UNIVERSIDADE FEDERAL DE PELOTAS
Centro de Desenvolvimento Tecnológico
Programa de Pós-Graduação em Computação

Dissertação

Uma Interface para Escalonamento de Algoritmos Iterativos em C++

Murilo Figueiredo Schmalfuss

Pelotas, 2020

Murilo Figueiredo Schmalfuss

Uma Interface para Escalonamento de Algoritmos Iterativos em C++

Dissertação apresentada ao Programa de Pós-Graduação em Computação do Centro de Desenvolvimento Tecnológico da Universidade Federal de Pelotas, como requisito parcial à obtenção do título de Mestre em Ciência da Computação.

Orientador: Prof. Dr. Gerson Geraldo H. Cavalheiro

Pelotas, 2020

Universidade Federal de Pelotas / Sistema de Bibliotecas
Catalogação na Publicação

S347i Schmalfuss, Murilo Figueiredo

Uma interface para escalonamento de algoritmos iterativos em C++ / Murilo Figueiredo Schmalfuss ; Gerson Geraldo Homrich Cavalheiro, orientador. — Pelotas, 2020.

84 f. : il.

Dissertação (Mestrado) — Programa de Pós-Graduação em Computação, Centro de Desenvolvimento Tecnológico, Universidade Federal de Pelotas, 2020.

1. Escalonamento. 2. Algoritmos iterativos. 3. Processamento paralelo. 4. Processamento de alto desempenho. I. Cavalheiro, Gerson Geraldo Homrich, orient. II. Título.

CDD : 005

Elaborada por Maria Inez Figueiredo Figas Machado CRB: 10/1612

Murilo Figueiredo Schmalfuss

Uma Interface para Escalonamento de Algoritmos Iterativos em C++

Dissertação aprovada, como requisito parcial, para obtenção do grau de Mestre em Ciência da Computação, Programa de Pós-Graduação em Computação, Centro de Desenvolvimento Tecnológico, Universidade Federal de Pelotas.

Data da Defesa: 23 de abril de 2020

Banca Examinadora:

Prof. Dr. Gerson Geraldo H. Cavalheiro (orientador)
Doutor em Computação pelo Institut National Polytechnique de Grenoble.

Prof. Dr. André Rauber Du Bois
Doutor em Computação pela Heriot-Watt University.

Prof. Dr. Dalvan Jair Griebler
Doutor em Computação pela University of Pisa.

Para Clarice e Waldemar.

AGRADECIMENTOS

Agradeço aos meus pais pelo apoio e confiança em todas as etapas dessa jornada. Sem a perseverança de vocês em me mostrarem o quão importante é o estudo não teria chegado até aqui. Usando as palavras de minha mãe: “A maior herança que posso te deixar é o estudo”.

Em especial agradeço à Nathália Rizzato, por ter me acompanhado desde o inicio do mestrado, sempre me apoiando, mesmo que por diversas vezes tenha ficado os finais de semana trabalhando no projeto ou no texto da dissertação. Sem teu apoio e incentivo não teria conseguido. Meu agradecimento vai além do que consigo expressar por palavras. Minha gratidão e amor por ti serão eternos.

Ao meu irmão Matheus agradeço pelas incontáveis vezes que pedi ajuda na correção de textos e pelos momentos de alegria que sempre me proporcionou.

Aos professores do PPGC da UFPel agradeço pelo conhecimento que compartilharam comigo, sou imensamente grato por tudo que pude aprender nesse período e pela amizade de vocês.

Agradeço aos meus amigos e colegas que participaram direta e indiretamente, seja ajudando no próprio projeto ou dando apoio para que eu conseguisse finalizar o meu mestrado.

Por fim gostaria de agradecer aos órgãos de fomento que me auxiliaram durante o mestrado.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Código de Financiamento 001.

“Só não existe o que não pode ser imaginado.”
— MURILO MENDES

“Todos los caminos están cerrados si uno no tiene una idea clara de a dónde quiere llegar.”
— MARIO LEVRERO

RESUMO

SCHMALFUSS, Murilo Figueiredo. **Uma Interface para Escalonamento de Algoritmos Iterativos em C++**. Orientador: Gerson Geraldo H. Cavalheiro. 2020. 84 f. Dissertação (Mestrado em Ciência da Computação) – Centro de Desenvolvimento Tecnológico, Universidade Federal de Pelotas, Pelotas, 2020.

Atualmente, as arquiteturas paralelas estão presente em praticamente todos os tipos de dispositivos, justificando uma busca por ferramentas e técnicas que facilitem o desenvolvimento de aplicações paralelas sem prejudicar o desempenho. As ferramentas atuais para programação paralela entendem essa necessidade e focam seus esforços em estratégias de escalonamento que tirem vantagem dos padrões paralelos comumente presentes no desenvolvimento de aplicações paralelas. Os algoritmos iterativos compõem uma grande parcela dos problemas computacionais, soluções que otimizem a execução desses algoritmos estão relacionadas à forma como as tarefas são divididas e escalonadas. Porém estas ferramentas não oferecem a flexibilidade para alterar a política de escalonamento, forçando o desenvolvedor a projetar a solução do problema de acordo com a ferramenta escolhida e sua forma de escalonamento, exigindo do desenvolvedor o conhecimento de detalhes sobre a implementação da ferramenta para extrair o máximo de desempenho. Essa abordagem oferece pouco reuso de código, e a portabilidade fica dependente das plataformas suportadas pela ferramenta.

Este trabalho propõe uma interface paralela orientada a objetos que permita a troca das políticas de escalonamento, permitindo a adequação da estratégia de escalonamento em função da necessidade do algoritmo. O padrão paralelo map foi utilizado como estudo de caso para validação e aferição do desempenho, comparando os resultados com outras ferramentas disponíveis para C++, como TBB e OpenMP.

Palavras-chave: Escalonamento. Algoritmos Iterativos. Processamento Paralelo. Processamento de Alto Desempenho.

ABSTRACT

SCHMALFUSS, Murilo Figueiredo. **An Interface for Scheduling Iterative Algorithms in C++**. Advisor: Gerson Geraldo H. Cavalheiro. 2020. 84 f. Dissertation (Masters in Computer Science) – Technology Development Center, Federal University of Pelotas, Pelotas, 2020.

Currently, parallel architectures are present in almost all kind of devices, justifying a search for tools and techniques that facilitate the development of parallel applications without jeopardizing performance. The current tools for parallel programming understand this need and focus their efforts on scheduling mechanisms that take advantage of common parallel patterns, present in the development of parallel applications. Iterative algorithms make up a large portion of computational problems, solutions that optimize the execution of these algorithms are related to the way the tasks are divided and scheduled. However, these tools do not offer the flexibility to change the scheduling policy, forcing the developer to solve problems according to the chosen tools and their way of scheduling, requiring the developer a knowledge of details about the implementation of tools to extract maximum performance.

This approach offers a parallel object-oriented interface that allows to switch scheduling policies, allowing the scheduling strategy to be adapted according to the use of the algorithm. The parallel pattern map was used for a case study of performance validation and performance, comparing the results with other tools available for C ++, such as TBB and OpenMP.

Keywords: Scheduling. Iterative Algorithms. Parallel Processing. High Performance Computing.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ALGORITMOS

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
DAG	Direct Acyclic Graph
EP	Embarrassingly Parallel
FIFO	First-In First-Out
GCC	GNU Compiler Collection
GPGPU	General Purpose Graphics Processing Unit
ICPC	Intel® C++ Compiler
LIFO	Last-In First-Out
NUMA	Non-Uniform Memory Access
OpenMP	Open Multi-Processing
PV	Processadores Virtuais
SIMD	Single Instruction Multiple Data
STL	Standard Template Library
TBB	Thread Building Block

SUMÁRIO

1 INTRODUÇÃO

Os avanços em sistemas computacionais construídos com processadores *multicores* demandam que ferramentas e programadores façam um melhor uso dos recursos de processamento paralelo agora disponível. As novas ferramentas de programação paralela buscam melhorar as estratégias de escalonamento para aproveitar melhor os recursos paralelos das arquiteturas, fazendo com que os processadores estejam ocupados na maior parte do tempo. De outro lado programadores precisam entender o problema e projetar as aplicações levando em consideração a ferramenta utilizada e o tipo de problema a ser resolvido, buscando encontrar padrões que facilitem a programação e permitam a reutilização de código, facilitando assim o desenvolvimento e ainda buscando utilizar todo o potencial de processamento oferecido na arquitetura.

Existem alguns itens que são inerentes a todos os problemas paralelos, que serão enumerados abaixo.

