Figura 2.1: Exemplo de imagem de lâmina de tecido biológico (Fonte: [1])

2.2 Análise histopatológica com imagens

A histopatologia é uma área de estudo da anatomia patológica. A partir da análise microscópica é possível identificar alterações e anormalidades estruturais dos tecidos e células. As alterações morfológicas presentes nesses tecidos fornecem informações valiosas sobre a saúde do paciente, auxiliando no diagnóstico de doenças [8].

Com o aumento do poder computacional, melhorias nos algoritmos de processamento de imagens e o advento de scanners de *WSI*, tornou-se possível realizar análises histopatológica assistidas por computador. Algoritmos começaram a ser desenvolvidos para facilitar a detecção de doenças, diagnóstico e prognóstico, complementando a opinião do patologista [5]. Com o crescente número de grupos de pesquisa trabalhando para desenvolver tais algoritmos [8, 26, 27, 28, 9], há uma alta demanda por sistemas ou *frameworks* orientados para uma execução eficiente dos *workflows* para a análise histopatológica.

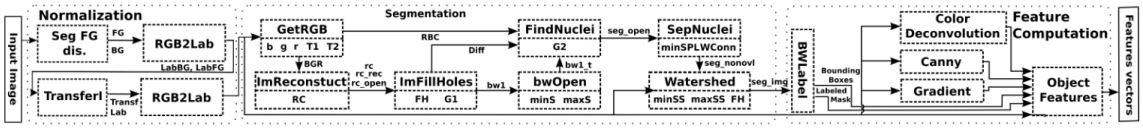

Figura 2.2: Exemplo de workflow usado na análise histopatológica. (Fonte: [2, 3])

O *workflow* de análise histopatológica usado neste trabalho está apresentado na Figura 4.1 e foi proposto por [17, 29]. Esse *workflow* consiste nas seguintes etapas:

1. Normalização: para obter uma redução de ruído e exposição homogênea para que a etapa de segmentação possa ser utilizada de forma mais consistente;

2. Segmentação: para se obter regiões de interesse e separar estruturas que serão avaliadas do fundo da imagem.

3. Extração de características: a partir das regiões de interesse as estruturas são mensuradas, como por exemplo: Circularidade e perímetro do núcleo de uma célula.

4. Avaliação ou classificação: onde as estruturas identificadas e caracterizadas são avaliadas e agrupadas segundo o modelo estabelecido, para que possa gerar informação que será utilizada para auxiliar o diagnóstico do médico.

A primeira etapa é responsável pela normalização da coloração e/ou condições de iluminação da imagem (Figura 2.3). Esta etapa faz parte do pré-processamento e não está preocupada em extrair informações da imagem, mas sim em melhorar os dados da mesma, de modo a diminuir distorções ou ressaltar características que serão importantes no restante do processamento [30]. Já a segmentação é o processo de identificação do núcleo de cada célula a ser analisada na imagem (Figura 2.4). Na próxima etapa, extração de características (Figura 2.5), um conjunto de características de forma e textura é gerado para cada núcleo segmentado. Essas características possuem relação direta com aquelas procuradas por patologistas durante exames visuais. Finalmente, para a caracterização final (Figura 2.6) são tipicamente utilizados algoritmos de mineração de dados para informações coletadas anteriormente de modo a destacar informações úteis que auxiliarão, dentre outras coisas, na distinção do subtipo de uma doença [5].

A qualidade da análise do *workflow* depende diretamente da qualidade dos parâmetros de entradas (Figura 2.7), os parâmetros são tanto referentes aos limiares característicos da imagem, como cor de fundo e cor de célula, como parâmetros de entrada para o controle dos algoritmos usados no *workflow*. Por exemplo os descritos na Tabela 2.1. Então, a fim de melhorar a análise e entender o impacto destes parâmetros no resultado, é feito um estudo de análise de sensibilidade de parâmetros, que será detalhado na seção seguinte.

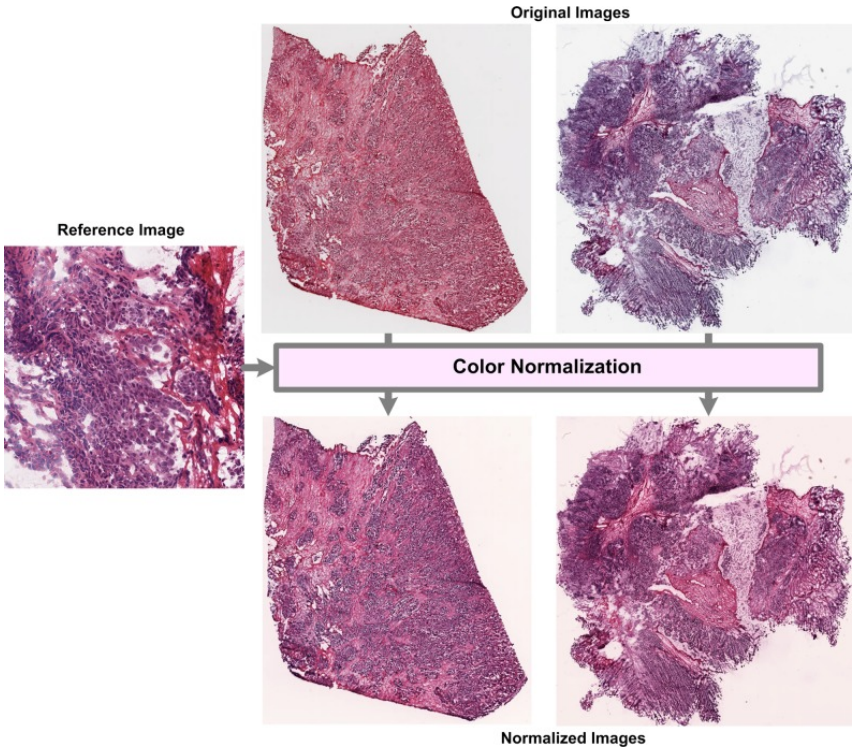

Figura 2.3: Exemplo de normalização de cor em uma amostra provida pelo *NIH Cancer Genome Atlas* (Fonte: [4])

2.3 Análise de sensibilidade

Segundo Andrea Saltelli, análise de sensibilidade é: "*The study of how uncertainty in the output of a model (numerical or otherwise) can be apportioned to different sources of uncertainty in the model input.*"[1]

Chamaremos de análise de sensibilidade (Análise de Sensibilidade (AS)) o processo de quantificar, comparar e correlacionar os parâmetros de entrada e o resultado de um *workflow*, a fim de quantificar o impacto de cada entrada no resultado final [31]. Este processo é realizado durante várias fases de uma pesquisa científica, mas não se limita à validação de modelos, estudos de otimização e estimativas de erros [32].

Normalmente o custo computacional para a realização da análise de sensibilidade para um determinado *workflow* é diretamente proporcional ao número de parâmetros que este possui, bem como ao espaço de busca de cada parâmetro. Uma maneira de simplificar a análise em aplicações com um grande número de parâmetros - reduzindo assim seu custo - é removendo parâmetros cujo efeito na saída não é significativo.

[1] O estudo de como a incerteza na saída de uma modelo (numérico ou não) pode ser dividido para diferentes fontes de incerteza na entrada do modelo. (tradução livre)

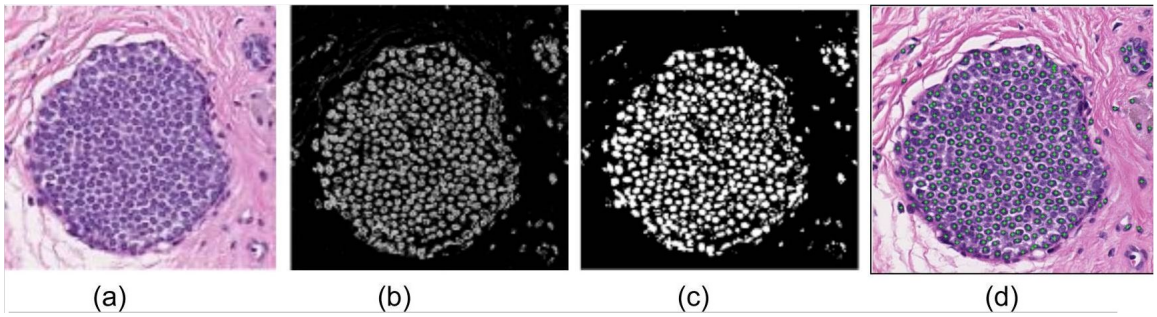

Figura 2.4: a) Uma *WSI*. b) c) e d) a imagem depois de aplicadas as operações de segmentação. (Fonte: [5])

Este processo de análise em várias etapas é abordado em [32] como alternativa para análises dispendiosas. O estudo de caso deste trabalho usa um modelo complexo com vários parâmetros de entrada (Tabela 2.1) e um alto custo de execução. Como tal, recomenda-se que um método mais leve seja executado preliminarmente em todos os parâmetros, como forma de reduzir a complexidade da análise. Após a execução deste método, estratégias de análise mais complexas podem ser executadas num subconjunto dos parâmetros de entrada. Os métodos de AS escolhidos para esse trabalho foram o *Morris One-At-A-Time (Morris One-At-A-Time (MOAT))* [33] para a primeira etapa e o *Variance-Based Decomposition (VBD)* [18] para a análise mais completa.

O método mais leve, *Morris One-At-A-Time (MOAT)* [33], executa uma série de vezes a aplicação alterando cada parâmetro individualmente, enquanto fixa os parâmetros restantes em um espaço de busca discretizado. Cada um dos k parâmetros analisados é dividido uniformemente em p níveis, assim resultando em p^k conjuntos de parâmetros a serem avaliados. A cada resultado de uma avaliação x_i da aplicação é criada um parâmetro efeito elementar (EE), calcula-se $EE_i = \frac{y(x_1,\ldots,x_i+\Delta_i,\ldots,x_k)-y(x)}{\Delta_i}$, sendo $y(x)$ o resultado da aplicação antes da alteração do parâmetro. Para contabilizar o resultado global da AS o *RTF* usa $\Delta_i = \frac{p}{2(p-1)}$ [7]. O método *MOAT* requer $r(k+1)$ avaliações, onde r vai de um intervalo de de 5 a 15 [34].

O segundo método de AS, *Variance-Based Decomposition (Variance-Based Decomposition (VBD))* é preferencialmente realizado depois de um método de AS mais leve. Isso é feito pois o *Variance-Based Decomposition (VBD)* requer $n(k+2)$ execuções da aplicação para k parâmetros e n amostras, com n na ordem de grandeza de milhares de execuções [18]. Portanto, para ser viável, é interessante usar um numero reduzido de parâmetros. *Variance-Based Decomposition (VBD)*, ao contrario do *Morris One-At-A-Time (MOAT)*, diferencia o efeito da incerteza da saída em parâmetros individuais (primeira ordem) e efeitos de alta ordem.

