UNIVERSIDADE DO VALE DO RIO DOS SINOS
UNIDADE ACADÊMICA DE GRADUAÇÃO
CURSO DE BACHARELADO EM SISTEMAS DE INFORMAÇÃO

ANDERSON CUNHA KRANEN

PARALELIZAÇÃO DO ALGORITMO PARAMETRIZAÇÕES PROGRESSIVAS PARA ARQUITETURAS *MULTICORE*

São Leopoldo
2020

Anderson Cunha Kranen

PARALELIZAÇÃO DO ALGORITMO PARAMETRIZAÇÕES PROGRESSIVAS PARA ARQUITETURAS *MULTICORE*

Artigo apresentado como requisito parcial para obtenção do título de Bacharel em Sistemas de Informação, pelo Curso de Sistemas de Informação da Universidade do Vale do Rio dos Sinos (UNISINOS)

Orientador(a): Profa. Dra. Andriele Busatto do Carmo

São Leopoldo
2020

PARALELIZAÇÃO DO ALGORITMO PARAMETRIZAÇÕES PROGRESSIVAS PARA ARQUITETURAS *MULTICORE*

Anderson Cunha Kranen[1]
Andriele Busatto do Carmo[2]

Resumo: Plataformas com processadores *multicore* se tornaram comuns há alguns anos, difundindo assim a computação paralela. Com a popularização das GPUs, se tornou comum encontrar computadores com uma placa gráfica, permitindo processamento heterogêneo quando demandado. Utilizando métodos de comunicação e sincronização entre os processadores da plataforma heterogênea, é possível atingir ganhos de desempenho antes viáveis somente em *clusters* e *grids*. Cada processador dentro da plataforma heterogênea se comporta de uma maneira, portanto, o balanceamento entre os diferentes tipos de cargas de processamento é essencial. Um problema que pode ser resolvido com esse tipo de computação é a parametrização de malhas triangulares. O processo de parametrização visa criar uma bijeção entre duas superfícies com uma equivalência de pontos para resolver diversos problemas da computação gráfica. O mais recente algoritmo de parametrização é o algoritmo de parametrizações progressivas. Esse algoritmo tem como princípio a rápida convergência para o resultado ideal da parametrização. Esse trabalho visa explorar o potencial paralelo do algoritmo em plataformas heterogêneas utilizando CPU e GPU. A implementação do algoritmo em paralelo na CPU foi realizada utilizando Intel TBB e na GPU as bibliotecas cuSolver, ArrayFire, MAGMA, ViennaCL e CUSP foram utilizadas. Com os testes realizados, verificou-se que o comportamento do algoritmo em GPU não é eficiente com as soluções existentes. Uma nova proposta com otimização do paralelismo em CPU foi então aplicada e quando configurada para 16 *threads* atingiu um *speedup* de até 6 vezes em algumas etapas do algoritmo em comparação com o algoritmo sequencial.

Palavras-chave: Computação Paralela. Arquiteturas Heterogêneas. Paralelização de Algoritmos de Parametrização. Parametrização de Malhas.

Abstract: Platforms with multicore processors have become common in the past few years, disseminating parallel computing. With the popularization of GPUs, it has become increasingly common to find computers with a graphic card, enabling heterogeneous processing when demanded. By using methods of communication and synchronization between the processors of the heterogeneous platform, it is possible to achieve performance gains previously feasible only in clusters and grids. Each processor within the heterogeneous platform behaves in a way, therefore, balancing between different types of processing loads is essential. One problem that can be solved with this type of computation is the parameterization of triangular meshes. The parameterization process aims to create a bijection between two surfaces with an equivalence of points to solve several problems of the computer graphics. The most recent parameterization algorithm is the algorithm of progressive parameterization. This algorithm has as a principle the rapid convergence for the ideal parameterization result. This work aims to explore the parallel potential of the algorithm in heterogeneous platforms using CPU and GPU. The implementa-

[1]Graduando em Sistemas de Informação pela Unisinos. Email: ackranen@edu.unisinos.br

[2]Doutora em Ciência da Computação pela Pontifícia Universidade Católica do Rio Grande do Sul - PUCRS (2017) na área de Processamento Paralelo e Distribuído (PPD). Mestre em Ciência da Computação (2009) também pela PUCRS na área de PPD. Bacharel em Ciência da Computação pela Universidade de Passo Fundo - UPF (2006) com trabalho de conclusão na mesma área de pesquisa do mestrado e doutorado. Email: acarmo@unisinos.br

tion of the parallel algorithm in the CPU was performed using Intel TBB and on the GPU using the libraries cuSolver, ArrayFire, MAGMA, ViennaCL and CUSP. With the tests performed, it was verified the behavior of the algorithm when running on the GPU is not efficient with the existing solutions. A new proposal with parallel optimization on the CPU was used and when configured to run with 16 threads achieved a speedup of 6 in some steps of the algorithm when compared with the sequential version.

Keywords: Parallel Computing. Heterogeneous Architectures. Parallelization of Parameterization Algorithms. Mesh Parameterization.

1 INTRODUÇÃO

Por vários anos, aumentar a frequência dos processadores foi uma técnica aplicada pela indústria quando o objetivo era aumentar o desempenho. No entanto, com as limitações impostas pelos processos de fabricação da CPU (*Central Processing Unit*) e com o consumo energético que esse aumento agregava, surgiram novas linhas de pesquisa para tornar a computação paralela também viável em plataformas de memória compartilhada (SANDERS; KANDROT, 2010).

A comunidade de computação de alto desempenho vem desenvolvendo programas paralelos por décadas. Esses programas são escritos geralmente para serem executados em computadores de larga escala, como os *clusters* e supercomputadores, por exemplo. Porém, somente algumas aplicações justificam o investimento necessário para construir e manter esse tipo de plataforma (KIRK; HWU, 2010).

Para manter o avanço em termos de poder de processamento e reduzir o consumo energético, fabricantes começaram a desenvolver processadores com várias unidades computacionais no mesmo chip, que eram utilizadas de maneira totalmente independente e que tinham acesso à mesma memória de maneira concorrente. Essa arquitetura foi chamada de *multicore*. A computação paralela para esse tipo de arquitetura foi se popularizando, tornando cada computador um pequeno computador paralelo (RAUBER; RÜNGER, 2010).

Em termos de *software*, desenvolvedores não puderam assumir que uma atualização de *hardware* causaria um aumento de desempenho de forma automática em seus *softwares*. Esforços se tornaram necessários para a reestruturação do programa para utilizar adequadamente a arquitetura paralela. O código que antes era desenvolvido com base na arquitetura de Von Neumann, onde somente uma instrução seria processada por vez, teve de ser atualizado para suportar mais do que um único fluxo de processamento (*threads*) (RAUBER; RÜNGER, 2010; KIRK; HWU, 2010).

A paralelização de algoritmos requer que várias decisões sejam tomadas no momento de seu design. O primeiro ponto a ser considerado é como o programa vai ser particionado. O nível de particionamento é o que vai determinar o tamanho da carga de processamento alocada para os processadores. O particionamento de grão grosso atribui cargas maiores para cada processador enquanto o particionamento de grão fino atribui tarefas pequenas para cada processador. O

segundo ponto importante é como o programa vai realizar a sincronização entre as unidades de processamento, definindo assim como se dará a integridade e validade dos dados em memória durante a execução (GEBALI, 2011).

Com o lançamento de aceleradoras gráficas programáveis na década de 90, era só uma questão de tempo até que a programação para GPUs (*Graphics Processing Unit*) se tornasse popular como nas CPUs. A partir da necessidade de gráficos cada vez mais realistas e de alta qualidade, novas funcionalidades foram surgindo e as APIs (*Application Programming Interface*) foram provendo cada vez mais controles para os programadores. Em 2001 foi lançada a primeira GPU com *pipeline* programável, possibilitando a programação dos chamados *shaders* e entregando maior controle das computações que ocorrem na GPU (SANDERS; KANDROT, 2010).

Os *shaders* foram um grande avanço para as GPUs e chamaram a atenção de muitos pesquisadores do ramo da computação, pois abria a possibilidade de explorar o potencial paralelo das GPUs. Estudos foram realizados para a utilização das APIs gráficas na computação de problemas não relacionados com gráficos. Para realizar essas computações, todos os valores numéricos eram transformados em canais de cor para que a GPU fosse enganada a realizar um processamento não gráfico. Os resultados desses estudos foram positivos devido ao alto poder de processamento das aceleradoras gráficas e demonstrou que existia uma demanda para o uso de GPUs em outras áreas da computação. Com o surgimento das APIs de computação para GPUs, foi possível obter aumentos de desempenho na ordem de centenas de vezes quando comparado ao código implementado para execução na CPU (SANDERS; KANDROT, 2010; KIRK; HWU, 2010).

Considerando que hoje em dia grande parte dos computadores possuem uma GPU em sua arquitetura, o acesso às plataformas de computação heterogênea se tornou mais comum. Por isso, tornou-se viável otimizar algoritmos que antes rodavam no seu pico de desempenho na CPU (KAUER; SIQUEIRA, 2013).

Um exemplo de algoritmo de alto custo computacional é o processo de parametrização de malhas triangulares. O algoritmo consiste em criar um mapeamento entre uma superfície 3D e outra superfície similar para diversos tipos de aplicações. Esse processo é fundamental para a computação gráfica, pois processos de mapeamento de textura, deformação de superfície, e animações por exemplo, necessitam de um mapeamento de alta qualidade entre as superfícies (TELAU, 2012; AIGERMAN; PORANNE; LIPMAN, 2014).

1.1 Motivação

A parametrização de malhas, quando utilizada na computação gráfica, requer alto poder computacional devido ao rápido tempo de resposta requisitado. Para isso, a busca por diferentes tipos de algoritmos para realizar a parametrização de malhas de maneira mais eficiente é constante. O mais recente deles, parametrizações progressivas, apresenta um método onde

a convergência para o resultado ideal é mais rápida e necessita de menos passos. Os benefícios trazidos pela redução do tempo de execução da parametrização podem ser maximizados ao executar o programa em paralelo e explorando plataformas heterogêneas.