1. **Identificar tarefas.** É a análise da aplicação para identificar trechos de código que podem ser executados concorrentemente. Existe situações em que as tarefas são independentes entre si e podem ser executadas paralelamente. Em outros casos, podem existir dependências mútuas entre as tarefas, fazendo com que a execução destas implique na utilização de mecanismos de sincronização e/ou comunicação;

2. **Divisão dos dados.** Como a aplicação é dividida em tarefas, os dados a serem manipulados também precisam, por sua vez, ser distribuídos entre elas. É necessário que essa distribuição seja feita coordenadamente para que não ocorra a sobreposição de dados e não seja consumida mais memória do que o necessário para a execução da aplicação;

3. **Dependência de dados.** Os dados acessados pelas tarefas precisam ser verificados para saber se existe dependência de dados gerada por outras tarefas, caso exista é necessário que a execução das tarefas seja sincronizada para que o comportamento da aplicação seja o esperado;

4. **Escalonamento de tarefas.** Identificando as tarefas, é importante que elas sejam escalonadas entre os núcleos de processamento de acordo com alguma política de escalonamento para a redução de algum índice de desempenho, tipicamente o tempo de execução.

Neste trabalho, o problema abordado é o desenvolvimento de programas paralelos sobre arquiteturas multiprocessadas. O aspecto considerado é a dificuldade encontrada pelos programadores em desenvolver código concorrente, ao mesmo tempo em que precisam garantir a exploração eficiente dos recursos de *hardware* disponíveis. Atualmente, estão disponíveis ferramentas para programação em tais arquiteturas que abstraem, até certo ponto, os mecanismos para exploração eficiente do *hardware* e oferecem, em suas APIs (*Application Programming Interface*), facilidade de decomposição de paralelismo. No entanto, observa-se que estas ferramentas foram propostas apresentando recursos de programação novos, dissociados da linguagem de programação sobre a qual se apoia e com estratégias de escalonamento que aplicam heurísticas fixas, para as quais os algoritmos das aplicações devem ser moldados.

O enfoque de solução, discutido neste texto, é explorar a dissociação da descrição da concorrência da aplicação do seu suporte de execução em prol da produtividade de *software* e da exploração eficiente do *hardware*. A proposta recai sobre uma interface de programação que ofereça tanto facilidades para desenvolvimento de programas concorrentes quanto para adequação da estratégia de escalonamento a ser utilizado ao algoritmo obtido. Como requisito, a solução deve ser compatível com alguma linguagem de programação amplamente adotada para desenvolvimento de sistemas comerciais.

A seção seguinte dá continuidade a este capítulo introdutório apresentando conceitos explorados no desenvolvimento desta dissertação. O capítulo é concluído com a sumarização dos objetivos e resultados atingidos e com a apresentação da organização do texto.

1.1 Conceitos Introdutórios

Nesta seção serão apresentados alguns dos conceitos introdutórios relacionados a programação concorrente, como os modelos de *threads* e suas respectivas aplicações. Também é apresentado os principais padrões paralelos encontrados nos algoritmos paralelos.

Finalizando discutimos a relação entre padrões paralelos e modelos de *threads* nas principais ferramentas disponíveis.

1.1.1 Modelos de Implementação de Ferramentas Multithread

A implementação de uma ferramenta para programação *multithread* é realizada sobre um dos seguintes modelos: 1×1, $n \times 1$ e $n \times m$. Estes diferentes modelos oferecem diferentes alternativas para organização de *threads* em nível usuário e *threads* em nível sistema.

Threads de usuário são manipulados em nível usuário, portanto não dependendo do sistema operacional para cadenciamento de sua execução. *Threads* de sistema, por sua vez, são implementados e gerenciados diretamente pelo sistema operacional. *Threads* de usuário e *threads* de sistema são oferecidos nos modelos $n \times 1$ e 1×1, respectivamente. O modelo $n \times m$ oferece uma implementação mista de *threads* usuário, vezes denominados tarefas, e *threads* sistema, muitas vezes denominados *workers* ou processadores virtuais. Este último modelo permite a inserção de estratégias de escalonamento em nível usuário, de forma a mapear as tarefas descritas no programa sobre os processadores virtuais disponibilizados nas ferramentas.

Modelo multithread 1×1

A concorrência é mapeada diretamente sobre um fluxo de execução gerenciado pelo sistema operacional, como demonstrado na Figura 1 (SILBERSCHATZ; GALVIN; GAGNE, 2012). Este modelo permite que outro *thread* seja executado enquanto um *thread* executa uma chamada de sistema bloqueante. Também permite que múltiplos *threads* executem paralelamente em multiprocessadores. O ponto negativo é que criando um *thread* de usuário é necessário criar o *thread* de sistema correspondente. Esse *overhead* da criação dos *threads* pode prejudicar o desempenho da aplicação. A maioria das implementações desse modelo limitam o número de *threads* suportados pelo sistema. Como exemplo de ferramentas para programação paralela que utilizam esse modelo podemos citar *PThreads* (BUTENHOF, 1997) e as implementações da classe *Threads* de *C++* a partir da especificação de 2011 (ISO/IEC, 2012).

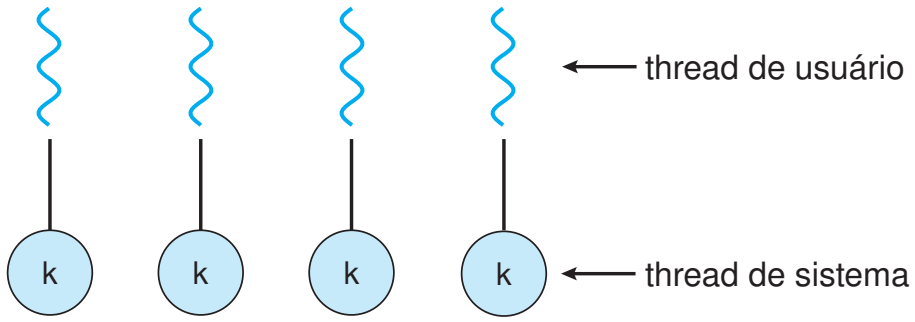

Figura 1 – Modelo *multithread* 1×1, o *thread* de usuário é mapeado diretamente sobre um fluxo de execução gerenciado pelo sistema operacional.

Modelo multithread $n \times 1$

O modelo $n \times 1$ representado na Figura 2 (SILBERSCHATZ; GALVIN; GAGNE, 2012) mapeia muitos *threads* de usuário para um *thread* de sistema. O gerenciamento do *thread* é feito pela biblioteca de *threads* no espaço do usuário, então é eficiente. No entanto, o processo todo será bloqueado se um *thread* fizer uma chamada de sistema bloqueante. Além disso, como somente um *thread* pode acessar o *kernel* de cada vez, vários *threads* não podem ser executados em paralelo em sistemas *multicore*. Os *green threads* (ORACLE, 2000) - uma biblioteca de *threads* disponíveis para sistemas *Solaris* e adotados em versões iniciais do *Java* - usavam o modelo $n \times 1$. No entanto, poucos sistemas continuam a usar o modelo devido à sua incapacidade de aproveitar múltiplos núcleos de processamento.

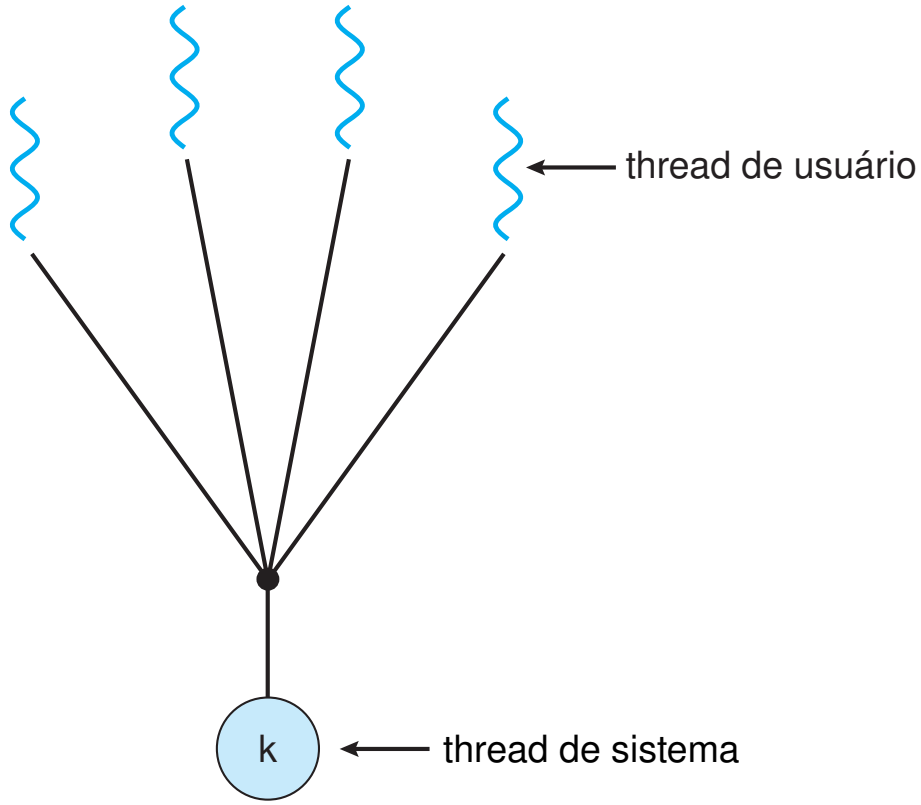

Figura 2 – Modelo *multithread* $n \times 1$, múltiplos *threads* de usuário são mapeados para um único *thread* de sistema, o gerenciamento dos *threads* é feito no espaço de usuário.