Independentemente do método usado para a AS escolhido, quando se usa um conjunto

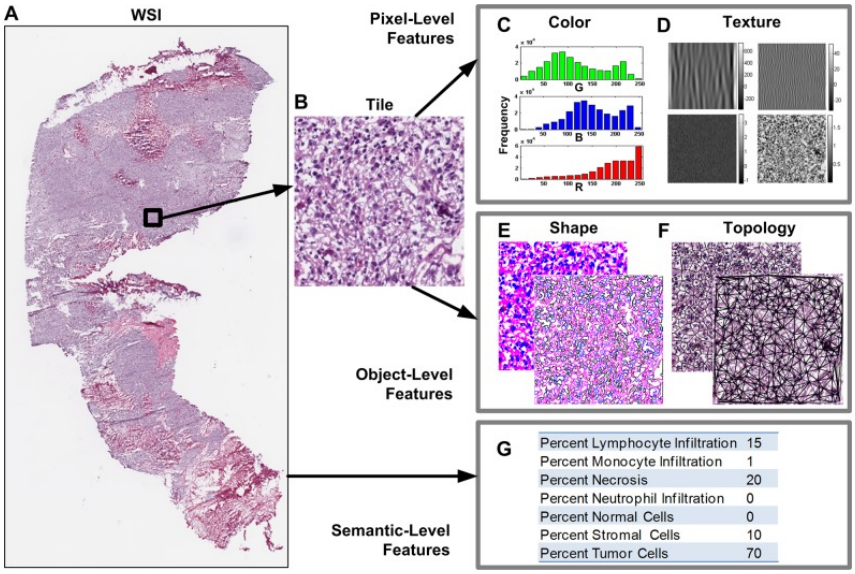

Figura 2.5: Extração de características de (B) uma partição da imagem. Características a nível quantitativo como (C) histograma de cores; a nível de objeto, como (E) formas segmentadas; e a nível semântico, como (G) porcentagem de propriedades clínicas. (Fonte: [4])

grande de parâmetros (Figura 2.1) não é prático executar a AS devido ao alto custo de se avaliar um domínio de busca tão grande. Para possibilitar a realização da AS no *workflow* apresentado, podemos executar a análise em um ambiente de computação distribuída. Além disso, o reúso computacional pode ser empregado para reduzir o custo computacional sem a necessidade de otimizações específicas [2]. Ambos os métodos mencionados são abordados nas seções seguintes.

2.4 Region Templates

O *Region Templates Framework* (Region Templates Framework (RTF)) é um sistema de execução de alto desempenho para ambientes distribuídos e híbridos [6]. Seus principais módulos são: plataforma de execução, abstração de dados do *Region Templates* (Figura 2.9) e diferentes mecanismos de persistência.

2.4.1 Abstração de dados

O RTF representa e executa fluxos de tarefas de uma aplicação. Além disso, implementa uma estrutura hierárquica onde descreve tais fluxos em dois níveis, o primeiro para tarefas

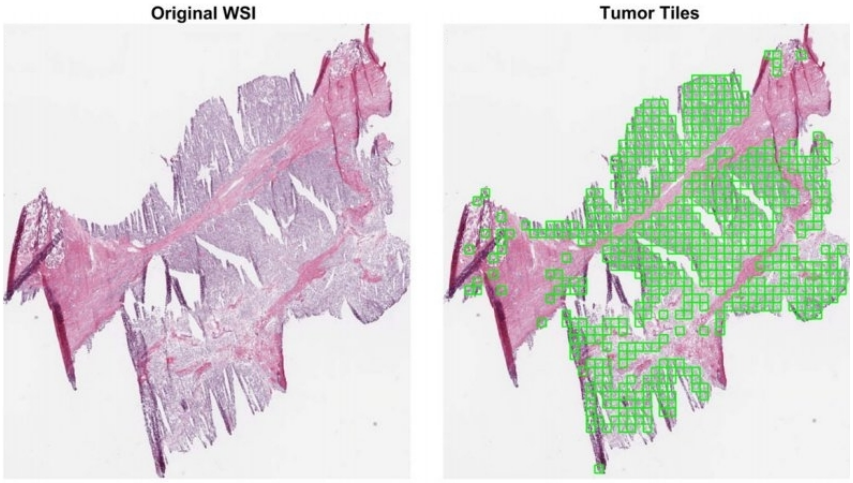

Figura 2.6: Classificação das áreas que foram identificados possíveis tumores. (Fonte: [4])

Tabela 2.1: Lista dos 15 parâmetros do estágio de segmentação da aplicação baseada em Operações Morfológicas. O espaço de busca desse estágio é de aproximadamente 21,4 trilhões de pontos (Fonte: [7]).

Parâmetros	**Descrição**	**Intervalo de valores**
B/G/R	Background detection thresholds	$[210, 220, ..., 240]$
T1/T2	Red blood cell thresholds	$[2.5, 3.0, ..., 7.5]$
G1	Thresholds to identify candidate nuclei	$[5, 10, ..., 80]$
G2	Thresholds to identify candidate nuclei	$[2, 4, ..., 40]$
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]

de grão grosso (que chamaremos de estágios), que por sua vez são compostas de tarefas de grão fino (ou simplesmente chamadas de tarefas) (Figura 2.10). Essa implementação permite maior flexibilidade e melhor desempenho no escalonamento em sistemas híbridos.

Dados consumidos e produzidos pelos componentes da aplicação são gerenciados em contêineres de armazenamento fornecidos pelo *region templates data abstraction*. Os tipos de dados suportados nessa abstração incluem as estruturas de dados que descrevem espaços de baixa dimensionalidade (1D, 2D ou 3D) com um componente temporal. A implementação suporta pixels, pontos, vetores, matrizes, volumes 3D, polígonos e superfícies que podem representar objetos segmentados e regiões. O *region templates data abstraction* implementa um mecanismo eficiente de transferência de dados entre componentes da aplicação, que pode ser executados em diferentes nós de uma máquina de memória

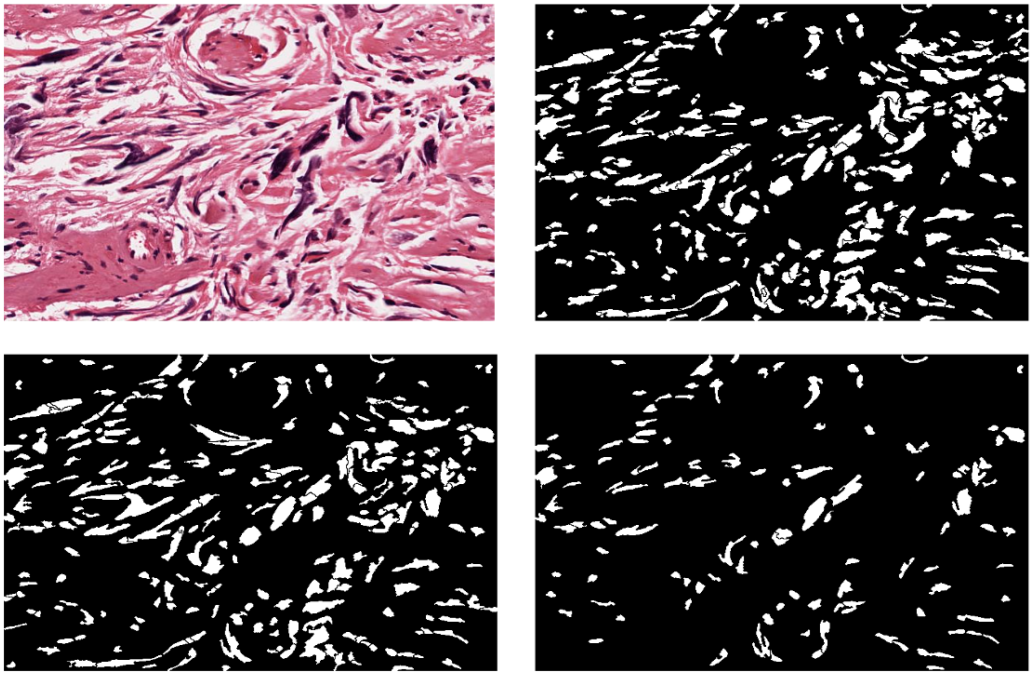

Figura 2.7: Pedaço de uma *Whole Slide Tissue Images (WSI)* com resultados do processo de segmentação, com diferentes parâmetros

distribuída. Ao invés de gravar os dados em um fluxo de dados (*stream*) ou enviar dados diretamente entre estágios, como em uma típica aplicação de fluxo de dados, os componentes da aplicação inserem os dados em uma instância do *region template data* e, assim, outros componentes da aplicação podem consumir esses dados.

As dependências entre componentes da aplicação e a instância do *region template data* são fornecidas pela própria aplicação. As transferências de dados são realizadas em segundo plano por *threads* de E/S, que interagem com as implementações apropriadas do armazenamento do *region template data* para recuperar e/ou armazenar os dados. No sistema de execução que coordena as transferência de dados respeitando as dependências entre os componentes da aplicação, quando completa-se a transferência, códigos apropriados da aplicação para o processamento dos dados são iniciados, respeitando as políticas de escalonamento implementadas.

2.4.2 Sistema de execução

O sistema de execução implementa o modelo *Manager-Worker*, como visto na Figura 2.11, em que um conjunto de estágios é gerenciado pelo *Manager* e processado nos *Workers*. O sistema funciona com os *Workers* solicitando estágios para o *Manager* quando eles tem disponibilidade para executá-los, até que todos os estágios sejam finalizados. O *Manager*

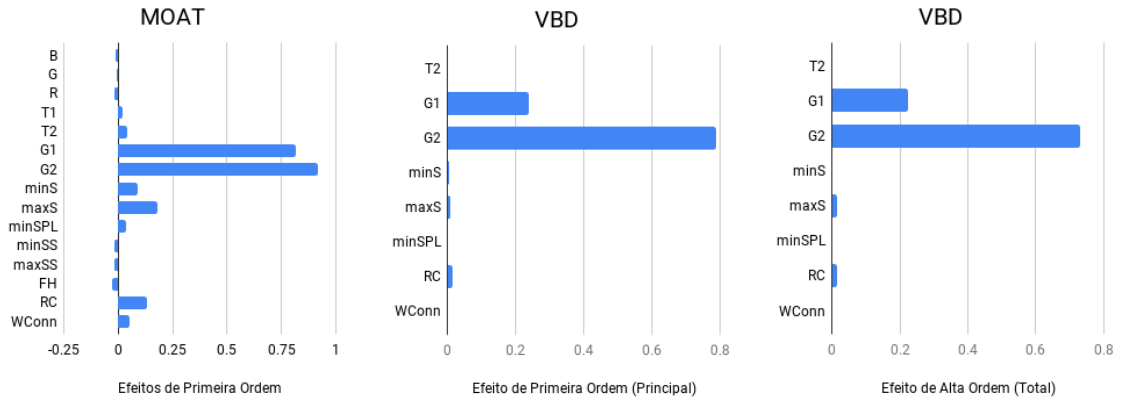

Figura 2.8: Exemplo de resultado da análise de sensibilidade do *MOAT* com todos os 15 parâmetros, e do *VBD* com os 8 parâmetros mais influentes.

Figura 2.9: Arquitetura do *Region Templates*. (Fonte: [6])

escalona e envia esses estágios para os *Workers*, apenas em nível de grão-grosso e conforme forem requisitados.

Ao receber um Estágio, o *Worker* deve alocar memória para os *data regions* que estão especificados no *RT* e requisitar os *data regions* ao sistema de armazenamento. O *Worker* também é responsável por gerenciar todas as tarefas de um estágio designado a ele (garantindo o correto fluxo de dados entre essas tarefas, como visto na Figura 2.10). Tal gerenciamento é feito utilizando um modelo de *Thread Pool*, no qual todas as tarefas são consumidas de uma fila de execução gerada considerando as dependências entre as tarefas. Desta forma os fluxos de dados dos estágios podem estar sendo executados em múltiplas máquinas, mas cada estágio é alocado em apenas uma máquina [6].