1.2 Objetivos

Tendo em vista os conceitos apresentados, o objetivo geral dessa pesquisa é melhorar o desempenho do algoritmo de parametrizações progressivas, paralelizando-o para execução em arquiteturas heterogêneas utilizando CPU e GPU.

Os objetivos específicos são:

- Paralelização do algoritmo;
- Balanceamento do processamento paralelo entre a CPU e a GPU;
- Análise do *speedup* da implementação paralela por número de *threads* do programa.

1.3 Organização

O trabalho está organizado da seguinte forma: na Seção 2, são apresentados os principais conceitos relacionados ao assunto desse trabalho. Na Seção 3, são apresentados e discutidos os trabalhos relacionados. A descrição do funcionamento e implementação do algoritmo de parametrizações progressivas são apresentadas na Seção 4. Na Seção 5, apresenta-se a metodologia do trabalho, discute-se a proposta inicial e suas limitações, e além disso, apresenta-se uma nova proposta de paralelização com os resultados obtidos. A Seção 6, apresenta as considerações finais da pesquisa realizada, discutindo os principais resultados, desafios e limitações.

2 FUNDAMENTAÇÃO TEÓRICA

Nessa seção serão apresentados os principais conceitos relacionados ao contexto no qual o trabalho está inserido. Desta forma, a Subseção 2.1 abordará os conceitos básicos de programação paralela, a classificação dos algoritmos paralelos e as métricas para análise de desempenho de programas paralelos. A Subseção 2.2 descreverá as arquiteturas paralelas e suas principais características. A Subseção 2.3 apresentará os métodos de parametrização de malhas, descrevendo suas principais características e aplicações.

2.1 Programação Paralela

O princípio básico da programação paralela, é que problemas maiores e mais complexos podem ser divididos em tarefas menores afim de permitir uma execução mais rápida de maneira paralela. Para viabilizar a paralelização diversos recursos estão disponíveis atualmente, entre

eles estão os computadores com mais de um processador, as redes de computadores interligados, os processadores *multicore* e os aceleradores gráficos (LEMOS, 2018).

Ao utilizar recursos de computação paralela, uma série de novos problemas são introduzidos em relação à computação sequencial: a divisão de tarefas entre as unidades de processamento, o balanceamento de carga, a sincronização e a tolerância a falhas. Além disso, bom desempenho e escalabilidade para aumento de unidades de processamento são essenciais para a resolução de um problema que utiliza técnicas de computação paralela (NASCIMENTO; MURTA, 2018).

Um dos maiores desafios encontrados no desenvolvimento dessas aplicações é o mapeamento do algoritmo para o *hardware* paralelo. Na primeira etapa do mapeamento, o *hardware* apropriado para a resolução do problema deve ser selecionado. Então, o algoritmo deve ser analisado para identificar as partes que podem usufruir de aumento de desempenho ao serem executadas em paralelo. Por último, a conversão do código sequencial deve ser realizada na linguagem do *hardware* selecionado e as devidas otimizações para a plataforma devem ser feitas. Esse mapeamento pode ser realizado de maneira pré-determinada em tempo de compilação alocando os recursos em cada unidade computacional ou de maneira dinâmica em tempo de execução utilizando informações do processador como voltagem, frequência e consumo de energia (WANG; PRAKASH; MITRA, 2018).

A programação paralela se mostra útil até mesmo em computadores de apenas um processador com vários *cores*. Através de técnicas de *multithreading* e utilizando as características multitarefas dos sistemas operacionais pode-se diminuir as exigências de *hardware* e melhorar a experiência de usuário em determinadas aplicações (LEMOS, 2018).

É importante destacar que nem todos os tipos de processadores paralelos funcionam da mesma maneira. É bastante comum arquiteturas em que o número de cores aumenta porém a velocidade do processador por *core* diminui. Esse tipo de processador é mais adequado para processamento bruto de grandes volumes de dados. No caso de processadores com poucos *cores* e com velocidade por *core* alta, algoritmos com alta interdependência de dados e de difícil paralelização podem ser aplicados (KIRK; HWU, 2010; EIJKHOUT; GEIJN; CHOW, 2016).

2.1.1 Métricas para Análise de Desempenho de Programas Paralelos

Nem todos os programas são completamente paralelizáveis. Na grande maioria dos casos, somente alguns trechos são eficientes quando paralelizados. A medida de desempenho do código paralelizado quando comparado com sua versão sequencial é chamada de *speedup*. A Equação 1 deve ser utilizada para obtenção do *speedup* de um programa (LEMOS, 2018).

$$S_p = \frac{T_1}{T_p}. \tag{1}$$

Onde T_1 é o tempo necessário para que um programa seja executado sequencialmente, T_p é o tempo necessário para que o programa seja executado em p unidades de processamento e S_p é

o *speedup*. A Lei de Amdahl (SHEN; PÉTROT, 2011) ainda determina um limite superior para o *speedup*, que é descrito pela Equação 2 (LEMOS, 2018).

$$S_p = \frac{1}{f_p/p + (1 - f_p)}. \quad (2)$$

Onde f_p é a fração do programa que pode ser paralelizada. Quando todo o código pode ser paralelizado, f_p será igual a 1 e portando o valor de S_p será p descrevendo o melhor caso possível (LEMOS, 2018).

2.2 Arquiteturas Paralelas

Algoritmos paralelos e arquiteturas paralelas estão bastante ligados de maneira que não é possível pensar em um algoritmo paralelo sem considerar a arquitetura que vai executá-lo. Não existe abstração quando se deseja executar um algoritmo em arquiteturas paralelas, os algoritmos são sempre enquadrados em quatro diferentes categorias: paralelismo a nível de dados, paralelismo a nível de instrução, paralelismo a nível de *thread* e paralelismo a nível de processo (GEBALI, 2011).

2.2.1 Arquiteturas *Multicore*

Devido a problemas de superaquecimento e altos custos de produção, o aumento da frequência de funcionamento dos processadores estava se estabilizando. Para sanar a necessidade do mercado de computadores com melhor desempenho surgiu a arquitetura *multicore*. Inserindo múltiplos núcleos de processamento no mesmo *chip*, a indústria encontrou uma alternativa para dar continuidade às melhorias esperadas pelo mercado (NASCIMENTO; MURTA, 2018). A trajetória da arquitetura *multicore* busca manter a velocidade da programação serial ao mesmo tempo em que introduz a possibilidade da computação paralela (KIRK; HWU, 2010).

Na arquitetura *multicore*, os recursos da unidade de processamento não são compartilhados. Portanto, cada unidade de processamento possui seus próprios recursos de execução de instruções e de armazenamento de memória *cache* separados (ROBERTS, 2006).

2.2.2 Arquiteturas *Many-Core*

A arquitetura *many-core* surgiu com o objetivo de otimizar a execução de programas paralelos. Ao contrário da arquitetura *multicore*, a arquitetura *many-core* é composta de um grande número de *cores* menores capazes de executar programas paralelos de forma massiva (KIRK; HWU, 2010). Coprocessadores e unidades de processamento gráficas (GPU) são exemplos de processadores *many-core*.

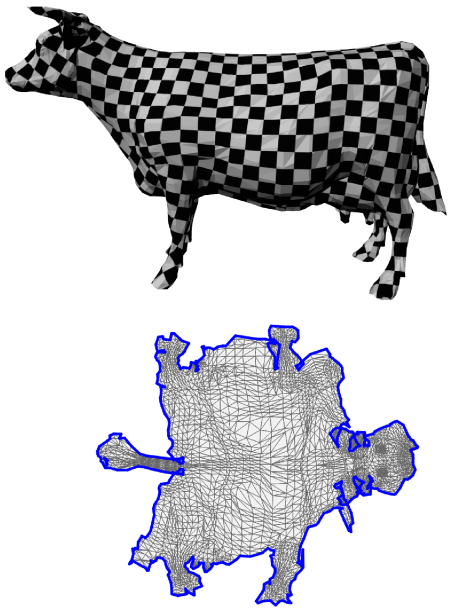
A arquitetura das GPUs é composta por uma matriz de multiprocessadores de streaming

(SM, do inglês *Streaming Multiprocessors*) altamente encadeados, geralmente com mais de uma unidade de processamento onde cada uma possui centenas de *cores* e recursos de memória *cache* próprios (RIOS, 2016).

2.3 O Problema da Parametrização de Malhas

A parametrização de malhas triangulares é o processo de mapear os vértices de uma superfície triangular em outra superfície, denominada de domínio parâmetro. Entre quaisquer superfícies com topologia similar existe um mapeamento bijetivo entre os pontos que as modelam (HORMANN; POLTHIER; SHEFFER, 2008). Para representar o domínio de parâmetro, a Figura 1 ilustra na parte superior o modelo e na parte inferior o domínio.

Figura 1 – Parametrização de um modelo 3D para uma superfície planar



Fonte: Smith e Schaefer (2015)

A parametrização é utilizada em diversos campos da computação. Entre os principais, podemos encontrar o mapeamento de texturas, mapeamento de detalhes, transformação de malhas e a compressão de superfícies. O mapeamento de texturas é utilizado em computação gráfica para facilitar o processo de renderização de texturas em modelos 3D. No mapeamento de detalhes, a parametrização é utilizada para armazenar características de determinado ponto da superfície como a normal ou o relevo. No caso da transformação de malhas, simulações e animações gráficas são os principais utilizadores. A parametrização permite que seja criado um mapeamento entre a malha inicial e a final para que seja realizada uma interpolação entre os pontos durante a animação. Na compressão de superfícies, a parametrização facilita o processo de compressão ao tornar a malha da superfície mais regular, gerando assim uma taxa de semelhança entre os triângulos (HORMANN; POLTHIER; SHEFFER, 2008; TELAU, 2012).