Modelo multithread $n \times m$

O modelo $n \times m$ mapeia muitos *threads* de usuário para um número menor ou igual de *threads* de sistema, um exemplo desse mapeamento pode ser visualizado na Figura 3 (SILBERSCHATZ; GALVIN; GAGNE, 2012). O número de *threads* de sistema pode ser específico para um aplicativo específico ou uma máquina particular (um aplicativo pode alocar mais *threads* de sistema em um multiprocessador do que em

um único processador). Considerando que o modelo $n \times 1$ permite que o programador crie tantos *threads* de usuário como desejar, ele não resulta em uma verdadeira concorrência, porque o *kernel* pode gerenciar apenas um *thread* por vez. O modelo 1×1 permite uma maior concorrência, mas o desenvolvedor deve ter cuidado para não criar muitos *threads* dentro de um programa (e em alguns casos pode ser limitado no número de *threads* que ele pode criar). O modelo $n \times m$ não sofre de nenhum desses problemas: os programadores podem criar tantos *threads* de usuário quanto necessário, e os *threads* de sistema correspondentes podem ser executados em paralelo em um multiprocessador. Além disso, quando um *thread* executa uma chamada de sistema bloqueante, o *kernel* pode agendar outro *thread* para execução.

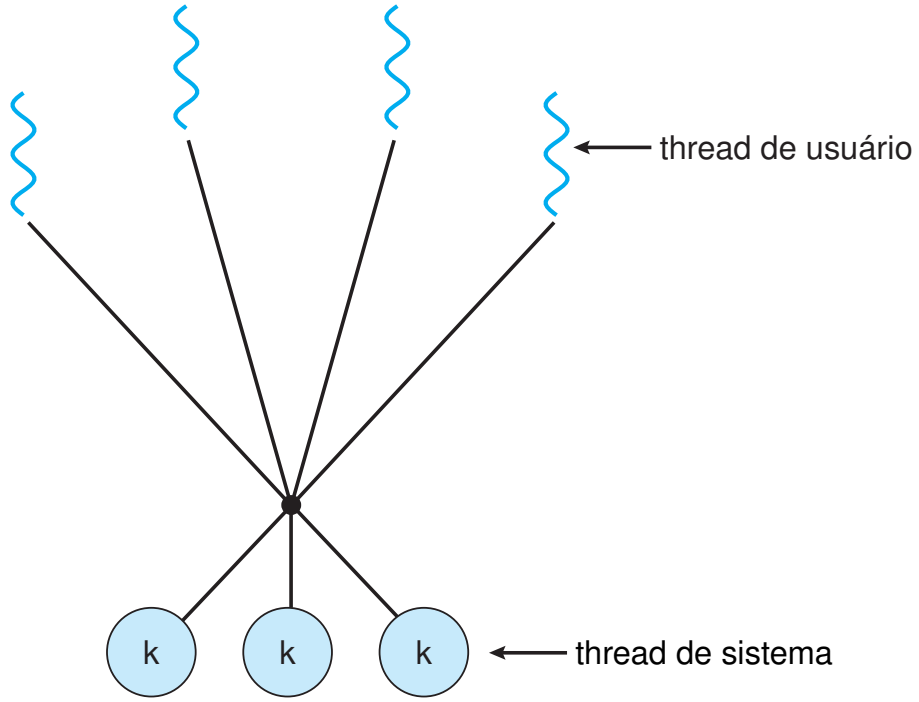

Figura 3 – Modelo *multithread* $n \times m$, múltiplos *threads* de usuário são mapeados para um número menor ou igual de *threads* de sistema, a heurística de escalonamento é implementada no espaço de usuário representado pela bolinha preta.

Uma variação no modelo $n \times m$ ainda mapeia muitos *threads* de usuário para um número menor ou igual de *threads* de sistema, mas também permite que um *thread* de usuário seja mapeado para um *thread* de sistema. Essa variação às vezes é referida como o modelo de dois níveis, Figura 4 (SILBERSCHATZ; GALVIN; GAGNE, 2012). O sistema operacional *Solaris* suportou o modelo de dois níveis em versões anteriores ao *Solaris* 9. No entanto, a partir do *Solaris* 9, esse sistema operacional utiliza o modelo 1×1.

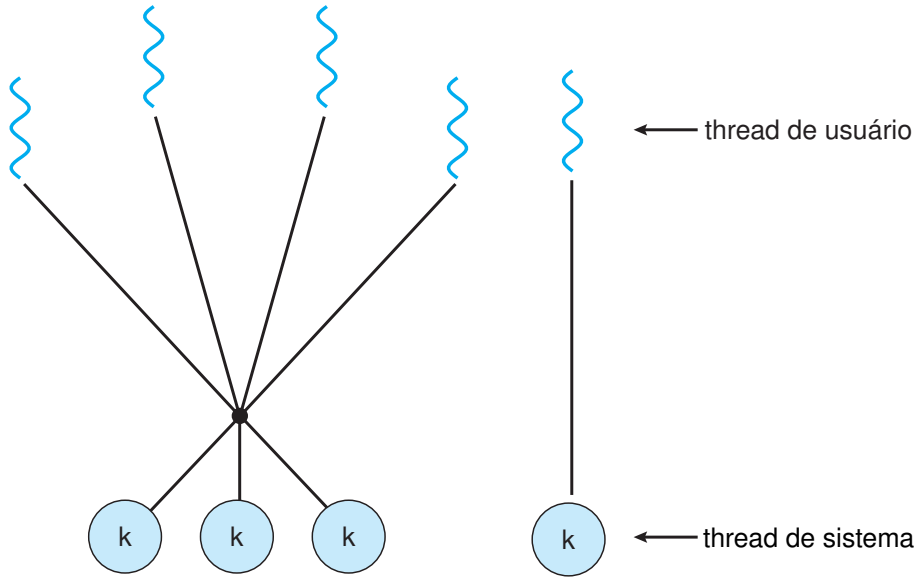

Figura 4 – Modelo *multithread* $n \times m$ de 2 níveis, múltiplos *threads* de usuário são mapeado sobre múltiplos *threads* de sistema e também é possível mapear um *thread* de usuário diretamente sobre um único fluxo de execução.

Considerações Sobre Modelos Multithread

Para o escalonamento de algoritmos iterativos é interessante o modelo $n \times m$, pois ele permite inserir estratégias de escalonamento no nível usuário. Esse controle do escalonamento permite que aplicações utilizem diferentes políticas de escalonamento que melhor se adaptem ao problema a ser resolvido. O modelo $n \times 1$ é ineficiente nas arquiteturas atuais, pois ele não explora efetivamente arquiteturas paralelas. O modelo 1×1 não é interessante pois delega o escalonamento ao sistema operacional, fugindo ao controle do programa.

Programar *multithread* no modelo de *threads* é limitar a aplicação a um número de *threads* disponíveis em um *hardware* especifico. O modelo de tarefas permite que o *hardware* seja explorado, mapeando as tarefas para os *threads* em tempo de execução sem afetar a lógica do problema.

1.1.2 Padrões Paralelos

Padrões para programação tornaram-se populares com as linguagens orientadas a objeto, como uma maneira de codificar boas práticas para a solução de problemas recorrentes em engenharia de software. Padrões também podem ser aplicados em programação paralela, e existem diversos padrões para programação paralela, alguns deles serão discutidos neste capítulo. Muitos trabalhos procuram entender e classificar a natureza de certas classes de problemas como em (ASANOVIC et al., 2006) e (ASANOVIC et al., 2009).