O sistema de execução instancia os componentes, estágios e tarefas da aplicação e executa o escalonador para gerenciar as suas execuções; e quando existem relações de dependência entre esse componentes, estas são respeitadas, para uma correta execução da

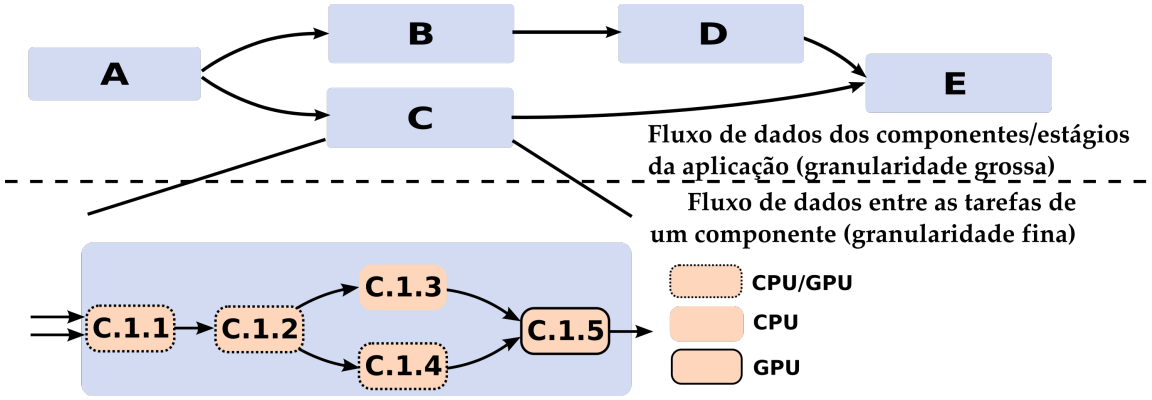

Figura 2.10: Modelo de aplicação de fluxo de trabalho hierárquico usado no *Region Templates*. (Fonte: [6])

aplicação. O escalonamento para os estágio pode ser feito entre máquinas com memória distribuída, ao passo que para as tarefas, isto é feito em um único nó computacional. Então, dado um ambiente de n nós, cada nó com k núcleos, um estágio qualquer deve ser executado em um único nó, com suas tarefas sendo executadas em qualquer um dos k núcleos. Note que, estando as dependências devidamente respeitadas, não só estágios podem ser executados em paralelo em cada um dos n nós, como as tarefas de um mesmo estágio podem ser paralelizadas em cada um dos k núcleos.

Em resumo, o *Region Templates Framework (RTF)* é um sistema de execução de alto desempenho para sistemas distribuídos híbridos. Ele oferece estruturas de dados otimizadas, que podem ser utilizadas por desenvolvedores e usuários de aplicações médicas para processar, sob demanda, grandes conjuntos de imagens de microscopia.

2.5 Trabalhos relacionados

A análise de sensibilidade de parâmetros pode otimizar o processamento de imagens e melhorar seus resultados. O trabalho de Teodoro *et al*[7] propôs uma solução paralela que realizou experimentos com dois *workflows* de segmentação que faziam um *pruning* para definir parâmetros que não influenciavam nos resultados. Também descobriu que a pesquisa em um pequeno espaço de busca pode resultar na melhora efetiva dos resultados de segmentação.

Além do uso do paralelismo computacional, é possível utilizar técnicas de reúso computacional para diminuir custos da análise de sensibilidade de parâmetros. O reúso computacional provoca agrupamento de *workflows* que contêm tarefas em comum[2, 6].

No geral, o reúso computacional de dados pode ser feito através de árvores de reúso. Árvores são boas estruturas de dados para representar reúso e tem sido usadas em diversos

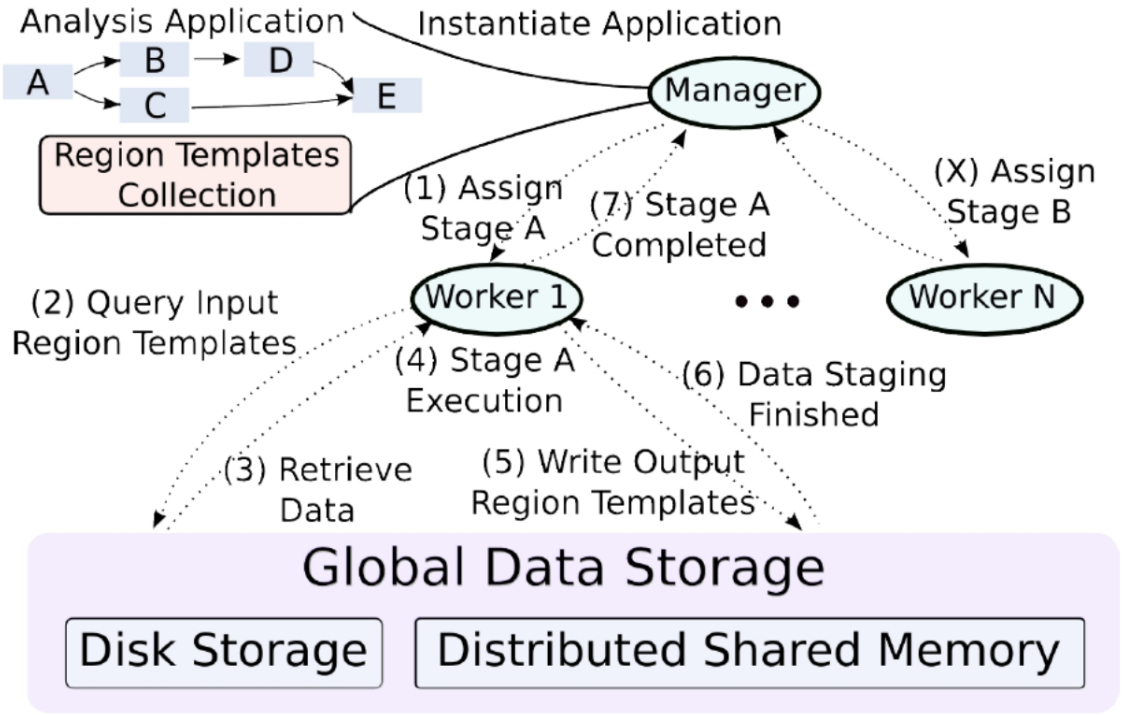

Figura 2.11: Diagrama mostrando a execução de uma coleção de regiões de dados em um sistema de execução seguindo um modelo *Manager-workers*. (Fonte: [6])

contextos que permeiam a área. Liu *et al*[35], por exemplo, se utilizaram delas para explorar o reúso de dados em plataformas FPGA aplicadas para a busca por movimentos no processamento de vídeos.

Já o trabalho de Barreiros *et al*[2] aplica o reúso computacional em um contexto bem diferente do trabalho [35], na área de processamento de imagens médicas. Essa área é o foco deste trabalho. Motivada pela grande variedade de reúsos computacionais existentes, Barreiros propôs uma taxonomia para o reúso computacional que pode ser vista abaixo:

- Nível de implementação: de *software* ou de *hardware*;
- Flexibilidade da aplicação: geral (não específica a um domínio), parcial (parcialmente específica a um domínio) ou de domínio específico;
- Estratégia de reúso: abordagem preditiva (prevê valores dados resultados anteriores salvos em *buffer*), de memorização (guarda resultados em cache para reúso posterior) ou analítica (as oportunidades de reúso são mapeadas antes da execução);
- Granularidade de tarefas: em nível de instrução CPU, grão-fino, grão-grosso e a aplicação total;

- Correspondência de tarefas utilizadas: idênticas (parâmetros de entrada iguais) ou similares (parâmetros de entrada equivalentes, mas não necessariamente iguais);
- Avaliação de reúso: dinâmica (durante execução) ou estática (antes da execução);
- Requerimento de treinamento: requer treinamento prévio à aplicação ou não requer treinamento prévio à aplicação;
- Escala de ambiente de reúso: as tarefas podem ser reusáveis em ambientes distribuídos ou locais;

Nesse contexto, categorizam-se as soluções de reúso propostas por Barreiros *et al*[2] em nível de *software* e como sendo parcialmente atreladas ao domínio do processamento de imagens *Whole Slide Tissue Images (WSI)*, sendo implementadas junto ao *Region Templates Framework (RTF)*. Isso pode ser visto no *workflow* da *workflow*, que mostra os ciclos da análise de sensibilidade realizada, sendo nessa análise aplicadas as soluções de reúso multinível para criação de árvores compactas, cujos resultados finais devem ser comparados dados os resultados de referência da análise da imagem.

Além disso, as soluções do trabalho [2] também possuem uma estratégia de reúso analítica, que aproveita estágios de grão-grosso e tarefas de grão-fino. A correspondência de tarefas deve ser idêntica e a avaliação de reúso é estática. E elas não requerem treinamento, funcionando em uma escala de ambiente dinâmica.

O presente trabalho tem como propósito melhorar o trabalho de Barreiros[2], para que ele possa ser executado em situações com memória restrita. Assim, quanto a taxonomia, pode-se também categorizar este trabalho como sendo em nível de software, parcialmente atrelado a um domínio, com uma estratégia de reúso analítica que necessita que a correspondência de tarefas seja idêntica e que a avaliação de reúso seja estática (sem requerer treinamento).

2.6 Reúso computacional multinível

Essa seção apresenta uma otimização da AS a partir de reúso computacional. A análise de sensibilidade executa o *workflow* de uma aplicação várias vezes, variando seus parâmetros de entrada e relacionando isto com o resultado da aplicação. Entre essas diversas execuções pode haver casos de computação comum, a qual poderia ser reutilizada para acelerar a análise. Uma computação comum é aquela que utiliza os mesmos dados e parâmetros de entrada, e por consequência terá o mesmo resultado de saída. A Figura 2.12 mostra duas formas de instanciar o *workflow* de uma aplicação no estudo de AS que executará para os diversos conjuntos de parâmetros mostrado. A primeira, *replica based composition*,

não considera o possível reúso, simplesmente instancia um *workflow* para cada conjunto de parâmetros. Já a segunda, *compact composition* une estágios que possui tarefas em comum, executando essas tarefas apenas um vez, reutilizando seus resultados.

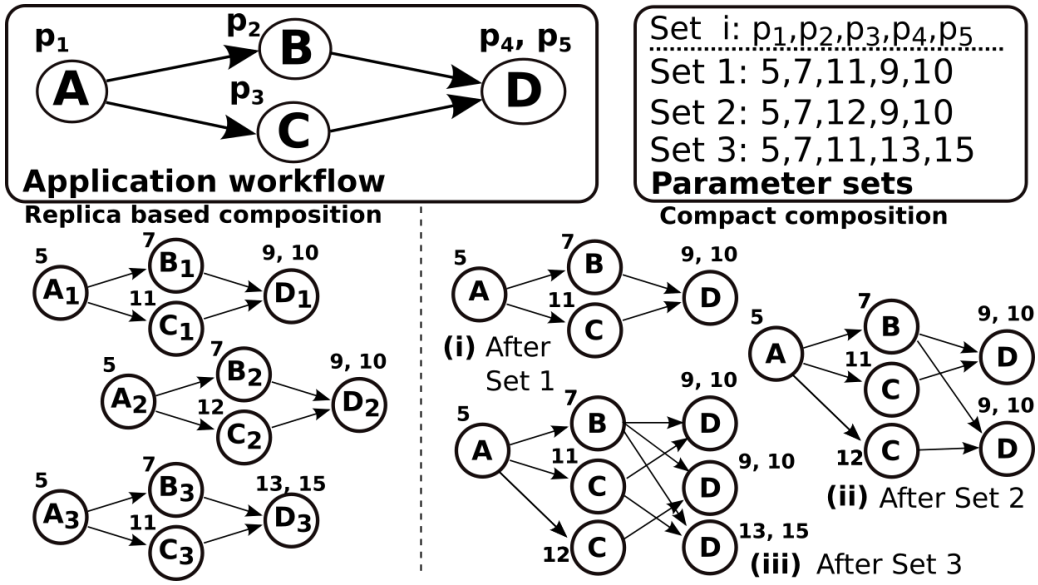

Figura 2.12: Comparação entre *workflows* que não se utiliza de reúso e os mesmos *workflows* unidos para se aproveitar de possíveis reúsos (Fonte: [2])

Pela representação hierárquica dos *workflows* suportada pelo *RTF* é possível se aproveitar desse tipo de reúso computacional tanto a nível de estágios quanto a nível de tarefas. Todavia, para reaproveitar estágios, os conjuntos de parâmetros usados pelas duas cópias devem ser iguais. Por outro lado, para reúso a grão fino, basta um subconjunto de parâmetros em comum para entrada de algumas tarefas do *workflow* para se unir estágios e reaproveitar tarefas entre eles. Basicamente, um reúso a grão grosso seria um em que todas as tarefas de grão fino entre dois estágios puderam ser reaproveitadas.