Para atingir um maior número de topologias diferentes, dois tipos de parametrizações são utilizados: a parametrização planar e a parametrização esférica. Na parametrização planar a superfície de domínio consiste em uma superfície triangular planificada, de modo que facilita tarefas que necessitem operar nos pontos da superfície, como o mapeamento de texturas. A parametrização esférica é mais apropriada para casos onde se deseja reduzir a distorção relativa à superfície original. A parametrização também pode ser realizada em uma superfície de domínio que seja topologicamente semelhante à superfície de manipulação para evitar distorções no resultado final (TELAU, 2012; LIU et al., 2008).

Considerando que o formato geométrico dos fragmentos da superfície domínio será diferente do formato dos triângulos originais, distorções de ângulo e área podem ser introduzidas. A distorção é um fator importante no processo de parametrização, por isso as aplicações tentam minimizar a distorção para a malha inteira (ATHANASIADIS; FUDOS, 2011).

O tempo e a quantidade de passos necessários para atingir uma solução ideal de parametrização é outro fator a ser considerado. Por ser utilizado em determinadas aplicações que necessitam um tempo de resposta rápido, implementações mais eficientes de algoritmos de parametrização podem produzir resultados melhores e com menos distorções (LIU et al., 2018).

3 TRABALHOS RELACIONADOS

Nessa seção serão apresentadas as pesquisas realizadas na área de computação paralela relevantes para a proposta inicial deste trabalho. Portanto, foram selecionados trabalhos que exploram algoritmos paralelos utilizando CPU e GPU. Na Subseção 3.1, será apresentada a paralelização de um algoritmo de parametrização esférica. Na Subseção 3.2, é tratada a otimização de algoritmos paralelos de malhas volumétricas em plataformas heterogêneas utilizando CPU e GPU. Por último, a Subseção 3.3 descreve um trabalho de refinamento de malhas de maneira adaptativa em programas para CPU e GPU.

3.1 Paralelização de Parametrizações Esféricas

Um dos principais problemas encontrados na parametrização de superfícies 3D é a distorção. A distorção introduzida pode inviabilizar o processo reverso ou diminuir a qualidade da malha. O problema é ainda maior quando se tenta produzir mapeamentos invertíveis para superfícies de domínio esférico. O trabalho de Athanasiadis (2011) e Fudos (2011) é introduzir uma nova forma de gerar mapas bijetivos entre topologias género 0[3] e superfícies esféricas.

Os algoritmos de parametrização esférica existentes podem ser divididos em duas categorias: métodos que tentam estender os métodos planares e métodos que utilizam otimizações não lineares. Os métodos da primeira categoria tentam primeiro realizar a parametrização para dis-

[3] O género de uma superfície caracteriza o número de buracos que a mesma possui. Portanto, uma esfera pode ser caracterizada como género 0 e um tóro como género 1.

cos planares para só depois fazer o mapeamento esférico. Já na segunda categoria, os métodos são conhecidos por introduzir menor distorção por serem realizados diretamente na esfera e por possuir alto custo computacional (ATHANASIADIS; FUDOS, 2011).

Para a resolução do problema, os autores propuseram um algoritmo de redução de energia combinado com um passo de otimização não linear para garantir o término do processamento e a validade do resultado. Para aumentar o *Speedup* nas técnicas de otimização não linear, os autores desenvolveram um *kernel* para a GPU escrito em OpenCL 1.1 (*Open Compute Library*). Para manter eficiente o processo entre as iterações, foi adicionado um passo de renormalização. Após a renormalização, ainda é feita a verificação da redução da energia afim de validar o processo de parametrização (ATHANASIADIS; FUDOS, 2011).

Os autores utilizaram implementações do código paralelo em CPU e GPU. Para maximizar o desempenho, vários fatores foram considerados: otimização do uso de memória, realização do teste de convergência a cada 1000 iterações para reduzir o custo de transporte de memória entre CPU e GPU, e implementação de alterações no algoritmo e nas estruturas de dados para viabilizar maior acerto de *cache* (*cache hit*).

3.2 Paralelização de Otimização de Malhas Volumétricas

Simulações computacionais são um ponto chave em muitas tentativas científicas e de engenharia que entre as suas aplicações, permitem que modelos e teorias sejam testados quando um protótipo físico é infactível ou impossível. Geralmente, essas simulações envolvem a criação de malhas de elementos compostos por triângulos e/ou tetraedros, e a qualidade e precisão dessas geometrias impactam diretamente o resultado da simulação, caracterizando um problema de otimização (CHENG et al., 2015).

Para resolver o problema de otimização, Cheng et al. (2015) utilizaram como métrica de qualidade a diferença da razão média inversa com um elemento ideal, um tetraedro equilátero. Para otimizações de interior e exterior da superfície, foram implementadas soluções sequencial, paralela na CPU e paralela na GPU para tornar possível uma comparação do aumento de desempenho entre múltiplas *threads* na CPU e a aceleração na GPU. Nelder-Mead (1965) foi utilizado como o principal algoritmo de otimização, com as variantes de duas e três dimensões. Para preservar as características da malha é utilizada uma técnica chamada quadriculação medial, que analisa a decomposição em autovalor da normal de um vértice. Esse método classifica as faces da malha, permitindo que o algoritmo de otimização execute diferentes ações para diferentes tipos de faces (CHENG et al., 2015).

Com o objetivo de tornar o processo de otimização das malhas mais rápido, foram utilizadas técnicas de programação paralela em GPU e em CPU. Como resultado foi percebido que a otimização por vértice é efetiva em sistemas heterogêneos. Em sua implementação, Cheng et al. (2015) utilizaram OpenMP (*Open Multi-Processing*) (CHANDRA, 2000) para paralelizar os algoritmos para CPU, e CUDA (NVIDIA Corporation, 2018) para criar os *kernels* executados

na GPU. Na GPU foram realizados testes utilizando as arquiteturas da Nvidia com o objetivo de otimizar o uso de memória e dos registradores de cada unidade de processamento. O paralelismo foi implementado dividindo o processamento dos vértices entre os processadores. Quando o vértice estiver localizado na superfície é atribuído para uma *thread* da CPU e quando não estiver para uma *thread* da GPU. Isso permitiu atingir paralelismo de grão fino a nível de vértice e paralelismo assimétrico utilizando CPU e GPU utilizando efetivamente os recursos de processamento.

3.3 Refinamento Adaptativo de Malhas em Arquiteturas Heterogêneas

A comunicação entre dois sistemas de memória é um dos principais desafios quando a aceleração por GPU é utilizada. Infelizmente, muitos modelos de programação para GPU de alto nível tem pouco conhecimento do algoritmo a ser executado. Por exemplo, eles podem traduzir uma função para ser executada na GPU mas o principal gargalo vai ser o estado da informação. Executar um algoritmo exclusivamente na GPU é uma opção para evitar o custo de transferência de memória, porém, em uma aplicação que utiliza refinamento adaptativo de malhas é interessante utilizar a CPU para tarefas seriais complexas (GUZIK; RILEY, 2018).

A técnica para o refinamento adaptativo de malhas utiliza uma série de grades sobrepostas que são decompostas em caixas e distribuídas entre nós da arquitetura paralela. Com os grãos mais finos atribuídos para a GPU, apenas informações de limites no entorno das grades finas precisam ser comunicados com a CPU. Os autores implementaram um escalonador dinâmico que é responsável por gerenciar as dependências de dados e organizar e disparar as transferências de memória entre os dispositivos, otimizando a utilização de todos os núcleos e comunicação de memória (GUZIK; RILEY, 2018).

Para esse estudo, Guzik e Jordan (2018) decidiram paralelizar um algoritmo de Lattice-Boltzmann[4] (BENZI; SUCCI; VERGASSOLA, 1992) para CPU e GPU. A implementação em ambas as plataformas foi otimizada usando toda a capacidade de vetorização do *hardware*. O desempenho do algoritmo é limitado pelo acesso à memória. Analisando a diferença de banda de memória entre as plataformas, é esperado que a implementação em GPU não seja mais do que duas vezes mais rápida do que a implementação em CPU. Tendo em vista que os algoritmos de dinâmica de fluídos possuem baixa intensidade de operações aritméticas, é mais eficiente particionar a memória em alto nível na aplicação do que utilizar a GPU como coprocessador e transferir a memória a cada tarefa.

Os autores decidiram por realizar a implementação usando as APIs do CUDA 9.0 para *kernels* da GPU e OpenMP para a paralelização na GPU. Além disso, a arquitetura paralela foi estendida para um *cluster* de computadores onde o escalonador dinâmico gerencia as cargas de trabalho utilizando OpenMPI (*Open Message Passing Interface*).

[4]Os algoritmos Lattice-Boltzmann são uma classe de algoritmos utilizados para simulação de dinâmica de fluídos.

3.4 Considerações da Seção

Os estudos apresentados mostraram que a paralelização e a exploração de ambientes híbridos pode trazer benefícios e viabilizar aplicações de algoritmos de parametrização de malhas, como no caso das parametrizações esféricas. Os resultados obtidos pela exploração do paralelismo de arquiteturas atuais podem viabilizar o uso aplicações que antes eram limitadas pelo poder computacional dessas mesmas arquiteturas, mas sem exploração do paralelismo. No entanto, fica clara a importância de compartilhar carga de processamento entre as diferentes unidades computacionais, um desafio ainda. Esse é um conhecimento que deve ser difundido e melhor estudado para que sejam obtidos melhores resultados em termos de desempenho e também o consumo energético seja reduzido.

O presente trabalho apresenta uma proposta de paralelização em CPU e GPU. O diferencial quando comparado a outros trabalhos apresentados, é o foco no compartilhamento de recursos entre as duas unidades de processamento para obter melhor performance na parametrização de malhas. A ideia inicial é utilizar a CPU para executar tarefas em paralelo que possuem maior interdependência entre os dados, enquanto a GPU será utilizada como coprocessador no processamento rápido de dados com pouca ou nenhuma interdependência.