Map

No padrão *map* uma função, chamada função elementar, é replicada e aplicada a diferentes dados. Em sua versão paralela o padrão *map* é associado ao modelo SIMD (*Single Instruction Multiple Data*) (FLYNN, 1972), onde um mesmo conjunto de instruções são aplicados normalmente a todos elementos do conjunto de dados de entrada. Por ser comumente implementado como um laço paralelo as ferramentas que implementam esse padrão referem-se a ele como *parallel for* ou variantes desse nome.

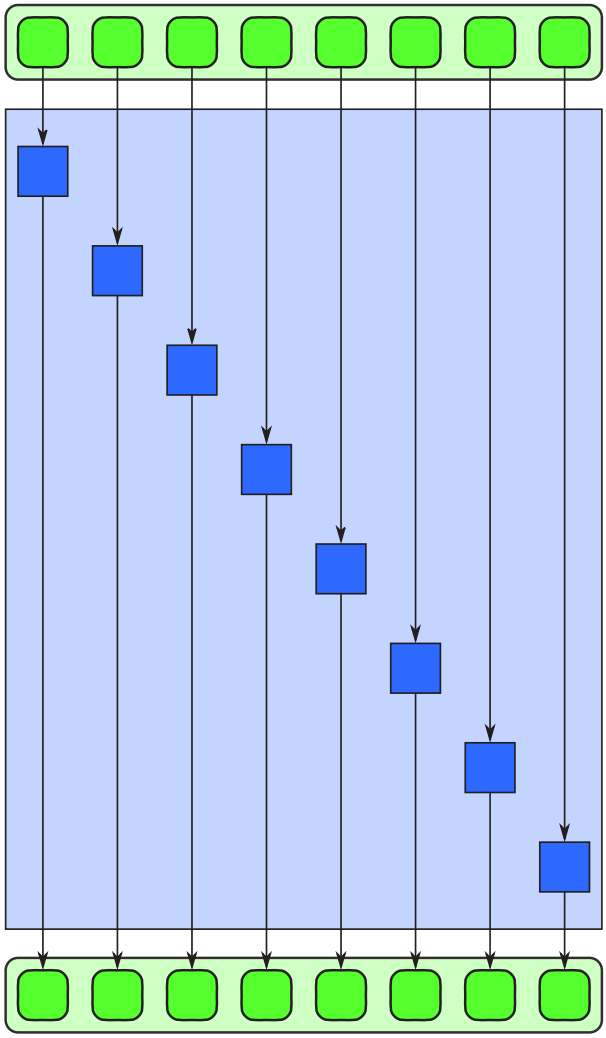

Figura 5 – Representação da versão sequencial do padrão *map*, a coleção de dados de entrada é representada pelo quadrado verde claro, os elementos pelos quadrados verdes e as funções elementares aplicadas a cada elemento da coleção de dados é representada pelos quadrados azuis, dentro do espaço de tempo representado pelo quadrado azul claro.

O padrão *map* corresponde a paralelização da iteração sequencial onde todas as iterações são independentes, na Figura 5 (MCCOOL; REINDERS; ROBISON, 2012) é apresentado um exemplo dessa iteração. Embora cada função seja aplicada em um tempo diferente, elas não possuem dependências entre si. Por esse motivo que o

padrão *map* é comumente expressado em modelos de programação como um *parallel for*, que é equivalente a forma utilizando uma função elementar.

A função elementar deve possuir algumas propriedades: (i) a função não pode possuir efeitos colaterais; (ii) a função pode ler dados de outras regiões de memória desde que estes dados não estejam sendo modificados em paralelo.

O padrão *map* é determinístico se os efeitos coltareis são evitados e não existe dependência entre as iterações. Isto garante que o conjunto de dados da saída não depende da ordem em que as várias instâncias da função elementar são executada.

Na Figura 6 (MCCOOL; REINDERS; ROBISON, 2012) é apresentada uma representação gráfica do padrão *map* em sua versão paralela, uma função elementar representada pelos quadrados azuis é aplicada a coleção de dados de entrada, gerando uma coleção de dados de saída de tamanho igual a de entrada.

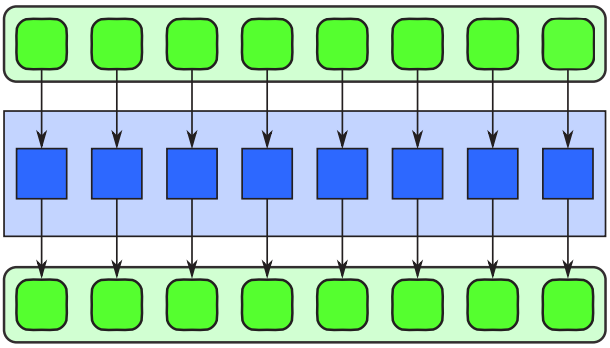

Figura 6 – Representação da versão paralela do padrão *map*, a coleção de dados é representada pelo quadrado verde claro, os elementos pelos quadrados verdes e as funções elementares aplicadas a cada elemento da coleção de dados é representada pelos quadrados azuis, dentro do espaço de tempo representado pelo quadrado azul claro, todas as funções são aplicadas no mesmo instante de tempo na representação.

Reduce

O padrão *reduce*, como o próprio nome sugere, reduz uma coleção de dados em um único elemento utilizando um operador combinatório associativo. O padrão *reduce* faz parte do conjunto de padrões conhecido como operações coletivas, por lidarem com uma coleção de dados ao invés de um único item.

Na redução uma função combinatória $f(a, b) = a \otimes b$ é utilizada para combinar todos os elementos de uma coleção em um elemento sumarizado, uma versão sequencial desse padrão pode ser visualizada na Figura 7 (MCCOOL; REINDERS; ROBISON, 2012). Assume-se que os elementos podem ser combinados em qualquer ordem, permitindo que múltiplas implementações possam ser exploradas.

Para algumas implementações a identidade da função combinatória é necessária, sendo esse valor interpretado também como o valor inicial da redução.

O problema da comutatividade e da associatividade é relevante quando a função

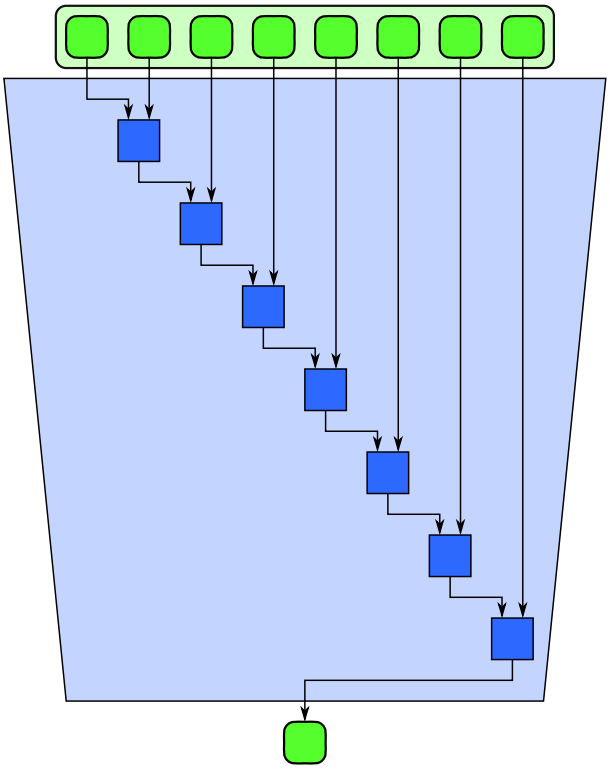

Figura 7 – Representação da versão sequencial do padrão *reduce*, a coleção de dados é representada pelo quadrado verde claro, os elementos pelos quadrados verdes e as funções combinatórias aplicadas aos elementos da coleção de dados é representada pelos quadrados azuis, dentro do espaço de tempo representado pelo quadrado azul claro. O resultado é um elemento sumarizado representado pelo quadrado verde na parte inferior da figura.

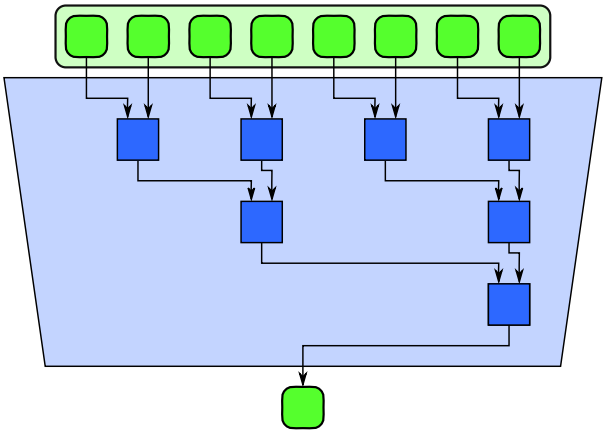

Figura 8 – Representação da versão paralela do padrão *reduce*, a coleção de dados é representada pelo quadrado verde claro, os elementos pelos quadrados verdes e as funções combinatórias aplicadas aos elementos da coleção de dados é representada pelos quadrados azuis, dentro do espaço de tempo representado pelo quadrado azul claro. O resultado é um elemento sumarizado representado pelo quadrado verde na parte inferior da figura, calculado a partir das reduções parciais.

combinatória deve ser definida pelo programador. É um problema difícil de identificar quando uma função arbitrária é associativa ou comutativa.