Em [2] Barreiros propôs métodos para calcular e se aproveitar do reúso computacional, tanto a grão grosso como a grão fino, desde algoritmos mais simples a algoritmos rebuscados e complexos. Isso se mostrou de grande eficacia na otimização da análise de sensibilidade.

Este trabalho se concentra no reúso a grão fino, mais especificamente no apresentado pelo *Reuse-Tree Merging Algorithm (Reuse-Tree Merging Algorithm (RTMA))*, algoritmo o qual teve os melhores resultados em [2]. Esse algoritmo junta estágios em uma única estrutura *(bucket)* para poder reutilizar as tarefas desses estágios que são passíveis. Para isso, leva-se em conta um parâmetro, *maxBucketSize*, passado na execução do *Region Template*, que define quantos estágios, no máximo, podem estar dentro do mesmo *bucket*.

O trabalho proposto neste documento desenvolve técnicas para evitar que esse parâmetro seja uma limitação nas otimizações da AS.

2.6.1 Limitações do *Reuse-Tree Merging Algorithm*

Esta seção tem como objetivo evidenciar a limitação do algoritmo *Reuse-Tree Merging Algorithm (RTMA)*, que resulta no desperdício de reúsos computacionais possíveis, principalmente em ambientes onde a memória disponível é limitada.

O *Reuse-Tree Merging Algorithm (RTMA)* descreve instâncias dos estágios usando uma árvore, onde cada conjunto de parâmetros do *workflow* é organizado pelas dependências das tarefas. A Figura 2.13 mostra como seria esta estrutura. Para o *workflow*, existem as tarefas de grão grosso S_1 a S_{12}, e cada uma delas é composta de tarefas de grão fino A, B e C com os parâmetros P_1, P_2 e P_3 respectivamente. Na árvore, cada tarefa é representada como um nó, de forma que caso se siga da raiz até uma folha, então se está representado todas as tarefas que um certo estágio possui, e isto é construído da seguinte forma:

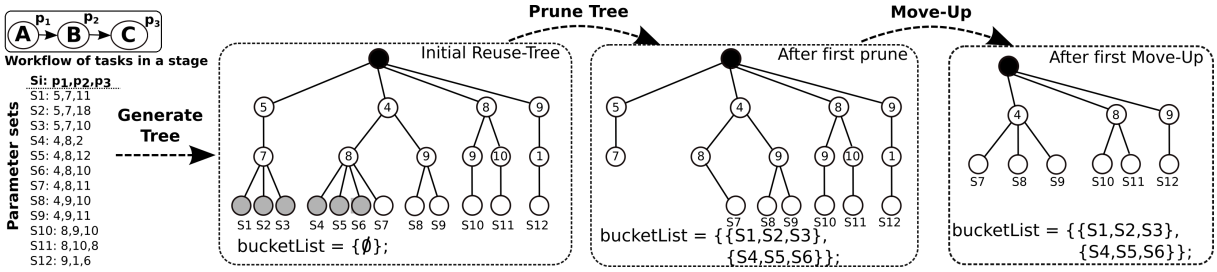

Figura 2.13: Esquemático do funcionamento do algoritmo *Reuse-Tree Merging Algorithm (RTMA)* (Fonte: [2])

- todo início de um *workflow* é filho do mesmo nó *init*;
- as tarefas que dependem de outra tarefa no *workflow* são filhas do nó que representa essa tarefa com um certo valor de parâmetro. Neste caso, as tarefas B dependem de tarefas A, então essas são filhas de A, em especial cada uma é filha de um A específico com o valor do parâmetro que está em seu estágio. Por exemplo, nos estágios S_7 e S_8, as tarefas B dependem de uma tarefa A cujo parâmetro P_1 é igual a 4; desta forma, em ambos a tarefa B é filha do mesmo nó.

Destaca-se que esta construção é feita da raiz em direção às folhas, uma vez que uma tarefa B com $P_2 = 9$ que depende de uma tarefa A com $P_1 = 4$ ocasiona em um resultado diferente de outra que depende de uma tarefa A com $P_1 = 8$.

Desta forma, as oportunidades de reúso já estão separadas, bastando que se una os estágios em que alguma tarefa possui pai em comum. Neste ponto é possível reutilizar

tal pai, de modo que essa união começa a ser realizada pelas folhas para que se consiga obter o máximo de reúso entre os estágios unidos. Por exemplo, foi colocado no mesmo *bucket*, isto é, foram unidos os estágios S_1, S_2 e S_3, e desta forma as tarefas A e B desses estágios serão reutilizadas. Assim, na totalidade, onde seriam executadas 9 tarefas, 3 vezes a tarefa A como parâmetro 5, três vezes a tarefa B com entrada 7, e as tarefa C com entradas 11,18 e 10 ($3xA^5$, $3xB^7$, C^{11}, C^{18} e C^{10}), agora serão executadas 5 tarefas, a tarefa A e B apenas uma vez, e a tarefa C três vezes (A^5, B^7, C^{11}, C^{18} e C^{10}).

Após fazer a união em conjunto de *bucket* com um número de estágios preestipulado por uma variável passada para a execução *maxBucketSize* - que no caso do exemplo é igual a 3 -, as folhas que não foram unidas se tornam filhas do seus avós, e a união prossegue. Neste sentido, este passo pode ser visto observando-se o último quadro da Figura 2.13. A partir daí será criado o *bucket* com os estágios S_7, S_8 e S_9, que apesar de não conseguirem reaproveitar totalmente a tarefa B -reaproveitada somente entre S_8 e S_9 - todos eles tem a tarefa A em comum então é vantajoso uni-los. Serão unidos também S_{10} e S_{11} pelo mesmo motivo, e S_{12} ficará em um *bucket* sozinho, já que ele não possui nenhuma tarefa em comum com outro estágio.

Perceba que seria interessante que o estágio S_7 ficasse junto dos estágios S_4, S_5 e S_6, pois assim poderia reusar as tarefas A e B, porém por conta da limitação do *maxBucketSize* isso não ocorre, fazendo com que a tarefa B com $P_2 = 8$ seja executada duas vezes. Quando isso acontece dizemos que uma oportunidade de reúso foi perdida. Análogo acontece com a tarefa A com $P_1 = 4$ e os estágios S_4 a S_9.

É factível interpretar que, quanto maior o *maxBucketSize*, mais oportunidades de reúso poderão ser aproveitadas. Porém, cada *bucket* é instanciado como um único estágio. Quando se unem n estágios, substitui-se a execução de vários estágios com um número T de tarefas cada, por um único estágio com T' tarefas, sendo $T' < n * T$. Essa diferença representa a quantidade de tarefas que não serão executadas devido ao reúso.

Como já visto na Seção 2.4, um estágio executa todas suas tarefas em um mesmo nó computacional, o mesmo vale para o *bucket*. Essa característica impõe um limite no valor do *maxBucketSize*, para que seja possível executar esse *bucket* com a memória disponível, porém muitas vezes a memória disponível em um nó não é suficiente para *buckets* grandes.

Este trabalho se propõe em escalonar um *bucket* para executar as tarefas da árvore em profundidade limitando a execução paralela entre galhos, permitindo assim a execução do *workflow* em ambiente onde a memória é um limitante. Desta forma permite que *buckets* maiores sejam gerados, dando mais possibilidade para se aproveitar de oportunidades de reúso. Nas próximas seções será explicado como foi feito o escalonador para esse fim e serão discutidos os resultados obtidos.

Capítulo 3

Escalonamento sensível ao gasto de memória

O principal objetivo desse trabalho é otimizar a execução da análise de sensibilidade através do aumento de reúso em comparação com *Reuse-Tree Merging Algorithm (RTMA)* desenvolvido por Barreiros W. em [2]. Desta forma esse capítulo tem como objetivo apresentar o escalonador proposto para resolver a limitação, onde mesmo em um ambiente onde a memória disponível é um limitante, permite-se maximizar o reúso.

3.1 Algoritmo proposto

Como explicitado na Seção 2.6.1, algoritmos propostos para avaliar o reúso computacional a grão fino, em especial o *Reuse-Tree Merging Algorithm (RTMA)*, necessitam de um conhecimento do ambiente no qual a aplicação executará, pois a quantidade de estágios que podem ser unidos em um mesmo *bucket*, aumentando o reúso, depende diretamente da quantidade de memória disponível no mesmo. De forma que em ambientes cuja memória disponível é pouca em comparação com a memória necessária para execução das tarefas da aplicação, esses algoritmos se tornam pouco eficientes, pois conseguem fazer pouco, ou nenhum reuso.

O Algoritmo 1 mostra o funcionamento geral desse gerenciamento de tarefas, os algoritmos em si são relativamente simples, pois a elegância esta em usar o ciclo de vida de execução do *Region Templates* em seu funcionamento. Para o funcionamento em *multithread* existe todo um controle de semáforos e *mutex* que é detalhado no Apêndice A. Antes de começar o sistema de execução em si, é inserido a tarefa raiz da árvore na pilha *tasksBlock*. Após isso se inicia o ciclo de execução, enquanto tiver tarefa a ser executada fica em *loop* fazendo os seguintes passos. Primeiro, libera pega uma tarefa do topo da pilha de tarefas pronta para execução, para isso o primeiro libera as tarefas para execu-

ção a medida que se tem memória para isso, apos isso retira a tarefa do topo da pilha e libera para o *worker* executá-la. Para liberar tarefas que estão na pilha de *tasksBlock* para pilha *taskStack*, deixando assim a tarefa pronta para execução, para isso se verifica uma variável de controle , *memoryAvailable*, que indica quantas tarefas ainda pode ser colocada em execução, após transferir a tarefa entre as pilhas é decrementado a variável *memoryAvailable*. Seguindo, agora o sistema de execução em posse de uma tarefa a executa. Ao finalizar a execução da tarefa, percorre todas as tarefas dependentes e libera a dependência da tarefa que acabou de ser executada e,quando possível, insere as tarefas dependentes na lista de *tasksBlock* para serem executadas futuramente, finalmente libera o espaço que a tarefa estava ocupando em memoria para permitir a próxima execução, isso é sinalizado incrementando a variável *memoryAvailable*. Ao finalizar todas as tarefas esse ciclo de execução acaba.

Algorithm 1 Gerenciamento da execução de tarefas

Input: $taskTree$
Algoritmo acaba quando todas as tarefas forem executadas

```
1:  tasksBlock ← taskTree.head
2:  while taskBlock.size > 0 or taskStask.size > 0 do
3:      for all task ∈ tasksBlock do   ▷ Libera tarefa a medida que tem disponibilidade
4:          if memoryAvailable > 0 then
5:              taskStack ← task
6:              memoryAvailable − −
7:          else
8:              break
9:          end if
10:     end for
11:     if taskStack.size > 0 then                          ▷ Pega tarefa para execução
12:         task = taskStack.front
13:         taskQueue.pop_front
14:         task.run()                                      ▷ Manda a tarefa para execução
15:         for all dep ∈ task.dependents do
16:             dep.nDependecies − −
17:             if dep.nDependecies equal 0 then
18:                 tasksBlock ← tasks                      ▷ Insere no topo da pilha
19:             end if
20:             memoryAvailable + +
21:         end for
22:     end if
23: end while
```

Para o projeto do escalonador *Runtime Memory-Aware Scheduler for Reuse (RMSR) (Runtime Memory-Aware Scheduler for Reuse)* proposto nesse trabalho, há dois fatores que são necessários. É necessário limitar a largura de execução da árvore, desta forma

limitando a quantidade de tarefas que podem estar instanciadas em memória simultaneamente. Também é necessário executar essa árvore em profundidade. Esta árvore nada mais é que a representação das dependências entre tarefas, apos o processamento do algoritmo de reúso computacional, quando uma tarefa finaliza nem toda a memória usada por ela é liberada, parte das estruturas continuam em memória pois as tarefas dependentes precisam dessas estruturas como entrada. Então se não executarmos essa árvore em profundidade, pode chegar o caso onde boa parte da memória está sendo gasta para manter essas estruturas e não seja possível executar mais nenhuma tarefa completa sem ultrapassar o limite da memória.