4 PARAMETRIZAÇÕES PROGRESSIVAS

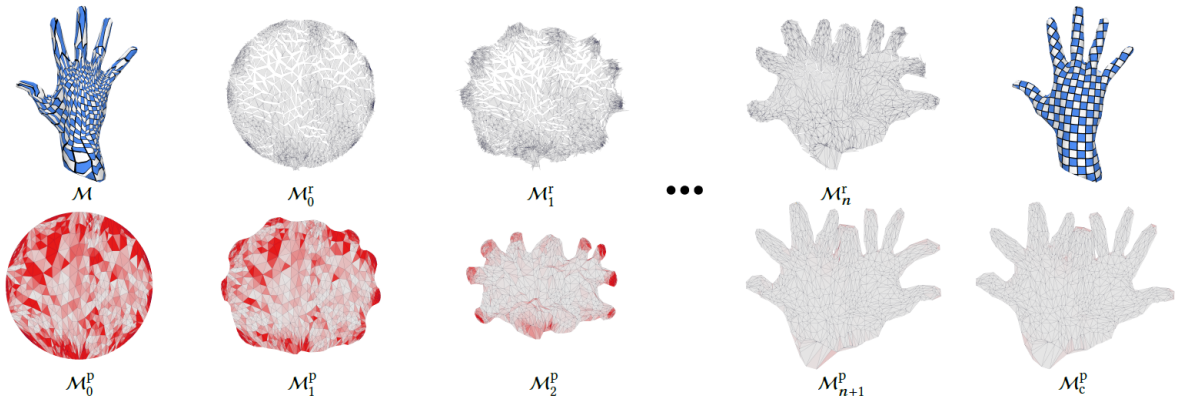
Ao contrário de outros métodos de parametrização que focam em resolver a otimização, o algoritmo de parametrização progressiva foca em observar a distorção ocorrida em cada triângulo da malha através de métricas de medição de distorção. O ponto chave do algoritmo é utilizar triângulos de uma malha de referência intermediária entre a malha original e a malha planar para determinar a energia de distorção. Liu et al. (2018) afirmam que em comparação com outros métodos de parametrização o método apresentado é mais simples e necessita de menos iterações para convergir para o resultado.

O processo de estimativa de distorção é realizado de forma iterativa para escolher os triângulos que melhor representam a malha. A cada iteração uma nova malha intermediária de referência é gerada e a energia de distorção calculada. Quando a malha de referência estiver próxima da malha ideal, a solução para o problema de otimização pode ser obtida. A Figura 2 demonstra o processo iterativo de geração de novas malhas de referência (LIU et al., 2018).

Para garantir uma parametrização sem dobras na superfície, é utilizado o método de incorporação de Tutte (TUTTE, 1963). Esse método consiste em processar a parametrização resolvendo um sistema linear esparso onde cada vértice é representado como uma média ponderada positiva de seus vizinhos (LIU et al., 2018).

A energia do sistema é numericamente difícil de otimizar, resultando em um grande número de iterações e alto custo computacional. Ao observar que a inicialização gerada pelo método Tutte causava pouca convergência nas primeiras iterações, os autores decidiram limitar as itera-

Figura 2 – Iterações de parametrização em uma superfície 3D



Fonte: Liu et al. (2018)

ções dos triângulos altamente distorcidos até que a redução obtida seja menor de 10%. Assim, o número de iterações necessário para a otimização foi reduzido pela metade para determinados modelos, resultando em uma convergência mais rápida (LIU et al., 2018; TUTTE, 1963).

Para obter a distorção da parametrização baseada nos triângulos ideais é necessário resolver sistemas de equações esparsas onde as matrizes são criadas a partir dos vértices dos modelos 3D. Para resolver essa equação, os autores propuseram um método híbrido utilizando os métodos já existentes: SLIM (do inglês, *Scalable Locally Injective Mappings*) (RABINOVICH et al., 2017) e CM (do inglês, *Composite Majorization*) (SHTENGEL et al., 2017). Ambos possuem características distintas e devem ser usados em determinados passos do algoritmo. O método SLIM é mais apropriado para reduzir distorções mais graves no começo do processo de minimização de distorção, porém, possui baixa taxa de convergência para a otimização ideal. O método CM não consegue reduzir superfícies altamente distorcidas, mas possui alta taxa de convergência para a otimização ideal. Portanto, o método SLIM é utilizado nas primeiras iterações do algoritmo afim de reduzir drasticamente o número de passos necessários, e o método CM é utilizado para convergir para o fim rapidamente quando as distorções mais graves já foram eliminadas (TELAU, 2012).

Para resolver problemas com equações lineares e matrizes esparsas, os autores decidiram implementar o algoritmo utilizando a biblioteca PARDISO (CONINCK et al., 2016; VERBOSIO et al., 2017; KOUROUNIS; FUCHS; SCHENK, 2018). Essa biblioteca permitiu abstrair algumas operações de código paralelo para CPU utilizando *multithreading* e instruções SIMD (*Single Instruction, Multiple Data*) em dados vetorizados.

Em comparação com as alternativas de parametrização, o algoritmo de parametrização progressiva tem em seu diferencial a rápida convergência para o resultado ideal. A biblioteca PARDISO contribui no resultado final, pois possibilita que todos os cálculos de sistemas lineares sejam executados de forma otimizada na CPU em termos de tempo de execução e de acesso à memória.

O código é basicamente dividido em 4 etapas:

- Carregamento: a superfície é carregada de um arquivo, os vetores que vão conter os índices de triângulos das faces e posição da malha são alocados e as normais das faces são calculadas.

- Pré-processamento: a fase de pré-processamento é onde acontecem os cálculos iniciais de área da superfície, e matrizes esparsas são calculadas utilizando as informações da malha previamente carregada. Nessa fase também é calculada a inserção de Tutte (TUTTE, 1963).

- SLIM: nas primeiras iterações da parametrização, o algoritmo SLIM é utilizado nas iterações em que ainda é possível reduzir a distorção em mais de 10%. Nessa etapa, matrizes esparsas são calculadas para serem utilizadas pela biblioteca PARDISO.

- CM: depois que a distorção é reduzida, o algoritmo CM é utilizado. Apesar de ser diferente do algoritmo SLIM, a implementação é bastante semelhante.

O algoritmo de parametrizações progressivas apresenta ganhos consideráveis quando comparado a outros algoritmos de parametrização. Porém, suas características de computação intensiva devem ser exploradas afim de torná-lo ainda mais eficiente. As fases de Pré-Processamento, SLIM e CM possuem laços de processamento de dados potencialmente paralelizáveis.

5 PARALELIZAÇÃO DO ALGORITMO *PARAMETRIZAÇÕES PROGRESSIVAS*

Nessa seção será apresentada a proposta de paralelização do algoritmo de parametrizações progressivas, bem como os resultados obtidos em plataformas *multicore*. A descrição de como o algoritmo foi paralelizado será dada a nível de código, para descrever as particularidades das etapas do algoritmo e as técnicas utilizadas na implementação. Na Subseção 5.1, será abordada a implementação da proposta inicial de paralelização do algoritmo para plataformas heterogêneas utilizando CPU e GPU, descrevendo os principais trechos e funções que podem ser paralelizados, além das bibliotecas de computação utilizadas. A Subseção 5.2 descreve os passos dados após a realização dos testes da proposta inicial, apresentando ferramentas utilizadas para inspeção do código e da paralelização. Na Subseção 5.3, será apresentado o *hardware* utilizado para realização dos testes do algoritmo em ambas as propostas. Na Subseção 5.4, os resultados da paralelização do algoritmo são comparados com a sua versão sequencial, utilizando para isso um conjunto de objetos previamente selecionados.

5.1 Proposta Inicial

Após o estudo do algoritmo e sua implementação por Liu et al. (2018), esta subseção apresenta os passos necessários para implementação do algoritmo em plataformas com CPU e GPU.

Visando executar o programa em ambientes sem interface gráfica e com o mínimo de interferência, a implementação do programa foi convertida para Linux. O código C++ foi extraído do projeto que tem como alvo a plataforma Windows e foi movido para uma estrutura CMake, permitindo assim utilizar diferentes compiladores. O foco da paralelização foi nas fases de pré-processamento, SLIM e CM. Na fase de pré-processamento somente em CPU, enquanto nas demais o processamento em GPU foi introduzido para substituir a biblioteca de sistemas lineares PARDISO.

O primeiro passo da paralelização foi na fase de pré-processamento. Existem três laços principais para os quais a paralelização se mostrou interessante. O primeiro, como pode ser visto no Apêndice A, calcula o inverso das coordenadas locais e não possui nenhum tipo de dependência com outros dados além das próprias coordenadas. Um laço paralelo se mostrou eficiente para melhorar o desempenho.

O segundo laço, demonstrado no Apêndice B, constrói uma matriz esparsa baseada nos vértices vizinhos do vértice de cada iteração. Como o número de vértices vizinhos não é constante entre a malha, a alocação dessa matriz é dinâmica e não permite que otimizações na memória sejam executadas quando convertido para execução em paralelo. Além disso, os índices dos vizinhos precisam estar ordenados dentro da matriz. Para resolver esse problema, foi adicionado um vetor de vetores de índices. Com os vetores auxiliares, uma matriz foi montada fora do laço paralelo evitando assim qualquer concorrência de acesso à memória.

Por último, o pré-processamento calcula uma matriz de índices baseada na matriz esparsa calculada anteriormente, para ser utilizada em outros processamentos da aplicação. Também é um laço sem interdependências e foi facilmente paralelizado. O código do laço paralelizado pode ser encontrado no Apêndice C.