Outro problema que surge com a redução em uma coleção de dados muito grande é a precisão. Grandes somatórios tendem a ficarem sem *bits* para representarem os resultados intermediários. Funções associativas que utilizam ponto flutuante não são totalmente associativas (GOLDBERG, 1991), é necessário tomar cuidado ao definir a função combinatória levando em conta que muitas implementações reassociam as operações não-deterministicamente. Uma forma de implementação é utilizando-se reduções parciais, como representado na Figura 8 (MCCOOL; REINDERS; ROBISON, 2012).

Scan

Assim como o padrão de redução, o padrão *scan* é categorizado como uma operação coletiva, pois é dependente de toda a coleção de dados de entrada e não elementos separados como acontece no padrão *map*.

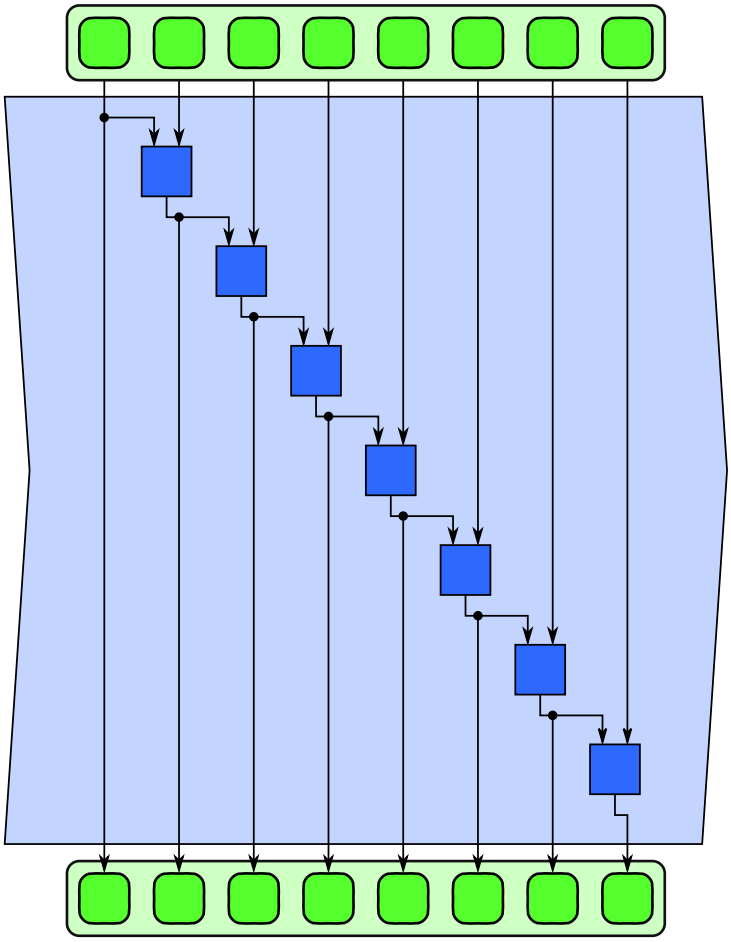

Figura 9 – Representação da versão sequencial do padrão *scan*, a coleção de dados é representada pelo quadrado verde claro, os elementos pelos quadrados verdes e as funções combinatórias aplicadas aos elementos da coleção de dados é representada pelos quadrados azuis, dentro do espaço de tempo representado pelo quadrado azul claro. O resultado é uma coleção de dados contendo as reduções parciais representado pelo quadrado verde claro na parte inferior da figura.

A operação coletiva *scan* produz todas as reduções parciais de um coleção de dados de entrada, gerando uma nova coleção de saída, como na versão sequencial exemplificada na Figura 9 (MCCOOL; REINDERS; ROBISON, 2012). Existem duas variantes do *scan*: (i) o *inclusive scan* onde o nth valor de saída é uma redução sobre os n primeiros valores de entrada; (ii) e o *exclusive scan* onde o nth valor de saída é uma redução sobre os $n-1$ primeiros elementos da coleção de entrada, ou seja, o nth elemento é excluído.

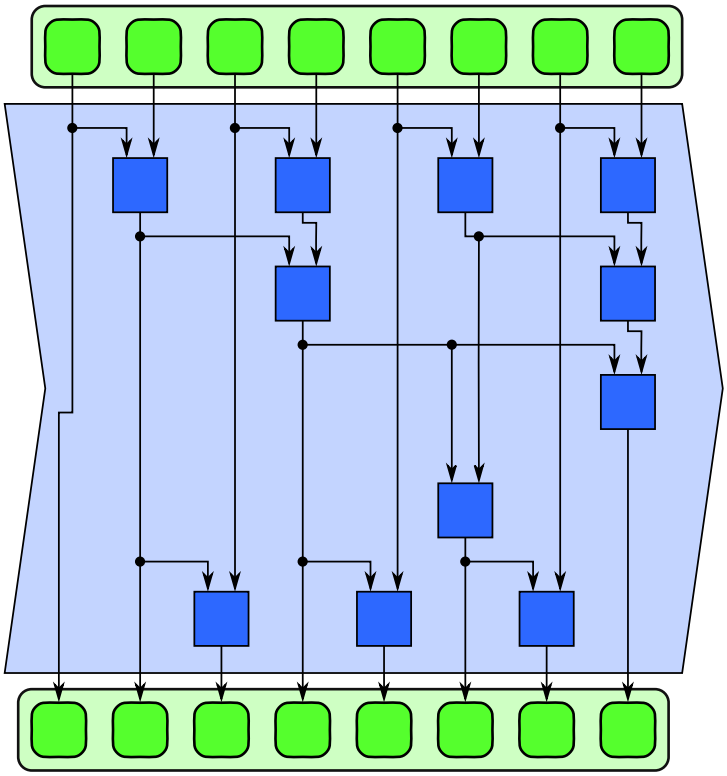

Figura 10 – Representação da versão paralela do padrão *scan*, a coleção de dados é representada pelo quadrado verde claro, os elementos pelos quadrados verdes e as funções combinatórias aplicadas aos elementos da coleção de dados é representada pelos quadrados azuis, dentro do espaço de tempo representado pelo quadrado azul claro. O resultado é uma coleção de dados contendo as reduções parciais executadas paralelamente num modelo *fork-join* representado pelo quadrado verde claro na parte inferior da figura.

De maneira análoga a paralelização do *reduce*, é possível tomar vantagem da associatividade da função redutora para reordenar as operações, como na Figura 10 (MCCOOL; REINDERS; ROBISON, 2012). Entretanto, ao contrário da redução, no *scan* existe a redundância de operações. Uma abordagem eficiente para paralelizar o padrão *scan* é baseado no padrão *fork-join*.

Stencil

O padrão *stencil* é um caso especial do padrão *map* discutido anteriormente, onde há um padrão de acesso regular aos dados. A saída do padrão *stencil* é uma função

de alguns elementos vizinhos da coleção de entrada, um exemplo desse parão de acesso pode ser visto na Figura 11 (MCCOOL; REINDERS; ROBISON, 2012).

Este acesso regular a coleção de dados de entrada permite algumas otimizações que podem ser aplicadas em sua implementação, por exemplo, reuso de dados e otimização da localidade de dados.

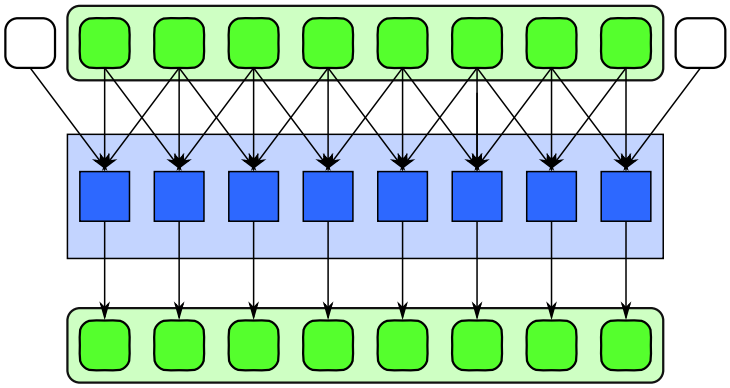

Figura 11 – Representação da versão paralela do padrão *stencil*. A coleção de dados de entrada é representada pelo quadrado verde claro, os elementos pelos quadrados verdes e os quadrados brancos representam os elementos das extremidades que são reusados para o cálculo do resultado. As funções combinatórias aplicadas paralelamente aos elementos da coleção de dados é representada pelos quadrados azuis, dentro do espaço de tempo representado pelo quadrado azul claro. A saída é uma coleção de dados representando os resultados em função do elemento e seus vizinhos.