A fim de simplificar o problema, foram feitas algumas premissas, são elas: o gasto máximo das tarefas de um *workflow* é conhecido e todas as tarefas têm o mesmo gasto de memória (considerando assim um ambiente homogêneo). Como sabemos a memória disponível, com essas duas suposições podemos calcular quantas tarefas, no máximo, podem ser instanciadas simultaneamente.

Assim, o primeiro requisito do escalonador consiste em verificar, no momento em que a tarefa tiver suas dependências resolvidas, se existe espaço em memória para instanciá-la. Caso não haja, a tarefa segue em uma lista de tarefas bloqueadas por memória, saindo deste local apenas quando uma outra tarefa é finalizada. Assim instanciamos um máximo de tarefas que caberá na memória, veja na Figura 3.1, teoricamente poderíamos instanciar até 7 tarefa simultaneamente, mas neste exemplo esta limitado a 4 tarefas então só essas 4 tarefas (em verde) estão executando.

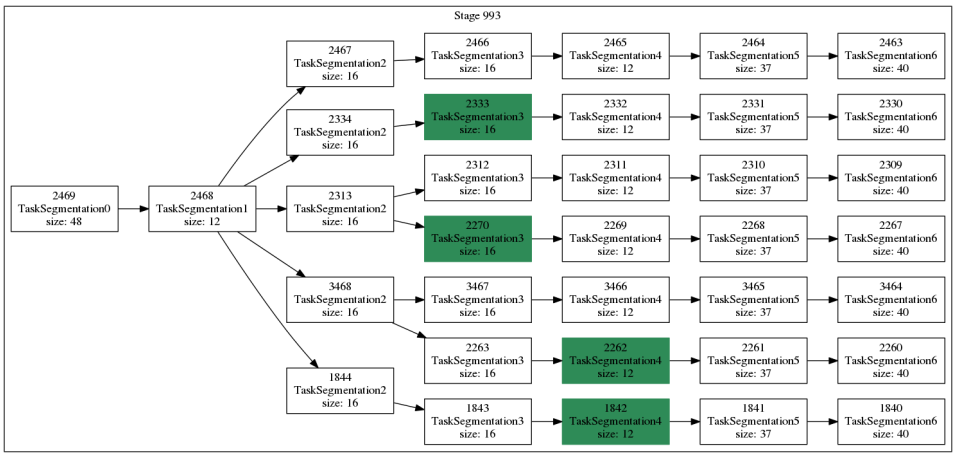

Figura 3.1: *Bucket* com 7 estágios unidos, porém com 4 tarefas sendo executadas concorrentemente (em verde)

Já o segundo requisito que é executar a árvore em profundidade. Dentro de um *worker* no *Region Templates* o sistema de execução das tarefas funciona da seguinte forma. Primeiro ele busca por uma tarefa que esteja pronta para ser executada, uma vez com

essa tarefa ele a executa. Ao finalizar essa tarefa é marcada como finalizada e p Region Template (RT) percorre todas as tarefas que são dependentes imediatas desta, liberando as mesmas desta dependência, se todas a dependências tiverem sido satisfeitas a tarefa dependente é inserida na lista de tarefas prontas para executar. Em seguida, busca outra tarefa na lista e esse ciclo se repete até que todas as tarefas tenham sido finalizadas.

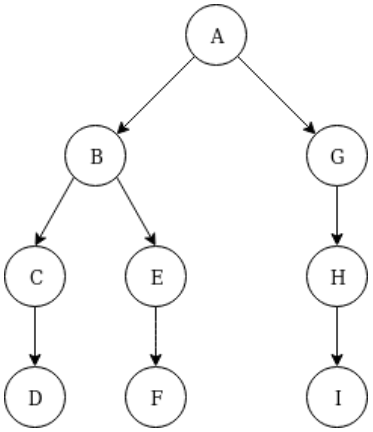

Figura 3.2: Árvore para exemplificar o funcionamento da execução em profundidade

Considerando esse ciclo, para garantir que a execução será feita em profundidade fazemos a lista de tarefas a serem executadas se comportar como uma pilha. Para a explicação detalhada desse funcionamento considere a árvore da Figura 3.2 como exemplo. No início tem uma tarefa na pilha de tarefas que podem ser executadas (a raiz da árvore ou nó A) então quando o sistema de execução solicitar uma tarefa, vai receber a tarefa A e, por consequência, retirar ela da pilha. Nesse momento a tarefa A estará sendo executada e a pilha estará vazia. No momento em que a tarefa A finalizar, ela resolverá as dependência de B e G e as colocará na pilha de execução, em alguma ordem, digamos $B \rightarrow G$, onde G é o topo da pilha. Agora, ao buscar uma tarefa para ser executada terá como retorno a tarefa G. Após executá-la colocará a tarefa H será empilhada, tendo a pilha a seguinte forma $B \rightarrow H$, continuando a execução de H, pois é o topo da pilha e finalizando a mesma, a pilha vai ficar com $B \rightarrow I$, desta forma a próxima a executar será a I, terminando a pilha ficará com B, pois a tarefa I não libera nenhuma outra tarefa para execução, ela é uma folha da árvore. Executando a tarefa B, serão liberadas as tarefas C e E, a pilha ficará com $C \rightarrow E$, por exemplo. E a execução se segue nesse ciclo até todas as tarefas serem executadas. A ordem de execução ficaria como $A, G, H, I, B, E, F, C, D$. Desta forma, percebe-se que a execução da árvore é em profundidade, executando o galho até sua folha antes de passar para o próximo galho.

O exemplo anterior considerou que apenas uma tarefa poderia ser executada por vez. Se considerarmos que podemos executar até duas tarefas, a ordem de execução ficaria algo

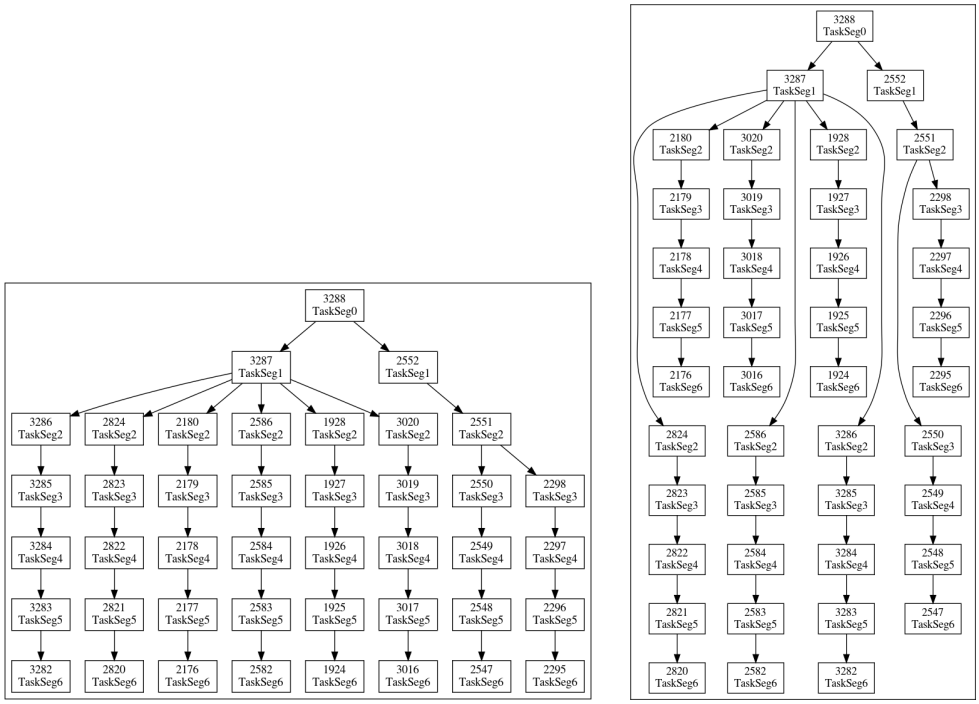

Figura 3.3: Exemplo de *workflow* que é submetido a um escalonamento em sua execução com o objetivo de não ultrapassar um limite de memória

como $A, B|G, E|H, F|I, C, D$, onde as tarefa separadas por $'|'$ estariam sendo executadas em paralelo, aqui considerando que as tarefas tempo de execução semelhantes. O importante é que desta maneira conseguimos aproveitar o paralelismo disponível, respeitamos as dependências entre as tarefas, não ultrapassamos a quantidade de tarefas a serem instanciadas simultaneamente, e executamos a árvore em profundidade. A Figura 3.3 mostra o funcionamento deste algoritmo, onde pegamos a árvore da Figura 3.3a e organizamos a ordem de execução em algo como mostrado na Figura 3.3b. Esse processo de afinamento da árvore é que nos permite executar *workflows* gerados pelo *RTMA* com um grande valor de *maxBucketSize* sem ultrapassar o limite da memória.

Capítulo 4

Resultados e Discussão

Esta seção trata dos resultados experimentais da otimização de reúso computacional proposta neste trabalho e integrada ao *Region Template framework*, explorando a comparação destas otimizações com as obtidas pelo algoritmo *RTMA* [2].

4.1 Ambiente de testes

O algoritmo aqui proposto foi avaliado usando um conjunto de imagens de tecido de câncer cerebral [17]. O *workflow* para análise das imagens consiste nos estágios de normalização, segmentação e comparação. O estágio de comparação calcula a diferença entre as máscaras geradas e o conjunto de máscara de referência criado usando a aplicação com os parâmetros padrões. Foi utilizado o aglomerado de computadores onde cada máquina é equipada com 2 CPUs Intel Haswell (E5-2695 v3), cada um com 14 cores e com 128GB de memória RAM DDR4 2133MHz.