A função de atualização da malha de referência é executada nas iterações dos algoritmos SLIM e CM. Os cálculos são executados por vértice ao percorrer todas as faces da superfície. Por possuir um laço que depende de uma variável com a distorção mínima, cada unidade de execução calculou a distorção mínima em seu conjunto de dados. Com a distorção calculada, a variável de distorção mínima global foi atualizada utilizando um mecanismo de sincronização de *threads* baseado em dispositivos de exclusão mútua (*mutex*). O resultado é demonstrado no Apêndice D.

Os algoritmos SLIM e CM podem ser divididos em duas partes relevantes para esse trabalho. A primeira parte é a montagem da matriz esparsa que será utilizada para calcular o sistema linear, e a segunda parte é a resolução do sistema linear. A montagem da matriz demanda certo processamento. Todas as faces precisam ser percorridas e os vértices utilizados para calcular a área e índices a serem utilizados no sistema. O processamento necessário cresce de maneira exponencial com o número de vértices do modelo. Por ter uma grande interdependência entre os dados da matriz a ser gerada, a paralelização só é viável para grandes volumes de dados. Portanto, nos resultados dessa pesquisa será considerada a implementação sequencial desse laço.

O último laço paralelizado foi o de atualização da malha de referência. Esse laço também não possuia nenhuma interdependência e foi possível aplicar uma estratégia simples de paralelização. A implementação pode ser encontrada no Apêndice E.

Para a resolução do sistema linear, uma biblioteca de resolução de sistemas lineares em GPU foi considerada para paralelização em plataformas heterogêneas. As bibliotecas investigadas para a resolução do problema foram: *ArrayFire* 3.6.4 (ARRAYFIRE, 2019), *cuSolver* 9.1 (NVIDIA Corporation, 2017), *CUSP* 0.5 (DALTON et al., 2014), *MAGMA* 2.5.1 (ANZT et al., 2014) e *ViennaCL* 1.7.1 (RUPP et al., 2016). No código original, a biblioteca PARDISO foi configurada para utilizar o método direto LU (MITTAL; AL-KURDI, 2002) na resolução do sistema e a matriz utilizada deve ser simétrica, onde somente a parte diagonal superior da matriz é calculada.

Nenhuma das bibliotecas possuía a funcionalidade de trabalhar com matrizes simétricas, onde somente um dos lados da matriz é informado. Para tornar possível a comparação dessas bibliotecas com a implementação original, foi implementada uma função para espelhamento da matriz esparsa. O tempo de execução do espelhamento foi desconsiderado do tempo total de execução para validar o tempo total da resolução do sistema em GPU.

A biblioteca *cuSolver* (NVIDIA Corporation, 2017) foi a única encontrada que possui métodos diretos implementados. O método LU na *cuSolver* (NVIDIA Corporation, 2017) só possui implementação em CPU, não sendo interessante para esse trabalho. O método QR (KERR; CAMPBELL; RICHARDS, 2009) possui implementação em GPU, porém ao utilizar com a matriz proveniente do algoritmo, a implementação apresenta instabilidades de execução por erros de corrupção de memória. Devido aos problemas encontrados, a biblioteca *cuSolver* (NVIDIA Corporation, 2017) foi descartada.

Devido à não existência de uma biblioteca com métodos diretos disponíveis, foi decidido a utilização de métodos iterativos para a resolução do sistema. Os métodos testados foram GMRES (*Generalized Minimal Residual*) (SAAD; SCHULTZ, 1986), CG (*Conjugate Gradiente*) (HESTENES; STIEFEL, 1952) e BiCGStab (*Biconjugate Gradient Stabilized*) (VORST, 1992).

A biblioteca *ArrayFire* (ARRAYFIRE, 2019) foi descartada pois sua implementação de resolução de sistemas lineares só funciona em matrizes densas, e a expansão da matriz esparsa para densa não é possível devido à uma limitação técnica das GPUs que possuem pouca memória para o algoritmo utilizado. A Tabela 1 demonstra o tamanho necessário para alocação somente da matriz densa dos oito maiores modelos testados com o algoritmo. Os modelos facilmente ultrapassam a memória disponível na GPU utilizada.

Tabela 1 – Tamanho das matrizes em memória utilizando o tipo elementos de dupla precisão

Modelo	**Linhas**	**Colunas**	**Memória**
D2_05030_n09	119608	119608	114,45GB
D1_05029	117361	117361	110,19GB
D2_05029_n05	116580	116580	108,73GB
D1_04199	60536	60536	29,32GB
D1_03441	57048	57048	26,03GB
D1_01885	47717	47717	18,21GB
D1_02331	44791	44791	16,05GB
D2_03914_n05	44488	44488	15,83GB

Fonte: Autor (2019)

As bibliotecas *MAGMA* (ANZT et al., 2014) e *ViennaCL* (RUPP et al., 2016) mostraram comportamentos semelhantes. Ambas atingiram o resultado esperado em determinados casos. Dependendo das configurações do método, pré-condicionador e tolerância numérica, diferentes tempos de execução eram obtidos devido ao fato da tolerância numérica não ser mantida comparando com a implementação original. Porém, nenhuma configuração mostrou redução considerável no tempo de execução, mantendo a mesma estabilidade numérica da implementação em CPU.

Como último recurso, a biblioteca *CUSP* (DALTON et al., 2014) foi utilizada. Por possuir algumas funcionalidades de depuração, foi possível verificar que os métodos utilizados precisavam de mais de mil iterações para convergir para a solução ideal, e alguns casos nunca atingiam a solução final.

Para identificar qual método iterativo seria ideal, um *script* Python foi desenvolvido utilizando a biblioteca *SciPy* 1.3.1 (VIRTANEN et al., 2019) para testar diferentes métodos e medir o número de iterações necessárias em cada um deles para atingir a solução ideal. Foi constatado que o método numérico LGMRES (*Loose Generalized Minimal Residual*) (BAKER; JESSUP; MANTEUFFEL, 2005) convergia em menos de trinta iterações em alguns casos. Porém, ainda não existe uma implementação do mesmo em GPU.

5.2 Paralelização para CPU

A implementação do algoritmo em plataformas utilizando GPU não foi possível, portanto novas estratégias foram pensadas. A primeira delas seria a implementação de um modelo paralelo *pipeline* em CPU. Pela divisão em etapas distintas do algoritmo, foi pensado em atribuir cada etapa para diferentes unidades de processamento. Porém, cada um dos passos do algoritmo cria uma nova matriz esparsa ou atualiza a malha de referência de forma que os dados não poderiam ser desacoplados de uma etapa e enviada para a próxima enquanto a etapa atual não fosse concluída. A nova proposta então foi entender por completo o comportamento da implementação paralela em CPU com o auxílio de ferramentas de verificação e tentar assim melhorar ainda

mais o paralelismo. Duas ferramentas foram utilizadas para melhorar a nova implementação do código: *Intel VTune Amplifier* 2019.6 (Intel Corporation, 2019d) e *Intel Advisor* 2019.5 (Intel Corporation, 2019a).

O *software Intel VTune Amplifier* (Intel Corporation, 2019d) foi utilizado para analisar o desempenho das partes paralelas da implementação. A ferramenta indicou algumas áreas onde existia tempo ocioso das *threads*, e também indicou taxa de boa utilização das *threads* na maior parte da execução.

Já a ferramenta *Intel Advisor* (Intel Corporation, 2019a) foi utilizada para obter sugestões de melhoria de desempenho na implementação. Com os resultados obtidos, foi possível investigar cada um dos laços da aplicação, entender os casos em que a memória é a principal a barreira para melhoria do desempenho, verificar a utilização da memória *cache* e ainda, verificar quais otimizações o compilador utiliza. Para uma melhor verificação na ferramenta, a aplicação teve de ser compilada com um compilador da Intel.

Com os resultados de ambas as ferramentas, foi definido que a aplicação necessita ser compilada com diretivas para uma melhor utilização das instruções de vetorização SSE (*Streaming SIMD Extensions*) (Intel Corporation, 2019c) e AVX (*Advanced Vector Extensions*) (Intel Corporation, 2019c) quando disponível. Também foi identificado que os principais laços do algoritmo paralelizado são afetados pelo desempenho da interface de memória.

Para a paralelização em CPU, a estratégia *map* foi aplicada utilizando o *template parallel_for* da biblioteca Intel TBB. O particionamento dos dados nos laços onde o *parallel_for* foi aplicado, foi realizado utilizando o particionador *auto_partitioner*. O particionador escolhido tenta balancear a carga de trabalho entre as *threads* dinamicamente. É ideal para execuções com cargas de trabalho de tamanhos diferentes. A biblioteca PARDISO também foi configurada para ser executada em modo paralelo.

5.3 Ambiente de Experimentação

Os testes foram realizados na plataforma computacional em nuvem da Google. Essa decisão foi tomada devido à facilidade de alocação de recursos e escalabilidade. As CPUs são virtualizadas, porém com dedicação exclusiva para a execução do programa.

O modelo da CPU utilizada foi Intel(R) Xeon(TM) da arquitetura *Cascade Lake* (Intel Corporation, 2019b) com frequência base 3,10GHz. A máquina possui 16 *cores* e o *Hyper Threading* não foi explorado para evitar concorrência de acesso à memória *cache*. Em termos de memória, a máquina possui 64GB de RAM (*Random Access Memory*). A GPU utilizada nos testes da proposta inicial foi uma Nvidia K80 com 12GB de RAM GDDR5 (*Graphics Double Data Rate*) e interface de 384 bits com capacidade de executar 2496 *threads* simultaneamente. Durante a execução dos testes, a máquina foi configurada para rodar em perfil de desempenho, com o processador no pico de seu funcionamento em 4GHz durante toda a execução.