Os padrões de acesso aos dados do padrão *stencil* são regulares e podem ser implementados tanto usando um conjunto de leituras aleatórias em cada função elementar ou utilizando um conjunto de *shifts*.

Considerações Sobre Padrões Paralelos

Na Tabela 1 estão listadas as principais ferramentas para programação paralela e seu suporte aos padrões apresentados neste Capítulo. A letra F simboliza que a ferramenta oferece suporte ao padrão como uma funcionalidade. A letra I indica que este padrão pode ser facilmente implementado utilizando-se outros recursos da ferramenta. Em C++ 17, como não existe uma biblioteca de escalonamento de *threads*, as diferente implementações utilizam-se de outras ferramentas para a execução, por exemplo, no *GCC* (*GNU Compiler Collection*) a implementação utiliza-se da biblioteca de execução do OpenMP (CARLINI et al., 2017), e no *ICPC* (*Intel C++ Compiler*), compilador da Intel, a implementação utiliza a biblioteca de execução do *TBB* (*Thread Building Blocks*) (INTEL, 2017a).

A vantagem dos padrões paralelos é apresentar soluções a problemas recorrentes de forma organizada, e abstraindo as decisões específicas da plataforma, tais como a sintaxe da linguagem de programação, a interface das bibliotecas, a arquitetura e o sistema operacional. A utilização de padrões faz com que as partes de identifica-

Tabela 1 – Comparação do suporte aos padrões paralelos entre as diferentes ferramentas mencionadas, onde F representa os padrões que estão implementados nas ferramentas e I representa os padrões que podem facilmente serem implementados nas ferramentas (MCCOOL; REINDERS; ROBISON, 2012; ISO/IEC, 2017).

Padrão	**TBB**	**Cilk Plus**	**OpenMP**	**C++ 17**
Map	F	F	F	F
Reduce	F	F	F	F
Scan	F	I	I	F
Stencil	I	I	I	I

ção de tarefas, divisão dos dados e dependência entre dados seja facilitadas para o programador. A implementação destes padrões em ferramentas para programação paralela acelera o desenvolvimento, uma vez que o programador apenas precisa identificar o tipo de padrão que deve ser utilizado e fazer uso da interface oferecida pela ferramenta.

O alto potencial de paralelismo de dados destes padrões paralelos mostra uma convergência para o modelo *multithread* $n \times m$. As ferramentas que implementam os padrões utilizam este modelo, e mesmo nos caso onde o modelo é 1×1, como em C++17, a implementação utiliza o *runtime* de alguma ferramenta paralela, seja OpenMP ou TBB.

1.2 Objetivos, Resultados e Impactos Esperados

No contexto da programação concorrente para sistemas *multithread*, devem ser considerados os problemas de desenvolvimento advindos do crescente número de recursos de processamento, chamados de *cores* ou CPUs, disponíveis. Em particular, no projeto de ferramentas de programação, deve ser considerada a capacidade dos programadores produzirem *software* robusto e eficiente de forma rápida. A ideia central da solução proposta é explorar a separação entre a expressão da concorrência do paralelismo oferecido pelo modelo de implementação de *threads* $n \times m$ de forma que estratégias de escalonamento possam ser implementadas ortogonalmente ao algoritmo da aplicação desenvolvida.

O objetivo central deste trabalho é apresentar uma interface de programação aplicativa (API) que disponibilize tanto recursos para decomposição de uma aplicação em um programa concorrente, como para desenvolvimento de estratégias de escalonamento das tarefas geradas pelo programa em execução.

O resultado materializando o objetivo central do trabalho é apresentado na forma de uma biblioteca de classes C++. A validação deste resultado é apresentada pela apresentação da implementação de um caso de estudo e comparação de desempenho deste caso de estudo com o desempenho obtido por OpenMP e TBB. TBB O

uso da biblioteca, em sua plena funcionalidade, é suficiente para desenvolvimento de programas concorrentes de forma rápida e robusta por programadores experientes em C++. No entanto, deve-se observar que espera-se também contribuir na discussão sobre as particularidades das estratégias de escalonamento pertencentes aos diferentes padrões de concorrência.

1.3 Organização do Texto

Após este capítulo inicial, onde, além de introduzir o trabalho desenvolvido, foram introduzidos conceitos associados aos fundamentos dos principais sujeitos da pesquisa, o restante do texto é organizado como segue.

No Capítulo 2 detalhamos os conceitos de escalonamentos de tarefas e o modelo de escalonamento aplicado nas principais bibliotecas para programação paralela (Cilk, TBB e OpenMP).

No Capítulo 3 apresentamos alguns trabalhos relacionados que apresentam abordagens semelhantes para solucionar problemas relacionado ao desempenho do algoritmo de escalonamento em arquiteturas ou algoritmos específicos.

No Capítulo 4 os conceitos aplicados no trabalho, como tarefas, *threads* e escalonamento, são discutidos.

No Capítulo 5 é descrito os detalhes da implementação da biblioteca.

No Capítulo 6 apresentamos a validação e os resultados da biblioteca desenvolvida, comparando nossa implementação com TBB e OpenMP.

E por fim no Capítulo 7 concluímos o trabalho com algumas considerações finais.

2 ESCALONAMENTO EM FERRAMENTAS DE PROGRAMAÇÃO MULTITHREAD

O escalonamento de tarefas atua como um gestor do uso dos recursos computacionais oferecidos por uma arquitetura de *hardware*. Seu papel, no contexto da discussão deste trabalho, é o de determinar quando e sobre qual unidade de processamento será lançada uma atividade do programa em execução. Na Seção 2.1 são apresentados conceitos do escalonamento de tarefas relevantes no contexto do escalonamento de tarefas em um ambiente de execução *multithread*. Nas seções seguintes, Seção 2.2, Seção 2.3 e Seção 2.4, são apresentadas algumas ferramentas de programação *multithread* implementadas no modelo $n \times m$ oferecendo um núcleo de escalonamento.

2.1 Conceitos de Escalonamento de Tarefas

O subsistema de escalonamento (CASAVANT; KUHL, 1988), em um sistema computacional, tem o papel de garantir a correta execução de um programa sobre os recursos de processamento disponíveis. Como requisitos de correção, este subsistema deve garantir que toda a demanda apresentada pelo programa será atendida em tempo finito e que um mesmo recurso não deve ser alocado a duas, ou mais, demandas no mesmo instante de tempo. Como requisito de desempenho, considera-se que o subsistema de escalonamento se apoie em alguma heurística para otimizar algum índice de desempenho. Dentre estes índices, encontramos a redução do tempo total de execução e o aumento de taxa de ocupação de CPU.

Considerando um núcleo de escalonamento instanciado para um determinado programa em execução, a expectativa é de que a heurística implementada vise a otimização da execução daquele programa. Um escalonamento deste tipo é denominado *escalonamento aplicativo* (FEITELSON, 1994), por executar no nível da aplicação. E exatamente por executar no nível da aplicação, este escalonador tem acesso a informações específicas do programa em execução, permitindo tomadas de decisões mais focadas nas características do algoritmo, expressa nas propriedades das tarefas geradas e suas relações de dependência.

Os núcleos de execução das modernas ferramentas de programação incorporam subsistemas de escalonamento aplicativo. Nestas ferramentas, o benefício é oferecido na implementação do modelo de *threads* $n \times m$. O escalonamento atua no ordenamento das n tarefas geradas pelo programa sobre m unidades de execução, usualmente *threads* sistema, disponibilizadas pelo núcleo. As heurísticas de escalonamento encontradas nestas ferramentas baseiam-se que o programador seguirá certas regras na construção dos seus programas. Estas regras definem como o relacionamento entre as tarefas deve ser construído para que a estrutura do programa possa ser conhecida e, então, explorada pelo escalonamento.