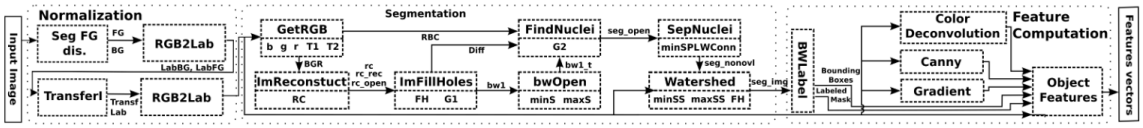

Figura 4.1: Workflow utilizado nos experimentos. (Fonte: [2])

4.2 Experimentos

Motivação

O primeiro experimento tem como objetivo demostrar o potencial de otimização pelo reuso de tarefas feito pelo *Reuse-Tree Merging Algorithm*, Na Figura 4.2 pode-se observar

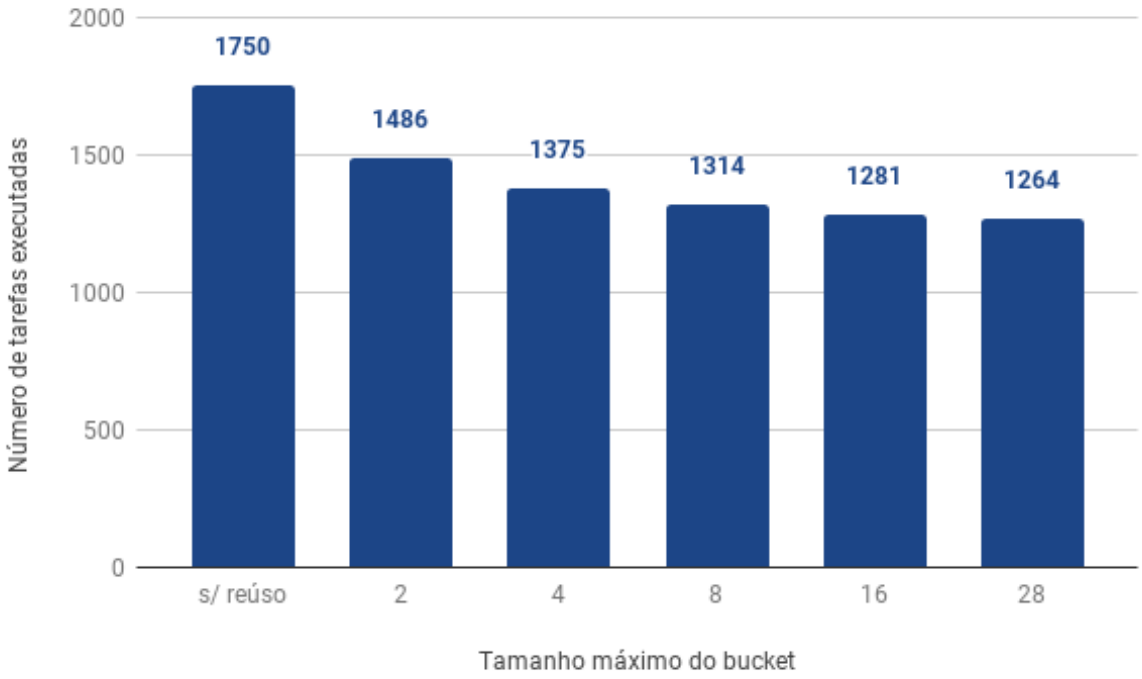

Figura 4.2: Quantidade de tarefas a serem executados, variando o tamanho máximo do *bucket*

os resultados da variação do tamanho máximo do *bucket*, onde chega-se a reutilizar mais 27% das tarefas da aplicação, esse reúso acarreta numa diminuição do tempo de execução, como mostrado na Figura 4.3. Isso acontece pois quando se aumenta o tamanho do *bucket* permite que mais reúso computacional seja feito, precisando assim executar menos tarefas. Este resultado serve de motivação para este trabalho. Com aumento do *bucket* diminui-se o tempo de execução porem acarreta em um aumento do gasto da memória. O objetivo deste trabalho foi manter os ganhos expressivos como os obtidos pelo *RTMA* [2], sem a necessidade de aumentar o uso de memória.

Resultados do *RSMR*

O experimento feito no *RMSR* consiste em comparar a execução de uma aplicação nele com a execução da mesma aplicação do *RTMA* (Figura 4.4). Para isso foi medido o tempo de execução para diversos cenários, a fim de simular os ganhos de cada algoritmos em diferentes ambientes. Foi feita a variação do "tamanho máximo o *bucket*", a proposta é medir em um ambiente onde o *RTMA* conseguiria executar com *bucket* no máximo esse valor, por conta da memória disponível, como o *RMSR* poderia permitir maiores ganhos,

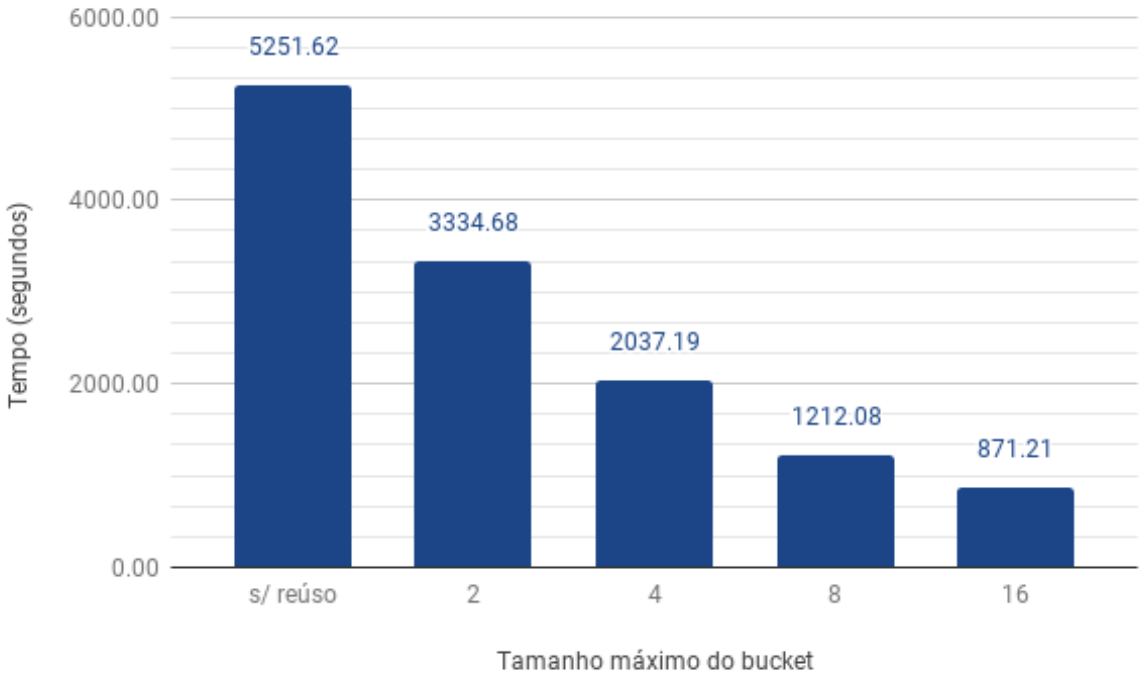

Figura 4.3: Tempo de execução pelo tamanho máximo do *bucket* no *RTMA*

gastando a mesma memória gasta pelo *RTMA*. Em todos os casos medido o valor de *bucket* utilizado pelo *RMSR* foi de 16.

É possível observar que nos casos onde o ambiente é mais limitado, os ganhos do *RMSR* se tornam mais expressivos, no caso mais critico onde o *RTMA* não conseguiria fazer nenhum reuso o o *RMSR* executou a mesma aplicação, gastando a mesma quantidade de memória, 63% mais rápido. Já quando a memória não é mais um limitante e ambos os casos está sendo feito a mesma quantidade de reúso, não há nenhum ganho por parte do *RMSR*. Importante frisar também, que nesses casos não há nenhuma perda considerável do algoritmo deste trabalho, mostrando assim que não foi inserido um *overhead* considerável, pois o que esse trabalho fez foi substituir o escalonador que estava sendo utilizado pelo *RT* por um outro escalonador.

Foi executado também o *RMSR* com $bucket = 28$, para cada caso do experimento anterior e os valores de tempo são $1864.8, 1191.2, 850.1, 736.7, 697.9$ segundos, respectivamente. No experimento onde o *RTMA* puro executara com $bucket = 28$ foi ultrapassado o limite de memória no ambiente, não sendo possível medir o tempo de execução neste caso.

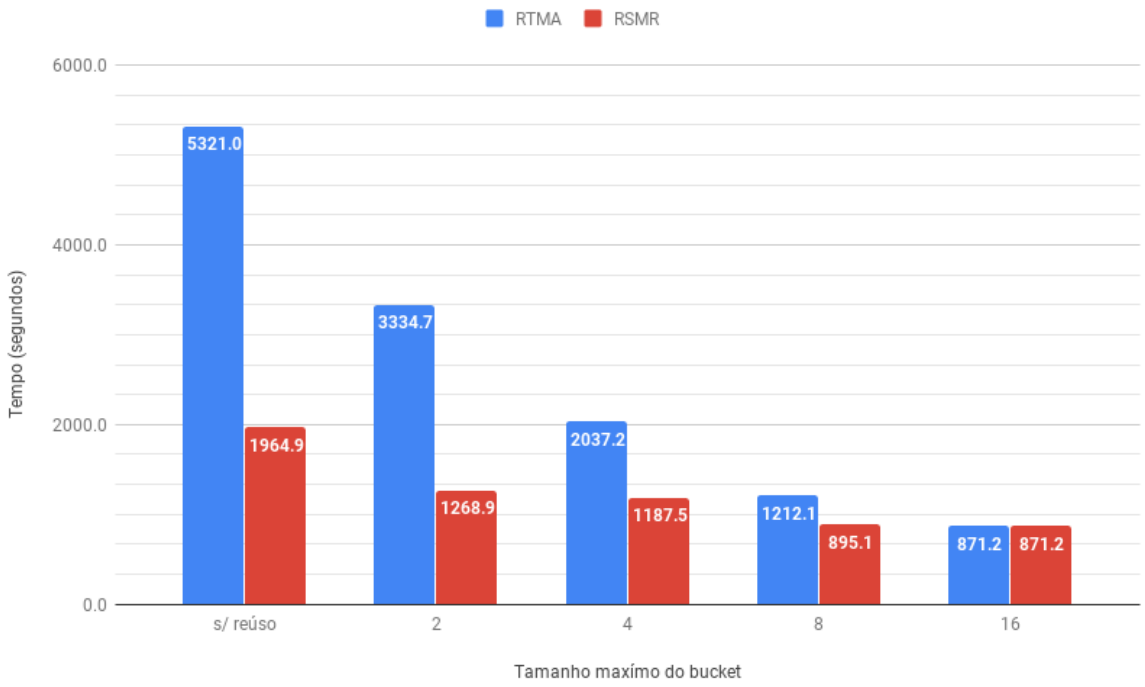

Figura 4.4: Tempo de execução comparando o *RTMA* com *RMSR* simulando diversos ambientes

Escalabilidade

Por fim, para finalizar essa seção, foram feitos os experimentos com o objetivo de mostrar que escalabilidade da aplicação não foi afetada pelo algoritmo aqui proposto. Estes experimentos foram feitos com uma aplicação com 1750 tarefas, divididas igualmente em 250 estágios. Primeiro a escalabilidade aumentando o número de maquinas, na Figura 4.5 pode-se ver o resultado desse experimento, ao calcular o *seepdup* desse experimento, ou seja $T_{serial}/T_{paralelo}$. Encontramos 23.56 para 28 máquinas, relativamente próximo do teórico, não chega a ele um dos motivos é que como explicado na Seção 2.4.2 uma máquina fica responsável pelo *manager*, assim não temos paralelismo com todas as máquinas.

Agora, para mostrar a escalabilidade entre número de *cores* no mesmo *worker* foi feito o experimento a seguir. Neste experimento, a fim de isolar a variável de estudo, foi utilizado apenas um *worker* e esse poderia requisitar mais de um estágio por vez, desta forma garante que todos os núcleos da maquina estão sendo utilizados. Na Figura 4.6 é mostrado o *speedup* desse experimento, novamente ficando próximo do teórico, chegando a 24.02 com 28 *cores*.