O programa foi compilado utilizando o compilador GCC 7.4.0 com as diretivas de compila-

ção -O3 e -march=native. A diretiva -O3 permite ao compilador realizar otimizações no código de forma a ser executado de maneira mais rápida, sendo o último nível antes de remover precisão matemática. A diretiva -march=native permite ao compilador aplicar otimizações baseadas na arquitetura atual da máquina onde está ocorrendo a compilação. No ambiente de testes, as otimizações permitiram que o compilador utilizasse instruções vetorizadas SSE4 e AVX-512. As mesmas configurações foram utilizadas para execução serial e paralela.

O código foi configurado para ser executado com até 16 *threads* e para cada objeto os algoritmos sequencial e paralelo foram executados 15 vezes cada em cada configuração de *thread*. A partir de 15 execuções o desvio padrão já não era mais significante, portanto não foi necessário realizar testes com um número maior de execuções.

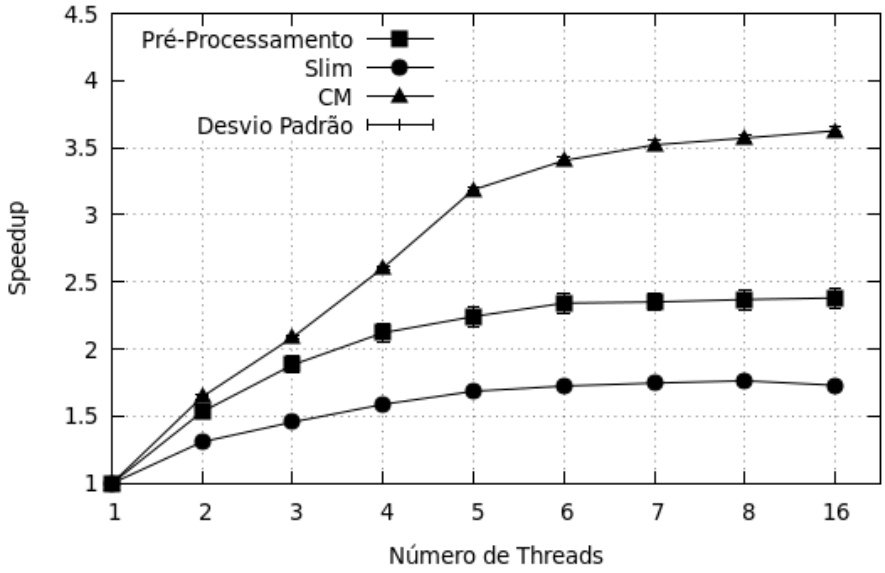
5.4 Resultados

Para a realização dos testes foram selecionados 15 objetos 3D dos *datasets* originais do algoritmo de parametrizações progressivas. Foram selecionados os objetos que necessitavam de mais tempo para a parametrização ideal. Necessitando de mais tempo e mais iterações para atingir o resultado ideal, os resultados obtidos pela paralelização do algoritmo forneceriam informações mais consistentes sobre o ganho de desempenho.

As implementações sequencial e paralela do algoritmo foram instrumentadas afim de obter o tempo de execução de cada parte do programa. As etapas selecionadas para obter informações de tempo de execução foram: a fase de pré-processamento, e os laços de processamento dos métodos SLIM e CM. Além disso, foi analisado o tempo total de execução do algoritmo.

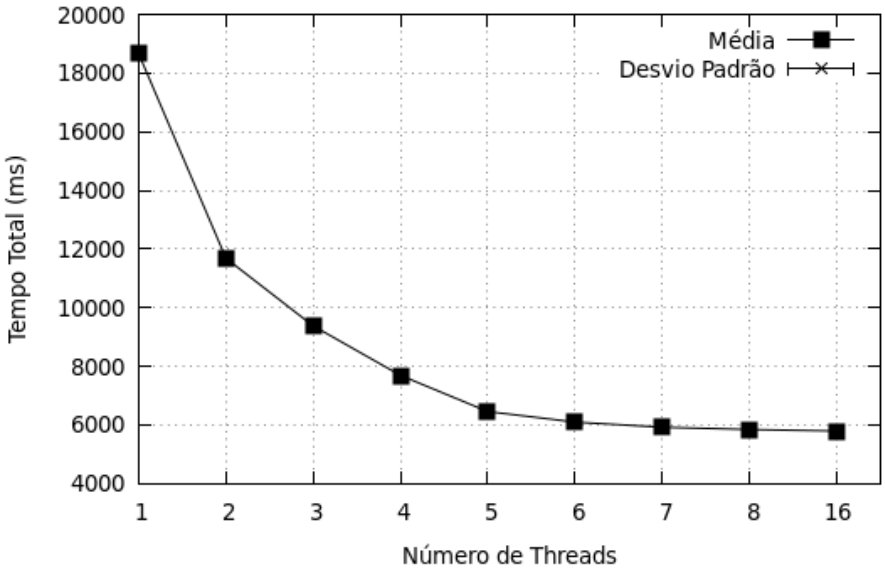
Com os resultados obtidos, foram calculadas a média aritmética e o desvio padrão de cada etapa. As Figuras 3 e 4 apresentam respectivamente o gráfico de *speedup* para cada fase do algoritmo e o gráfico de tempo de execução total do modelo D1_03441.

Figura 3 – *Speedup* obtido para o modelo D1_03441



Fonte: Autor (2019)

Figura 4 – Tempo de execução do modelo D1_03441



Fonte: Autor (2019)

A Tabela 2 mostra as médias de *speedup* atingido em cada um dos modelos 3D utilizados nos testes quando configurado para ser executado em 16 *threads*. A Tabela 3 apresenta o tempo de execução médio dos modelos quando configurado para 1 e 16 *threads*.

Tabela 2 – *Speedup* médio e desvio padrão dos modelos utilizados

Modelo	**Pré-Processamento**	**SLIM**	**CM**
D1_01885	2,150 ± 0,127	2,005 ± 0,028	3,270 ± 0,089
D1_02331	2,178 ± 0,100	1,775 ± 0,018	3,544 ± 0,069
D1_03441	2,381 ± 0,079	1,731 ± 0,020	3,626 ± 0,030
D1_04199	2,326 ± 0,069	1,958 ± 0,016	3,548 ± 0,024
D1_04824	2,297 ± 0,136	0,969 ± 0,022	3,387 ± 0,043
D1_05029	2,657 ± 0,145	2,062 ± 0,030	4,317 ± 0,051
D2_02697_n09	2,230 ± 0,029	2,640 ± 0,048	3,345 ± 0,055
D2_03914_n05	2,093 ± 0,053	4,520 ± 0,842	4,017 ± 0,451
D2_04592_n09	2,283 ± 0,100	2,189 ± 0,026	3,225 ± 0,307
D2_05029_n05	2,654 ± 0,129	4,139 ± 0,039	4,633 ± 0,143
D2_05030_n09	2,692 ± 0,142	4,205 ± 0,058	4,765 ± 0,073
D3_00069_per30_nr	1,852 ± 0,108	3,862 ± 0,076	3,062 ± 0,079
D3_04517_per30_nr	2,335 ± 0,054	6,133 ± 0,046	3,677 ± 0,371
D3_04531_per30	2,441 ± 0,062	6,065 ± 0,051	4,529 ± 0,337
D3_04532_per40_nr	2,458 ± 0,125	5,680 ± 0,046	4,278 ± 0,367

Fonte: Autor (2019)

Tabela 3 – Tempo de execução médio e desvio padrão dos modelos utilizados (em segundos)

Modelo	1 Thread	16 Threads
D1_01885	13606,867 ± 43,772	4601,067 ± 109,376
D1_02331	14038,667 ± 29,444	4424,467 ± 70,308
D1_03441	18660,467 ± 70,205	5793,067 ± 44,051
D1_04199	19246,667 ± 59,135	6063,267 ± 24,435
D1_04824	1583,533 ± 10,474	570,200 ± 5,144
D1_05029	34826,333 ± 241,084	9608,733 ± 73,425
D2_02697_n09	2323,533 ± 27,273	778,533 ± 4,704
D2_03914_n05	67282,000 ± 448,516	16252,533 ± 1418,641
D2_04592_n09	2992,067 ± 18,522	1050,000 ± 76,328
D2_05029_n05	88174,467 ± 575,684	19647,667 ± 300,178
D2_05030_n09	76816,667 ± 568,115	16931,533 ± 155,196
D3_00069_per30_nr	36890,200 ± 238,173	11431,067 ± 356,531
D3_04517_per30_nr	54047,667 ± 404,995	13075,667 ± 1274,406
D3_04531_per30	57245,533 ± 382,864	11693,867 ± 800,460
D3_04532_per40_nr	57889,067 ± 432,792	13061,533 ± 944,316

Fonte: Autor (2019)

Como se pode verificar nos resultados apresentados, o algoritmo demonstra capacidade de escalabilidade baixa. A etapa de pré-processamento atinge valores muito baixos em relação a configuração de *threads* utilizadas. Já as etapas SLIM e CM possuem variações dependendo do modelo. Alguns modelos necessitam de mais iterações na etapa SLIM e outros mais iterações na etapa CM. Para alguns modelos o *speedup* foi acima de 6 na etapa SLIM e para outros acima de 4,5 na etapa CM. Para uma configuração intermediária de *threads*, a redução no tempo total de execução ainda é relevante.

6 CONSIDERAÇÕES FINAIS

No contexto de aplicações de computação gráfica, a redução no tempo total de execução da aplicação pode ser considerada significativa quando o consumo energético da adição de novas unidades computacionais não for relevante. Em termos de *speedup*, ao comparar a execução paralela em 16 *threads* com a versão sequencial, determinadas etapas atingiram uma melhora de 6 vezes para alguns modelos. Porém, a melhora total entre todos os modelos foi de apenas 3 vezes, não sendo ideal para o número de *threads* utilizadas. Em termos de escalabilidade, o algoritmo demonstrou não escalar para mais de 8 *threads*. A partir de 6 *threads* o desempenho estabiliza e qualquer configuração após 8 *threads* não tem efeito positivo ou afeta negativamente e isso se deve ao fato dos principais laços também dependerem do desempenho de acesso à memória.