Nas seções a seguir, três ferramentas de programação oferecendo escalonamento aplicativo são apresentadas: Cilk Plus (Seção 2.2), TBB (Seção 2.3) e OpenMP (Seção 2.4). Destaque é dado para a interface de programação que oferecem e à estratégia de escalonamento aplicada

2.2 Modelo de Escalonamento do Cilk Plus

Uma execução *multithreaded* eficiente requer que existam *threads* prontas suficientes para manter o processador ocupado. Na tentativa de explorar esse paralelismo, os escalonadores, muitas vezes, acabam por criar muito mais paralelismo do que o computador é capaz de lidar, fazendo com que tais escalonadores facilmente excedam a capacidade de memória do computador.

Cilk Plus é uma extensão da linguagem C/C++, a criação de *threads* em Cilk Plus é mais eficiente que a criação de *threads* pelo sistema operacional (INTEL, 2017b), ele precisa apenas inserir uma tarefa em uma estrutura de dados, sem fazer chamadas ao sistema operacional ou involver o escalonador de *threads* do sistema operacional. Cilk Plus utiliza um escalonador *work-stealing* (BURTON; SLEEP, 1981) para balancear dinamicamente a carga.

Para obter um *speedup* linear e um consumo de memória linear, Cilk Plus restringe seu foco na classe de computações *multithreadeds* estritas (BLUMOFE; LEISERSON, 1998). Uma computação estrita é aquela em que nenhuma subrotina é chamada até que todos os parâmetros estejam disponíveis, embora eles possam ser gerados em paralelo.

Para obter essa expansão linear, Cilk Plus utiliza um algoritmo randômico, distribuído e *online*, que será explicado detalhadamente a seguir.

A Figura 12 demonstra um exemplo de computação *multithreaded* (BLUMOFE; LEISERSON, 1999). Cada bloco sombreado representa um *thread*, e os círculos dentro destes blocos representam tarefas. As arestas horizontais representam a ordenação sequencial das tarefas dentro de um *thread*. Para executar um *thread*, Cilk Plus aloca uma porção de memória que é chamada de *activation frame*, onde as tarefas de

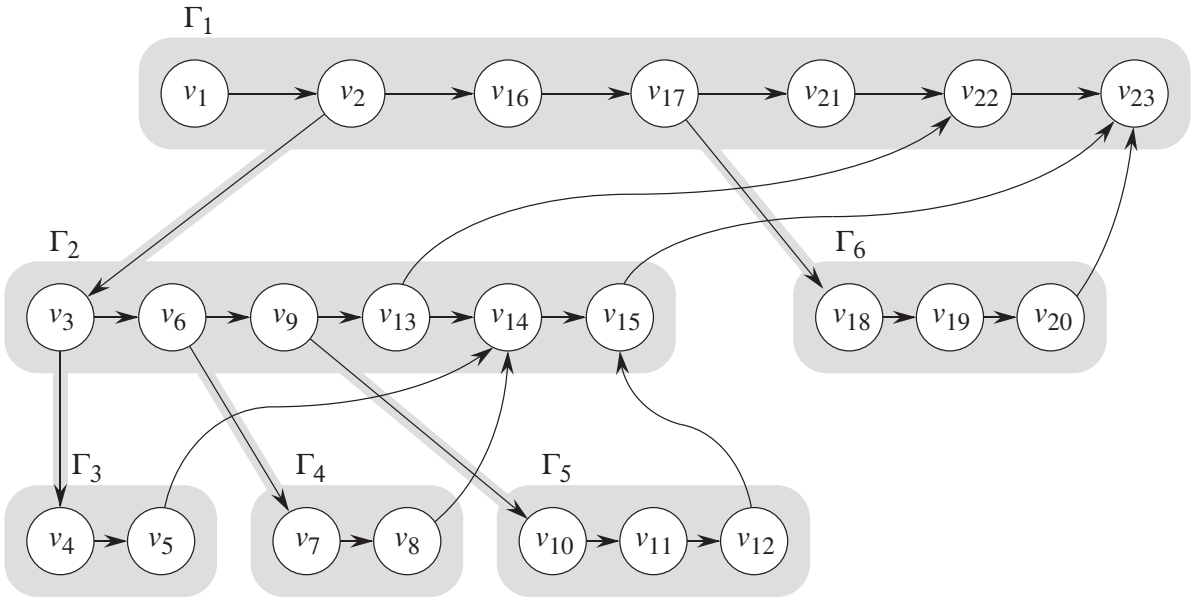

Figura 12 – Modelo de memória de Cilk Plus, os blocos sombreados representam um *thread* e os círculos dentro dos blocos representam as tarefas. As linhas horizontais representam a ordenação sequencial das tarefas dentro de um *thread*. As arestas sombreadas conectam *threads* com os *threads* filhos e as arestas curvadas representam os dados que a tarefa gera e que serão consumidos por outra tarefa.

um determinado *thread* podem armazenar os valores calculados.

Um *thread* durante sua execução pode criar novos *threads*, o termo utilizado por Cilk Plus para essa criação de novos *threads* é chamado *spawn*. A criação de um *thread* é semelhante a chamada de uma sub-rotina, exceto pelo fato de que ambos podem ser executadas concorrentemente. Um *thread* criado dessa forma é considerado um *thread* filho, e um *thread* pode criar quantos *threads* novos ele desejar.

As arestas sombreadas apontando para baixo conectam os *threads* com os *threads* filhos. Cada tarefa pode criar no máximo uma nova tarefa, quando a tarefa é executada é alocada um *activation frame*. Quando todas as tarefas de um *thread* são executadas, o *activation frame* é liberado e o *thread* é terminado.

O escalonador deve respeitar mais um tipo de dependência, uma tarefa que produz um dado que será consumido por uma outra tarefa, impede a execução da tarefa "consumidora". Está dependência está representada pelas arestas curvadas na Figura 12. Se um *thread* chega no ponto onde um *thread* "consumidor" depende de um dado que ainda não foi produzido, esse *thread* para. Assim que a dependência é resolvida, o *thread* torna-se apto a execução.

A decisão de escalonamento, desta forma, privilegia, em termos de desempenho algoritmos concorrentes escritos na forma aninhada-recursiva. A atuação do escalonamento é independente entre todos os *processadores virtuais* (PV) instanciados para suportar a execução do programa. Localmente, cada PV prioriza a execução da tarefa criada mais recentemente, com a perspectiva de atingir o nó folha da recursão e, en-

tão, desfazer o subgrafo do DAG que o representa, liberando memória. Novas tarefas criadas por um PV são mantidas em uma sublista de tarefas locais a ele. Caso esta sublista esteja vazia, ocorre o *roubo* de uma tarefa pertencente a sublista de outro PV selecionado aleatoriamente – neste caso, é priorizado o roubo da tarefa mais antiga. A hipótese neste caso é que esta tarefa por estar mais próxima à origem da estrutura aninhada-recursiva possua mais trabalho a ser desenrolado, mantendo o PV ocupado por mais tempo.

O escalonador deve obedecer as restrições de dependência de dados, *spawn* e ordenação da computação dentro de um *thread*. Estas arestas formam um grafo dirigido acíclico (DAG). A qualquer passo de execução um *thread* está pronto apenas se não há nenhuma dependência pendente. Um *thread* só pode executar se ele estiver no estado de pronto.

Em (BLUMOFE; LEISERSON, 1999) é provado que um limite superior de tempo de execução pode ser atingido por um escalonador guloso, aquele onde se P *threads* estão prontos então P *threads* serão executados, se um número menor que P de *threads* estão prontos então todos serão executados.

Em Cilk Plus uma tarefa, denominada *work*, é criada quando a palavra reservada `cilk_spawn` precede a chamada de uma função. Ela difere de uma chamada de função em C/C++ pelo fato de que a função invocadora pode executar concorrentemente com a função chamada.

A biblioteca de tempo de execução consulta o sistema operacional para determinar quantos núcleos de processamento estão presentes e cria uma *work thread* para cada núcleo.

Para uma execução rápida de uma computação em um *hardware* paralelo, requer que *threads* suficientes estejam vivas para manter os processadores ocupados. Em uma tentativa de explorar o paralelismo, os escalonadores podem acabar gerando mais paralelismo do que o *hardware* consegue suportar, cada *thread* ativo requer uma certa quantidade de memória, fazendo que estes escalonadores esgotem a capacidade de memória da máquina.

Embora o desenvolvimento de Cilk tenha sido descontinuado pela Intel (INTEL, 2017c) o desenvolvimento de Cilk continua ativo com o Open Cilk (SCHARDL; MOSES; LEISERSON, 2017).