Com isso foi possível mostrar, nessa seção, o potencial de otimização do *RTMA*, os ganhos que este trabalho obtém, principalmente e ambientes com memória limitada, e

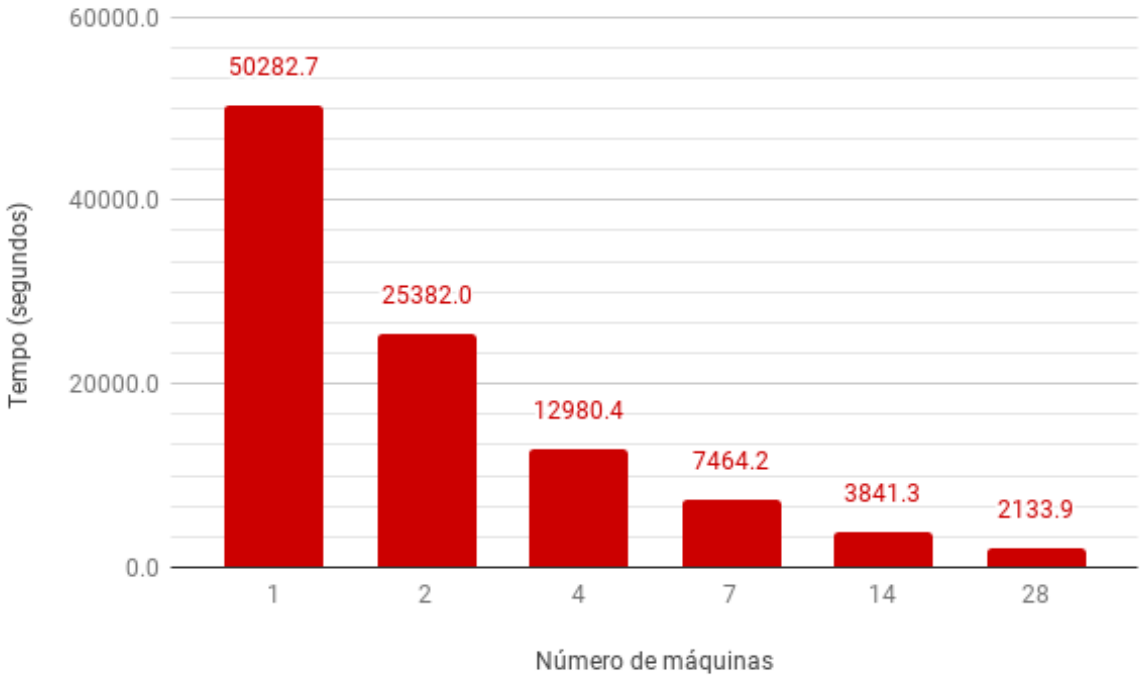

Figura 4.5: Tempo de execução, pela variação do número de máquinas

por fim a escalabilidade da aplicação.

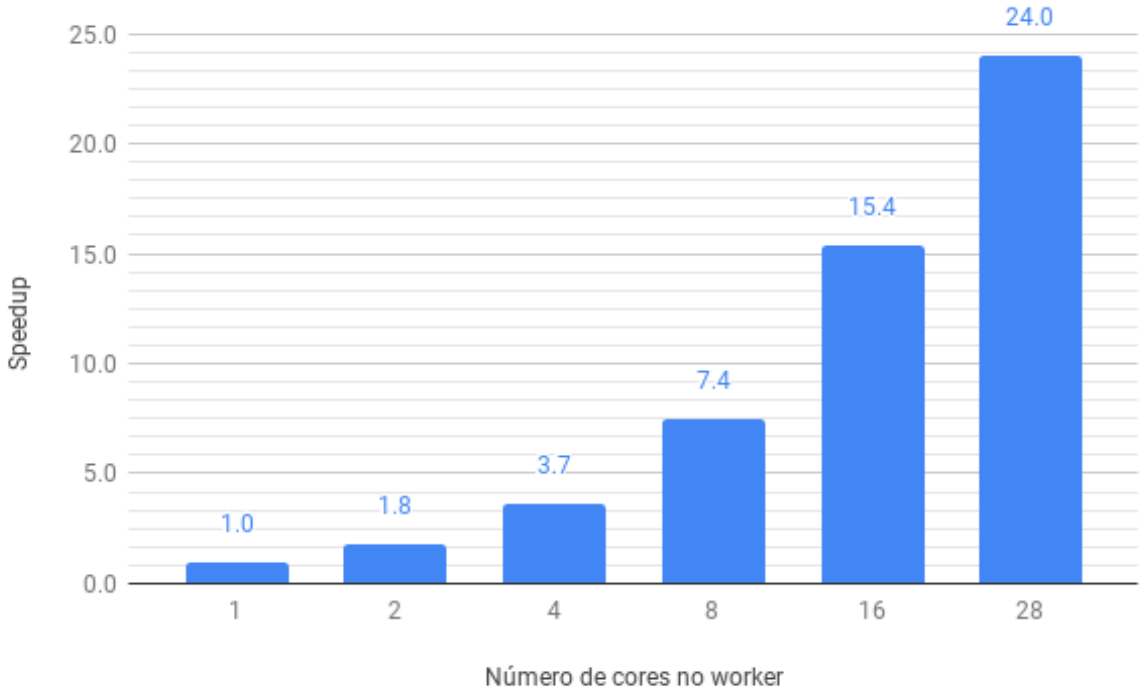

Figura 4.6: *Speedup* para a quantidade de *cores* de um *worker*

Capítulo 5

Conclusão

Este trabalho atuou a partir de trabalhos anteriores de reúso computacional visando estender a visibilidade e o alcance de tais soluções. Para isso, foi proposto um algoritmo que de forma *online* gerencia a execução das tarefas de modo que seja possível a execução do *bucket* independentemente da quantidade de estágios unidos nele. Desta forma os algoritmos propostos por trabalhos anteriores, conseguem se aproveitar mais das possibilidades de reúso mesmo em ambientes onde os recursos de memória são limitados.

Os experimentos abordados neste trabalho não são tão abrangentes. Isso se deve à dificuldade de executar testes que explorassem as limitações de memória para aplicações muito dispendiosas. Conforme o escopo do teste crescia também cresciam os tempos para executar cada teste e os requisitos dos ambientes em que seriam executados.

Todavia, os experimentos aqui feitos foram suficiente para indicar algumas características interessantes do trabalho aqui proposto:

1. A proposta traz consigo um desacoplamento do algoritmo que executará o reúso, ao não trabalhar diretamente na estrutura de dados que o algoritmo do reúso cria, o algoritmo aqui proposto consegue ser válido para uma gama de métodos de otimização, preocupando-se apenas em executar as tarefas enviadas no ambiente disponível, não se preocupando em como essas tarefas estão representadas e tampouco a forma que esses reúsos foram calculados.

2. O gerenciamento das tarefas da forma aqui proposto se mostrou eficaz para fazer que algoritmos como o *RTMA* consigam ter o seu melhor desempenho, com uma certo nível de independência do ambiente que será executado.

3. Em execuções onde o ambiente já permitia o melhor ganho possível ao algoritmo de reúso, o gerenciador se comporta como apenas um *dispatcher* simples, não gerando um *overhead* significativo no tempo de execução da aplicação.

4. A proposta tem um maior valor quando o ambiente disponível possui pouca memória.

Assim, se cria mais um parâmetro para customizar a execução, de forma que em ambientes onde haja paralelismo a nível de estágios esse parâmetro pode gerenciar o *trade-off* do ganho pelo reúso em detrimento do paralelismo a vista do tempo de execução. Isto se torna bastante eficaz para personalizar a execução a depender da aplicação e do ambiente de cada caso, de forma a conseguir os melhores resultados. Este trabalho também faz com que algoritmos como o *RTMA* sejam úteis em ambientes limitados, onde antes os mesmos não conseguiam executar com um maior nível de reúso.

5.1 Trabalhos futuros

Como trabalho futuro é importante um estudo quantitativo do gasto de memória de cada aplicação que utiliza o *Region Templates* para uma correta utilização deste trabalho e uma generalização do algoritmo aqui proposto para tarefas heterogêneas onde o gasto de memória de cada tarefa do *workflow* é significativamente diferente.

Referências

[1] *Openslide library*. http://openslide.org/. ix, 5

[2] Barreiros, W., G. Teodoro, T. Kurc, J. Kong, A. C. M. A. Melo e J. Saltz: *Parallel and efficient sensitivity analysis of microscopy image segmentation workflows in hybrid systems*. Em *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, páginas 25–35, Sept 2017. ix, x, 2, 6, 9, 13, 14, 15, 16, 17, 19, 24, 25

[3] Teodoro, George, Tahsin M. Kurç, Guilherme Andrade, Jun Kong, Renato Ferreira e Joel H. Saltz: *Application performance analysis and efficient execution on systems with multi-core cpus, gpus and mics: a case study with microscopy image analysis*. IJHPCA, 31(1):32–51, 2017. ix, 6

[4] Kothari, Sonal, John H Phan, Todd H Stokes e May D Wang: *Pathology imaging informatics for quantitative analysis of whole-slide images*. Journal of the American Medical Informatics Association, 20(6):1099–1108, 2013. ix, 7, 9, 10

[5] Gurcan, Metin N., Laura Boucheron, Ali Can, Anant Madabhushi, Nasir Rajpoot e Bulent Yener: *Histopathological image analysis: A review*. IEEE Rev Biomed Eng, 2:147–171, 2009, ISSN 1937-3333. http://www.ncbi.nlm.nih.gov/pmc/articles/PMC2910932/, 20671804[pmid]. ix, 1, 5, 6, 8

[6] Teodoro, George, Tony Pan, Tahsin Kurc, Jun Kong, Lee Cooper, Scott Klasky e Joel Saltz: *Region templates: Data representation and management for high-throughput image analysis*. Parallel Computing, 40(10):589 – 610, 2014, ISSN 0167-8191. ix, 1, 2, 9, 12, 13, 14

[7] Teodoro, George, Tahsin M. Kurç, 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, 33(7):1064–1072, 2017. xi, 1, 8, 10, 13

[8] Veta, Mitko, Paul J. van Diest, Robert Kornegoor, André Huisman, Max A. Viergever e Josien P. W. Pluim: *Automatic nuclei segmentation in h&e stained breast cancer histopathology images*. PLOS ONE, 8(7), julho 2013. https://doi.org/10.1371/journal.pone.0070221. 1, 2, 5

[9] Cooper, Lee A. D., Jun Kong, David A. Gutman, Fusheng Wang, Jingjing Gao, Christina Appin, Sharath Cholleti, Tony Pan, Ashish Sharma, Lisa Scarpace, Tom Mikkelsen, Tahsin Kurc, Carlos S. Moreno, Daniel J. Brat e Joel H. Saltz: *Integrated morphologic analysis for the identification and characterization of disease subtypes*.