Os principais desafios encontrados durante a realização deste trabalho foram buscar por um método numérico ideal para os modelos, e encontrar implementações do mesmo em GPU. O algoritmo visa atingir uma grande gama de modelos 3D diferentes e cada modelo pode resultar em uma matriz esparsa com características distintas. A grande variedade de características geradas, limitou as possibilidades de métodos numéricos a serem utilizados por não existir um método ideal e inviabilizou a execução do algoritmo em GPU. Ainda, diversos casos de interdependência de dados foram encontrados e contornados com o particionamento dos dados ou com mecanismos de sincronização. Para validar o algoritmo implementado, foi necessário executar ambas implementações no *dataset* original e comparar os resultados.

As limitações de plataforma se deram no armazenamento e acesso à memória na GPU. Determinados métodos não puderam ser utilizados devido à pouca memória disponível para expandir as matrizes na GPU.

Como perspectiva para trabalhos futuros, a investigação na GPU ainda é relevante. Implementando novos métodos numéricos para execução em GPU é possível aumentar ainda mais a eficiência do programa. Outro fator importante a ser investigado são as diferentes características das matrizes esparsas produzidas pelo algoritmo em cada etapa. Assim, será possível determinar diferentes métodos a serem aplicados em cada situação.

Os resultados parciais desta pesquisa foram publicados nos Anais da XIX Escola Regional de Alto Desempenho da Região Sul em 2019 (KRANEN; CARMO, 2019).

Referências

AIGERMAN, N.; PORANNE, R.; LIPMAN, Y. Lifted Bijections for Low Distortion Surface Mappings. **ACM Transactions on Graphics**, Association for Computing Machinery (ACM), Nova Iorque, Nova Iorque, EUA, v. 33, n. 4, p. 1–12, jul 2014. Disponível em: <https://doi.org/10.1145/2601097.2601158>.

ANZT, H. et al. Optimizing Krylov Subspace Solvers on Graphics Processing Units. In: IEEE. **Fourth International Workshop on Accelerators and Hybrid Exascale Systems (AsHES)**.

Phoenix, Arizona, EUA: IEEE, 2014.

ARRAYFIRE. **ArrayFire 3.6.4**. 2019. <https://github.com/arrayfire/arrayfire>. Acesso em: 23 outubro 2019.

ATHANASIADIS, T.; FUDOS, I. Parallel Computation of Spherical Parameterizations for Mesh Analysis. **Computers & Graphics**, Elsevier BV, Reino Unido, v. 35, n. 3, p. 569–579, jun 2011. Disponível em: <https://doi.org/10.1016/j.cag.2011.03.022>.

BAKER, A. H.; JESSUP, E. R.; MANTEUFFEL, T. A Technique for Accelerating the Convergence of Restarted GMRES. **SIAM Journal on Matrix Analysis and Applications**, Filadélfia, Pensilvânia, EUA, v. 26, n. 4, p. 962–984, 2005.

BENZI, R.; SUCCI, S.; VERGASSOLA, M. The Lattice Boltzmann Equation: Theory and Applications. **Physics Reports**, Elsevier BV, Reino Unido, v. 222, n. 3, p. 145–197, dez 1992. Disponível em: <https://doi.org/10.1016/0370-1573(92)90090-m>.

CHANDRA, R. **Parallel Programming in OpenMP**. São Francisco, Califórnia, EUA: Morgan Kaufmann, 2000.

CHENG, Z. et al. Efficient Parallel Optimization of Volume Meshes on Heterogeneous Computing Systems. **Engineering with Computers**, Springer Nature, Alemanha, v. 33, n. 4, p. 717–726, jan 2015. Disponível em: <https://doi.org/10.1007/s00366-014-0393-7>.

CONINCK, A. D. et al. Needles: Toward Large-Scale Genomic Prediction with Marker-by-Environment Interaction. **Genetics**, Genetics, Rockville, Maryland, EUA, v. 203, n. 1, p. 543–555, 2016. ISSN 0016-6731. Disponível em: <http://dx.doi.org/10.1534/genetics.115.179887>.

DALTON, S. et al. **"Cusp: Generic Parallel Algorithms for Sparse Matrix and Graph Computations"**. 2014. <http://cusplibrary.github.io>. Acesso em: 23 outubro 2019.

EIJKHOUT, V.; GEIJN, R. van de; CHOW, E. **Introduction to High Performance Scientific Computing**. Genebra, Suíça: Zenodo, 2016. Disponível em: <https://zenodo.org/record/49897>.

GEBALI, F. **Algorithms and Parallel Computing**. Hoboken, Nova Jersey, EUA: John Wiley & Sons, Inc., 2011. Disponível em: <https://doi.org/10.1002/9780470932025>.

GUZIK, S. M.; RILEY, J. Adaptive Mesh Refinement on Parallel Heterogeneous (CPU/GPU) Architectures. In: **AIAA Aerospace Sciences Meeting**. Fort Collins, Colorado, EUA: American Institute of Aeronautics and Astronautics, 2018. p. 1–14. Disponível em: <https://doi.org/10.2514/6.2018-1831>.

HESTENES, M. R.; STIEFEL, E. Methods of conjugate gradients for solving linear systems. **J Res NIST**, Gaithersburg, Maryland, EUA, v. 49, n. 6, p. 409–436, 1952. Disponível em: <http://dx.doi.org/10.6028/jres.049.044>.

HORMANN, K.; POLTHIER, K.; SHEFFER, A. Mesh Parameterization. In: **ACM SIGGRAPH ASIA 2008 courses on - SIGGRAPH Asia 08**. Singapura: ACM Press, 2008. p. 1–87. Disponível em: <https://doi.org/10.1145/1508044.1508091>.

Intel Corporation. **Intel Advisor**. 2019. <https://software.intel.com/en-us/advisor>. Acesso em: 23 outubro 2019.

Intel Corporation. **Intel Cascade Lake**. 2019. <https://www.intel.com/content/www/us/en/design/products-and-solutions/processors-and-chipsets/cascade-lake/2nd-gen-intel-xeon-scalable-processors.html>. Acesso em: 04 novembro 2019.

Intel Corporation. **Intel Intrinsics Guide**. 2019. <https://software.intel.com/sites/landingpage/IntrinsicsGuide>. Acesso em: 04 novembro 2019.

Intel Corporation. **Intel VTune Amplifier**. 2019. <https://software.intel.com/en-us/vtune>. Acesso em: 23 outubro 2019.

KAUER, A. U.; SIQUEIRA, M. L. de. Anais da XIII Escola Regional de Alto Desempenho da Região Sul. In: . Porto Alegre, RS, Brasil: SBC, 2013. p. 135–138.

KERR, A.; CAMPBELL, D.; RICHARDS, M. QR Decomposition on GPUs. In: **Proceedings of 2Nd Workshop on General Purpose Processing on Graphics Processing Units**. Nova Iorque, Nova Iorque, EUA: ACM, 2009. (GPGPU-2), p. 71–78. ISBN 978-1-60558-517-8. Disponível em: <http://doi.acm.org/10.1145/1513895.1513904>.

KIRK, D. B.; HWU, W. mei W. **Programming Massively Parallel Processors: A Hands-on Approach**. Burlington, Massachusetts, EUA: Morgan Kaufmann, 2010. ISBN 0123814723.

KOUROUNIS, D.; FUCHS, A.; SCHENK, O. Towards the Next Generation of Multiperiod Optimal Power Flow Solvers. **IEEE Transactions on Power Systems**, EUA, v. 33, n. 4, p. 4005–4014, 2018. ISSN 0885-8950. Disponível em: <https://doi.org/10.1109/TPWRS.2017.2789187>.

KRANEN, A. C.; CARMO, A. B. do. Paralelização do Algoritmo Parametrizações Progressivas em Arquiteturas Multicore. In: **Anais da XIX Escola Regional de Alto Desempenho da Região Sul**. Porto Alegre, RS, Brasil: SBC, 2019. ISSN 2595-4164. Disponível em: <https://sol.sbc.org.br/index.php/eradrs/article/view/7050>.

LEMOS, J. D. **Aplicação de Técnicas de Paralelização de Programas usando OpenMP na Solução Numérica da Equação de Transporte de Nêutrons**. Tese (Doutorado) — Universidade Federal do Rio Grande do Sul, Porto Alegre, RS, Brasil, jul 2018.

LIU, L. et al. Progressive Parameterizations. **ACM Transactions on Graphics(SIGGRAPH)**, Nova Iorque, Nova Iorque, EUA, v. 37, n. 4, 2018.

LIU, L. et al. A Local/Global Approach to Mesh Parameterization. In: **Proceedings of the Symposium on Geometry Processing**. Aire-la-Ville, Suíça: Eurographics Association, 2008. p. 1495–1504. Disponível em: <http://dl.acm.org/citation.cfm?id=1731309.1731336>.

MITTAL, R.; AL-KURDI, A. LU-decomposition and numerical structure for solving large sparse nonsymmetric linear systems. **Computers & Mathematics with Applications**, Reino Unido, v. 43, n. 1-2, p. 131–155, 2002.

NASCIMENTO, J. P.; MURTA, C. Um Algoritmo Paralelo em Hadoop para Cálculo de Centralidade em Grafos Grandes. In: **XXX Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos**. Ouro Preto, Minas Gerais - Brasil: SBRC, 2018. p. 393–406.

NELDER, J. A.; MEAD, R. A Simplex Method for Function Minimization. **The Computer Journal**, Oxford University Press (OUP), Oxford, Reino Unido, v. 7, n. 4, p. 308–313, jan 1965. Disponível em: <https://doi.org/10.1093/comjnl/7.4.308>.

NVIDIA Corporation. **NVIDIA cuSolver Library**. 2017. <https://developer.nvidia.com/cusolver>. Acesso em: 23 outubro 2019.