J Am Med Inform Assoc, 19(2):317–323, Jan 2012, ISSN 1067-5027. http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3277636/, amiajnl-2011-000700[PII]. 1, 5

[10] Di Cataldo, Santa, Elisa Ficarra, Andrea Acquaviva e E Macii: *Automated segmentation of tissue images for computerized ihc analysis*. 100:1–15, março 2010. 1

[11] Coelho, Luís Pedro, Aabid Shariff e Robert F. Murphy: *Nuclear segmentation in microscope cell images: A hand-segmented dataset and comparison of algorithms*. Proc IEEE Int Symp Biomed Imaging, 5.193098E6:518–521, 2009, ISSN 1945-7928. http://www.ncbi.nlm.nih.gov/pmc/articles/PMC2901896/, 20628545[pmid]. 1

[12] Gomes, Jeremias M., Alba Cristina Magalhaes Alves de Melo, Jun Kong, Tahsin M. Kurç, Joel H. Saltz e George Teodoro: *Cooperative and out-of-core execution of the irregular wavefront propagation pattern on hybrid machines with Intel® Xeon Phi™*. Concurrency and Computation: Practice and Experience, 30(14), 2018. https://doi.org/10.1002/cpe.4425. 1

[13] Kurç, Tahsin M., Xin Qi, Daihou Wang, Fusheng Wang, George Teodoro, Lee A. D. Cooper, Michael Nalisnik, Zhi-Yang Li, Joel H. Saltz e David J. Foran: *Scalable analysis of big pathology image data cohorts using efficient methods and high-performance computing strategies*. BMC Bioinformatics, 16:399:1–399:21, 2015. https://doi.org/10.1186/s12859-015-0831-6. 1

[14] Teodoro, George, Tahsin M. Kurç, Jun Kong, Lee A. D. Cooper e Joel H. Saltz: *Comparative Performance Analysis of Intel (R) Xeon Phi (TM), GPU, and CPU: A Case Study from Microscopy Image Analysis*. Em *2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19-23, 2014*, páginas 1063–1072, 2014. 1

[15] Teodoro, George, Rafael Sachetto Oliveira, Olcay Sertel, Metin N. Gurcan, Wagner Meira Jr., Ümit V. Çatalyürek e Renato Ferreira: *Coordinating the use of GPU and CPU for improving performance of compute intensive applications*. Em *Proceedings of the 2009 IEEE International Conference on Cluster Computing, August 31 - September 4, 2009, New Orleans, Louisiana, USA*, páginas 1–10, 2009. 1

[16] Saltelli, Andrea, Karen Chan, E Marian Scott *et al.*: *Sensitivity analysis*, volume 1. Wiley New York, 2000. 1

[17] Kong, Jun, Lee A. D. Cooper, Fusheng Wang, Jingjing Gao, George Teodoro, Lisa Scarpace, Tom Mikkelsen, Matthew J. Schniederjan, 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, 8(11), novembro 2013. https://doi.org/10.1371/journal.pone.0081049. 1, 6, 24

[18] Weirs, V. Gregory, James R. Kamm, Laura P. Swiler, Stefano Tarantola, Marco Ratto, Brian M. Adams, William J. Rider e Michael S. Eldred: *Sensitivity analysis techniques applied to a system of hyperbolic conservation laws*. Reliability Engineering & System Safety, 107:157 – 170, 2012, ISSN 0951-8320. http://www.

sciencedirect.com/science/article/pii/S0951832011002717, SAMO 2010. 2, 8

[19] Sodani, A. e G. S. Sohi: *Understanding the differences between value prediction and instruction reuse*. Em *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture*, páginas 205–215, Dec 1998. 2

[20] Sodani, Avinash e Gurindar S. Sohi: *An empirical analysis of instruction repetition*. SIGPLAN Not., 33(11):35–45, outubro 1998, ISSN 0362-1340. http://doi.acm.org/10.1145/291006.291016. 2

[21] Cornish, Toby C., Ryan E. Swapp e Keith J. Kaplan: *Whole-slide imaging: Routine pathologic diagnosis*. Advances in Anatomic Pathology, 19(3), 2012, ISSN 1072-4109. https://journals.lww.com/anatomicpathology/Fulltext/2012/05000/Whole_slide_Imaging___Routine_Pathologic_Diagnosis.3.aspx. 4

[22] Wienert, Stephan, Michael Beil, Kai Saeger, Peter Hufnagl e Thomas Schrader: *Integration and acceleration of virtual microscopy as the key to successful implementation into the routine diagnostic process*. Diagnostic Pathology, 4:3 – 3, 2009. 4

[23] Ghaznavi, Farzad, Andrew Evans, Anant Madabhushi e Michael Feldman: *Digital imaging in pathology: Whole-slide imaging and beyond*. Annual Review of Pathology: Mechanisms of Disease, 8(1):331–359, 2013. https://doi.org/10.1146/annurev-pathol-011811-120902, PMID: 23157334. 4

[24] Rojo, Marcial García, Gloria Bueno García, Carlos Peces Mateos, Jesús González García e Manuel Carbajo Vicente: *Critical comparison of 31 commercially available digital slide systems in pathology*. International journal of surgical pathology, 14 4:285–305, 2006. 5

[25] Pantanowitz, Liron, John H. Sinard, Walter H. Henricks, Lisa A. Fatheree, Alexis B. Carter, Lydia Contis, Bruce A. Beckwith, Andrew J. Evans, Avtar Lal e Anil V. Parwani: *Validating whole slide imaging for diagnostic purposes in pathology: Guideline from the college of american pathologists pathology and laboratory quality center*. Archives of Pathology & Laboratory Medicine, 137(12):1710–1722, 2013. https://doi.org/10.5858/arpa.2013-0093-CP, PMID: 23634907. 5

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

[27] Filippi-Chiela, Eduardo C., Manuel M. Oliveira, Bruno Jurkovski, Sidia Maria Callegari-Jacques, Vinicius Duval da Silva e Guido Lenz: *Nuclear morphometric analysis (nma): Screening of senescence, apoptosis and nuclear irregularities*. PLOS ONE, 7(8):1–10, agosto 2012. https://doi.org/10.1371/journal.pone.0042522. 5

[28] Gurcan, M. N., T. Pan, H. Shimada e J. Saltz: *Image analysis for neuroblastoma classification: Segmentation of cell nuclei*. Em *2006 International Conference of the IEEE Engineering in Medicine and Biology Society*, páginas 4844–4847, Aug 2006. 5

[29] Teodoro, George, Tony Pan, Tahsin M. Kurç, Jun Kong, Lee A. D. Cooper, Norbert Podhorszki, Scott Klasky e Joel H. Saltz: *High-throughput analysis of large microscopy image datasets on CPU-GPU cluster platforms*. Em *27th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2013, Cambridge, MA, USA, May 20-24, 2013*, páginas 103–114, 2013. 6

[30] Sonka M., Hlavac V., Boyle R.: *Image pre-processing In: Image Processing, Analysis and Machine Vision.* Boston, MA, 1993. página 56–111. 6

[31] Andrea Saltelli, Marco Ratto, Terry Andres Francesca Campolongo Jessica Cariboni Debora Gatelli Michaela Saisana Stefano Tarantola: *Global Sensitivity Analysis: The Primer.* Wiley, 2008. 7

[32] Andrea Saltelli, Stefano Tarantola, Francesca Campolongo Marco Ratto: *Sensitivity Analysis in Practice: A Guide to Assessing Scientific Models.* Wiley, 2004. 7, 8

[33] Morris, Max D.: *Factorial sampling plans for preliminary computational experiments.* 33(2):161–174, 1991, ISSN 00401706. `http://www.jstor.org/stable/1269043`. 8

[34] Iooss, Bertrand e Paul Lemaître: *A Review on Global Sensitivity Analysis Methods*, páginas 101–122. Springer US, Boston, MA, 2015, ISBN 978-1-4899-7547-8. `https://doi.org/10.1007/978-1-4899-7547-8_5`. 8

[35] Liu, Qiang, Konstantinos Masselos e George A Constantinides: *Data reuse exploration for fpga based platforms applied to the full search motion estimation algorithm.* Em *Field Programmable Logic and Applications, 2006. FPL'06. International Conference on*, páginas 1–6. IEEE, 2006. 14

Apêndice A

Detalhes de implementação no *Region Templates*

Os métodos propostos neste trabalho não estão restritos ao caso de uso em questão. Estes algoritmos servem para otimizar a execução de uma análise de sensibilidade de diversas aplicações, não somente a da apresentada na Seção 2.2. Generalizando ainda mais, esses algoritmos servem para otimizar qualquer aplicação em que seja possível uma execução distribuída, e além disso que possa ser representada como tarefas com granularidade em dois níveis e que tenham algum tipo de repetição nas execuções dos *workflows* de granularidade fina. Neste contexto uma das contribuições deste trabalho é integrar o que foi aqui proposto ao *Region Templates Framework*, e a aplicação apresentada é tipicamente usada neste *framework*.

O *Region Templates Framework* foi desenvolvido em C++ e se utiliza de orientação a objeto, de forma que é relativamente simples a criação de uma nova estrutura, no caso o escalonador. Aqui, por uma questão de compatibilidade com o que já existia, essa estrutura receberá o nome de *TaskQueue*, isto é, uma fila de tarefas. Na Figura A.1 é mostrada a classe do escalonador *TaskQueueMB*, herda de *TaskQueue* que é justamente a estrutura que faz esse gerenciamento do *RTF*. Nessa classe há duas listas: uma que enumera as tarefas que estão efetivamente liberadas para serem executadas *tasksQueue*, a qual é consumida por uma *ThreadPool* do *RTF*; e uma lista para as tarefas bloqueadas por falta de memória (*memBlockQueue*), conforme explicado anteriormente. Além disso, há ainda uma variável inteira (*available*), que faz o controle de quantas tarefas ainda podem ser colocadas para execução.

O escalonador efetivamente está dividido em quatro partes. A primeira é responsável pela inserção de novas tarefas (Figura A.2), que acontece quando as dependências de uma certa tarefa já foram resolvidas e ela está livre para poder ser executada; assim, as novas tarefas são diretamente adicionadas à lista *memBlockQueue*. Se houver memória

```
class TasksQueueMB: public TasksQueue {
private:
        list<Task*> tasksQueue;
        list<Task*> memBlockQueue;

        int available;

public:
        TasksQueueMB(int cpuThreads, int gpuThreads, int available){
                this->cpuThreads = cpuThreads;
                this->gpuThreads = gpuThreads;

                this->available = available;
        }
        bool insertTask(Task* task);
        Task* getTask(int procType=ExecEngineConstants::CPU);
        int getSize();
        Task* getByTaskId(int id);

        void retrieveResources(int memory);
        void unblockTasks();

};
```

Figura A.1: Classe em $C++$ que representa o escalonador aqui proposto

suficiente ($available > 0$) , a tarefa logo será liberada pela segunda rotina *unblockTasks* (Figura A.4), cuja função é exatamente esta. Outra parte que compõe o escalonador é a *getTask* (Figura A.5), chamada pela *ThreadPool*, e que entrega tarefas para execução; além disso, convoca a *unblockTasks* para liberar possíveis tarefas que possam ser executadas. Por fim, a rotina *retrieveResources* (Figura A.3) que incrementa o *available*, serve para o controle de que, quando uma tarefa for finalizada, o recurso que estava sendo gasto com ela se torne disponível.

```
1  bool TasksQueueMB::insertTask(Task *task)
2  {
3          pthread_mutex_lock(&queueLock);
4
5          this->memBlockQueue.push_front(task);
6
7          pthread_mutex_unlock(&queueLock);
8          sem_post(&tasksToBeProcessed);
9          return true;
10
11 }
```

Figura A.2: Método em $C++$ cuja função é inserir tarefas que podem ser executadas para gerência do escalonador

```
1  void TasksQueueMB::retrieveResources(int memory)
2  {
3          pthread_mutex_lock(&queueLock);
4          this->available += memory;
5          pthread_mutex_unlock(&queueLock);
6          unblockTasks();
7  }
```

Figura A.3: Método em $C++$ disponibilizado para informar ao escalonador que recursos foram disponibilizados após o fim de uma tarefa

```
void TasksQueueMB::unblockTasks() {
        //Para cada elemento da lista de block
        //se puder se jogado para execução va
        pthread_mutex_lock(&queueLock);
        for (list<Task*>::iterator it = memBlockQueue.begin(); it !=
                                        memBlockQueue.end(); ++it ) {
                if(this->available > 0) {
                        this->tasksQueue.push_back((*it));
                        this->available -= 1;

                        this->memBlockQueue.erase(it--);
                } else {
                        break;
                }
        }
        pthread_mutex_unlock(&queueLock);

}
```

Figura A.4: Rotina interna do escalonador responsável por liberar tarefas cujo a qual podem ser executadas de acordo com a política de gerenciamento

```
Task* TasksQueueMB::getTask(int procType)
{
        Task *retTask = NULL;

        sem_wait(&tasksToBeProcessed);
        unblockTasks();
        pthread_mutex_lock(&queueLock);
        if(tasksQueue.size() > 0){
                retTask = tasksQueue.front();

                if(retTask != NULL){
                        tasksQueue.pop_front();
                }
        }
        pthread_mutex_unlock(&queueLock);
        return retTask;

}
```

Figura A.5: Método em $C++$ disponível para que o sistema de execução da *threadPool* possa pedir novas tarefas para executar