NVIDIA Corporation. **CUDA C Programming Guide**. Santa Clara, Califórnia, EUA, 2018.

RABINOVICH, M. et al. Scalable Locally Injective Mappings. **ACM Trans. Graph.**, ACM, Nova Iorque, Nova Iorque, EUA, v. 36, n. 4, abr 2017. ISSN 0730-0301. Disponível em: <http://doi.acm.org/10.1145/3072959.2983621>.

RAUBER, T.; RÜNGER, G. **Parallel Programming**. Alemanha: Springer Berlin Heidelberg, 2010. Disponível em: <https://doi.org/10.1007/978-3-642-04818-0>.

RIOS, E. F. S. **Exploração de Estratégias de Busca Local em Ambientes CPU/GPU**. Tese (Doutorado) — UFF Universidade Federal Fluminense, Niterói, RJ, Brasil, jul 2016.

ROBERTS, J. A. S. **Multi-core Programming Increasing Performance through Software Multithreading**. Hillsboro, Oregon, EUA: Intel Corporation, 2006. ISBN 0976483246.

RUPP, K. et al. ViennaCL-Linear Algebra Library for Multi- and Many-Core Architectures. **SIAM Journal on Scientific Computing**, Filadélfia, Pensilvânia, EUA, out 2016. Disponível em: <http://dx.doi.org/10.1137/15m1026419>.

SAAD, Y.; SCHULTZ, M. H. GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems. **SIAM J. Sci. Stat. Comput.**, Society for Industrial and Applied Mathematics, Filadélfia, Pensilvânia, EUA, v. 7, n. 3, p. 856–869, jul. 1986. ISSN 0196-5204. Disponível em: <https://doi.org/10.1137/0907058>.

SANDERS, J.; KANDROT, E. **CUDA by Example: An Introduction to General-Purpose GPU Programming**. Boston, Massachusetts, EUA: Addison-Wesley Professional, 2010. ISBN 9780131387683.

SHEN, H.; PÉTROT, F. Using Amdahl's Law for Performance Analysis of Many-Core SoC Architectures Based on Functionally Asymmetric Processors. In: **Architecture of Computing Systems - ARCS 2011**. Alemanha: Springer Berlin Heidelberg, 2011. p. 38–49. Disponível em: <https://doi.org/10.1007/978-3-642-19137-4_4>.

SHTENGEL, A. et al. Geometric Optimization via Composite Majorization. **ACM Trans. Graph.**, ACM, Nova Iorque, Nova Iorque, EUA, v. 36, n. 4, p. 1–11, jul 2017. ISSN 0730-0301. Disponível em: <http://doi.acm.org/10.1145/3072959.3073618>.

SMITH, J.; SCHAEFER, S. Bijective Parameterization with Free Boundaries. **ACM Transactions on Graphics**, Association for Computing Machinery (ACM), Nova Iorque, Nova Iorque, EUA, v. 34, n. 4, p. 1–9, jul 2015. Disponível em: <https://doi.org/10.1145/2766947>.

TELAU, A. C. **Parametrizações de Superfícies Triangulares**. Tese (Doutorado) — Universidade Federal do Espirito Santo, Vitória, ES, Brasil, set 2012.

TUTTE, W. T. How to Draw a Graph. **Proceedings of the London Mathematical Society**, Oxford University Press (OUP), Oxford, Reino Unido, s3-13, n. 1, p. 743–767, 1963. Disponível em: <https://doi.org/10.1112/plms/s3-13.1.743>.

VERBOSIO, F. et al. Enhancing the Scalability of Selected Inversion Factorization Algorithms in Genomic Prediction. **Journal of Computational Science**, Reino Unido, v. 22, p. 99–108, 2017. ISSN 1877-7503. Disponível em: <https://doi.org/10.1016/j.jocs.2017.08.013>.

VIRTANEN, P. et al. SciPy 1.0–Fundamental Algorithms for Scientific Computing in Python. **arXiv e-prints**, Ithaca, Nova Iorque, EUA, p. arXiv:1907.10121, Jul 2019.

VORST, H. A. V. D. Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems. **SIAM Journal on Scientific and Statistical Computing**, Filadélfia, Pensilvânia, EUA, v. 13, n. 2, p. 631–644, 1992.

WANG, S.; PRAKASH, A.; MITRA, T. Software Support for Heterogeneous Computing. In: **IEEE Computer Society Annual Symposium on VLSI (ISVLSI)**. Hong Kong: IEEE, 2018. p. 756–762. Disponível em: <https://doi.org/10.1109/isvlsi.2018.00142>.

APÊNDICE A – PARALELIZAÇÃO DO LAÇO DE CÁLCULO DE COORDENADAS

```
for(int indiceFace = 0; indiceFace < numero_faces; ++indiceFace)
{
    auto coordenadas = ObtemCordenadasLocaisInversas(indiceFace);
    AtualizaMatrizReferencia(coordenadas);
}
```

```
tbb::parallel_for(tbb::blocked_range<int>(0, numero_faces),
                        [&]( const auto& intervalo )
{
    //Cada bloco é responsável por um intervalo de indices de faces
    for(const auto indiceFace : intervalo)
    {
        auto coordenadas = ObtemCordenadasLocaisInversas(indiceFace);
        AtualizaMatrizReferencia(coordenadas);
    }
});
```

APÊNDICE B – PARALELIZAÇÃO DO LAÇO DE CARREGAMENTO DE VÉRTICES

```
for(int i = 0; i < 2 * V_N; i++)
{
    if(i < V_N)
    {
        CarregaVertice(i);
    }
    else
    {
        CarregaVerticeOposto(i);
    }
}
```

```
tbb::parallel_for(tbb::blocked_range<int>(0, 2 * V_N), [&](const auto& range)
{
    for(const auto i : range)
    {
        if(i < V_N)
        {
            CarregaVertice(i);
        }
        else
        {
            CarregaVerticeOposto(i);
        }
    }
});
```

APÊNDICE C – PARALELIZAÇÃO DO LAÇO DE CARREGAMENTO DA MATRIZ ESPARSA

```
for(int indiceFace = 0; indiceFace < numero_faces; indiceFace++)
{
    auto verticesFace = ObtemVerticesFace(indiceFace);
    auto verticesFaceOposta = ObtemVerticesFace(indiceFace + numero_vertices);

    auto min_max = CalculaMinimoMaximo(verticesFace, verticesFaceOposta);

    carregaMatriz(indices_i, indices_j, min_max);
}
```

```
tbb::parallel_for(tbb::blocked_range<int>(0, F_N, 100),
                  [&](const auto& range)
{
    for(const auto indiceFace : range)
    {
        auto verticesFace = ObtemVerticesFace(indiceFace);
        auto verticesFaceOposta = ObtemVerticesFace(indiceFace
                                                    + numero_vertices);

        auto min_max = CalculaMinimoMaximo(verticesFace, verticesFaceOposta);

        carregaMatriz(indices_i, indices_j, min_max);
    }
});
```

APÊNDICE D – PARALELIZAÇÃO DO LAÇO DE CÁLCULO DE DISTORÇÃO

```
for(int indiceFace = 0; indiceFace < numero_faces; indiceFace++)
{
    auto verticesFace = ObtemVerticesFace(indiceFace);
    auto componente_q = CalculaComponenteQ(VetorPosicoes,
                                           verticesFace);
    auto componente_p = CalculaComponenteP(VetorPosicoes,
                                           verticesFace);

    auto j = componente_q * componente_p;
    auto energia = CalculaEnergia(j);
    double sig0 = CalculaSig0(j);
    double sig1 = CalculaSig1(j);

    if(energia <= limite_distorcao)
    {
        interacao++;
    }
    else
    {
        auto tt = newton_equation(sig0,
                                  sig1,
                                  limite_distorcao);
        if( tt < t_min )
        {
            t_min = tt;
        }
    }
}
```

↓

```
tbb::parallel_for(tbb::blocked_range<int>(0, F_N),
                        [&](const auto& range)
{
    double t_min_local = 1;

    for(const auto indiceFace : range)
    {
        auto verticesFace = ObtemVerticesFace(indiceFace);
        auto componente_q = CalculaComponenteQ(VetorPosicoes,
                                               verticesFace);
        auto componente_p = CalculaComponenteP(VetorPosicoes,
                                               verticesFace);
```

```
        auto j = componente_q * componente_p;
        auto energia = CalculaEnergia(j);
        double sig0 = CalculaSig0(j);
        double sig1 = CalculaSig1(j);

        if(energia <= limite_distorcao)
        {
            interacao++;
        }
        else
        {
            auto tt = newton_equation(sig0,
                                      sig1,
                                      limite_distorcao);
            if( tt < t_min_local )
            {
                t_min_local = tt;
            }
        }
    }

    mutex.lock();
    if( t_min_local < t_min )
    {
        t_min = t_min_local;
    }
    mutex.unlock();
});
```

APÊNDICE E – PARALELIZAÇÃO LAÇO ATUALIZAÇÃO DA MATRIZ REFERÊNCIA

```
for(int indiceFace = 0; indiceFace < numero_faces; indiceFace++)
{
    double sig0 = CalculaSig0(indiceFace);
    double sig1 = CalculaSig1(indiceFace);

    auto componente_w = CalculaComponenteW(sig0, sig1);
    auto componente_p = ObtemMatrizIteracaoAnterior();

    AtualizaMatrizReferencia(componente_w, componente_p)
}
```

```
tbb::parallel_for(tbb::blocked_range<int>(0, F_N), [&](const auto& range)
{
    for(const auto indiceFace : range)
    {
        double sig0 = CalculaSig0(indiceFace);
        double sig1 = CalculaSig1(indiceFace);

        auto componente_w = CalculaComponenteW(sig0, sig1);
        auto componente_p = ObtemMatrizIteracaoAnterior();

        AtualizaMatrizReferencia(componente_w, componente_p);
    }
});
```