2.3 Modelo de Escalonamento do TBB

Tarefas são as porções de computação executadas pelos processadores. O escalonador é responsável pelo mapeamento destas tarefas para *threads* de sistema e este mapeamento é não preemptivo. Em TBB, cada tarefa possuí seu método `execute()`, uma vez que um *thread* executa este método, a tarefa é associada a este *thread* até

que o método `execute()` retorne (INTEL, 2017d). Durante este período de tempo, o *thread* só executa outras tarefas caso esteja esperando que tarefas criadas a partir da tarefa associada sejam terminadas ou esteja aguardando a conclusão de construções paralelas aninhadas. Enquanto um *thread* espera ele pode executar outras tarefas que estejam disponíveis, sejam elas criadas por este *thread* ou não.

O escalonador de tarefas é destinado a paralelizar cargas de trabalhos intensivas. Como objetos de tarefas não são escalonados preemptivamente, ele devem de uma maneira geral evitar fazer chamadas que bloqueiem o *thread* por longos períodos durante o qual o *thread* não pode executar outras tarefas.

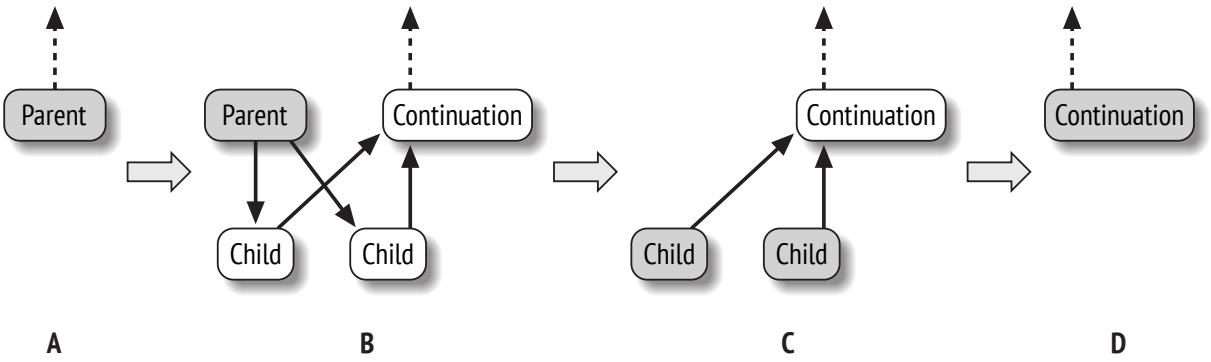

Figura 13 – Modelo *split*/*join* contínuo. A tarefa executando em cada passo é representado pelos blocos sombreados. No passo A a tarefa pai começa sua execução, no passo B cria duas tarefas filhas e uma tarefa de continuação, no passo C as tarefas filhas executam e no passo D ao terminarem as execuções das tarefas filhas é executada a tarefa de continuação.

O paralelismo em potencial é tipicamente gerado por um padrão *split/join* (REINDERS, 2007). Dois padrões básicos de *split/join* são suportados. O mais eficiente é a forma *continuation-passing*, onde o programador cria explicitamente tarefas de continuação. A tarefa pai cria tarefas filhas e especifica uma tarefa de continuação para quando as tarefas filhas terminarem. A tarefa de continuação herda então a tarefa pai da antecessora. A tarefa pai termina, não bloqueando as tarefas filhas. As tarefas filhas executam e a tarefa de continuação começa a rodar. A Figura 13, mostra estes passos. As tarefas executando em cada passo estão sombreadas.

O método de especificar a tarefa de continuação é eficiente pois ele desacopla a pilha de tarefas das tarefas, entretanto este método é mais complexo para programar. Um segundo padrão é o estilo bloqueante, que utiliza continuações implícitas. Este método é muitas vezes menos eficiente em termos de performance, mas conveniente para programação. Neste padrão a tarefa fica bloqueada enquanto suas tarefas filhas não são completadas, como demostrado na Figura 14.

A conveniência tem um custo, enquanto a tarefa pai bloqueada permanece na pilha, o *thread* pode roubar e rodar outra tarefa. Roubar tarefas e bloquear pode causar um crescimento na pilha. Para solucionar esse problema, o escalonador pode restringir

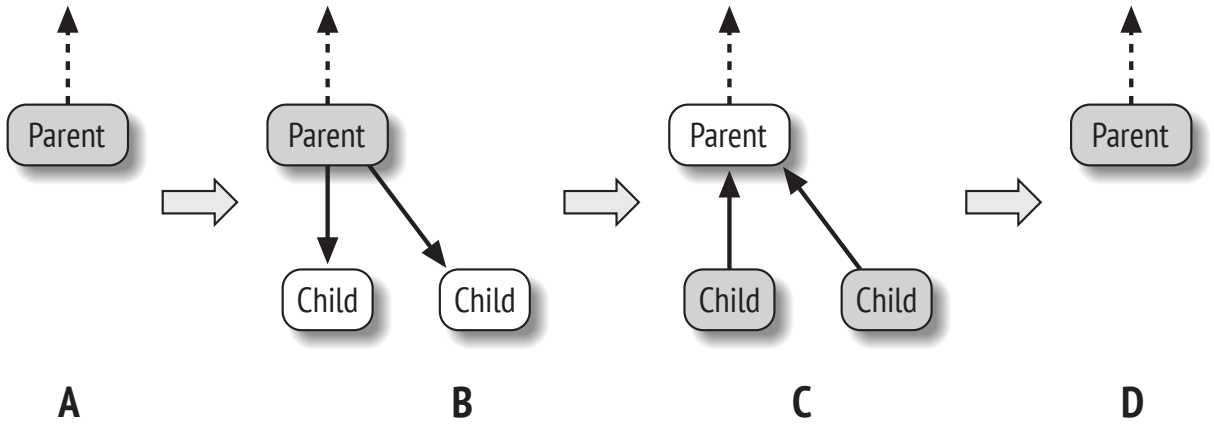

Figura 14 – Padrão *split*/*join* bloqueante. A tarefa executando em cada passo é representado pelos blocos sombreados. No passo A a tarefa pai começa a sua execução, no passo B ela cria duas tarefas filhas, no passo C a tarefa pai é bloqueada e as tarefas filhas começam a sua execução e no passo D ao terminarem as execuções das tarefas filhas a tarefa pai retoma a sua execução.

um *thread* bloqueado, para ele parar de roubar tarefas. Estas restrições limitam o paralelismo disponível e podem impactar na performance.

As construções de alto nível do TBB implementados no topo do escalonador de tarefas, como os algoritmos paralelos e o fluxo de grafos, utilizam passagem continua para paralelismo recursivo e bloqueando a conclusão de toda a construção paralela.

Tarefas em TBB são eficientes pois o escalonador é injusto, escalonadores de *threads* normalmente distribuem fatias de tempo numa política *round-robin*. Esta distribuição é chamada de “justa” pois cada *thread* de usuário recebe uma fatia de tempo aproximadamente igual. Os escalonadores geralmente utilizam esta maneira, pois é a política mais segura a tomar sem um conhecimento de alto-nível da organização da aplicação. Em programação baseada em tarefas, o escalonador de tarefas possui algumas informações de alto-nível e pode sacrificar a igualidade de distribuição de carga em favor da eficiência. Muitas vezes atrasando o início de uma tarefa até que ela possa fazer algum progresso útil.

O escalonador é o responsável pelo balanceamento de carga, além de usar o número certo de *threads* é importante distribuir a carga de trabalho uniformemente entre estes *threads*. Dividindo a aplicação em pequenas tarefas, o escalonador usualmente irá executar um bom trabalho mapeando tarefas para *threads* para balancear a carga. Com programação baseada em *threads*, o programador acaba lidando com o balanceamento da carga, algo que muitas vezes é difícil de se obter.

Algo importante a se observar ao utilizar tarefas ao invés de *threads*, é que elas permitem que o programador pense num nível mais abstrato, facilitando o desenvolvimento. Com o desenvolvimento em nível de *threads* o programador é forçado a pensar no nível de *threads* de sistema para obter uma boa eficiência, pois existe uma *thread* de usuário por *thread* de sistema, para evitar subutilização dos processadores, ou um

excesso de *threads* para serem executadas. Com *threads* também é necessário se preocupar com o grão da execução, com tarefas a preocupação é com as dependências lógicas entre tarefas, e a tarefa de escalonar eficientemente as tarefas é delegada ao algoritmo de escalonamento.

O escalonador avalia um grafo de tarefas. O grafo é um DAG (*Direct Acyclic Graph*) onde cada nodo é uma tarefa. Cada tarefa aponta para o seu sucessor, que é uma outra tarefa que está esperando para para ser executada, ou para NULL. Cada tarefa tem um parâmetro chama `refcount` que conta o número de tarefas que a tem como sucessora.

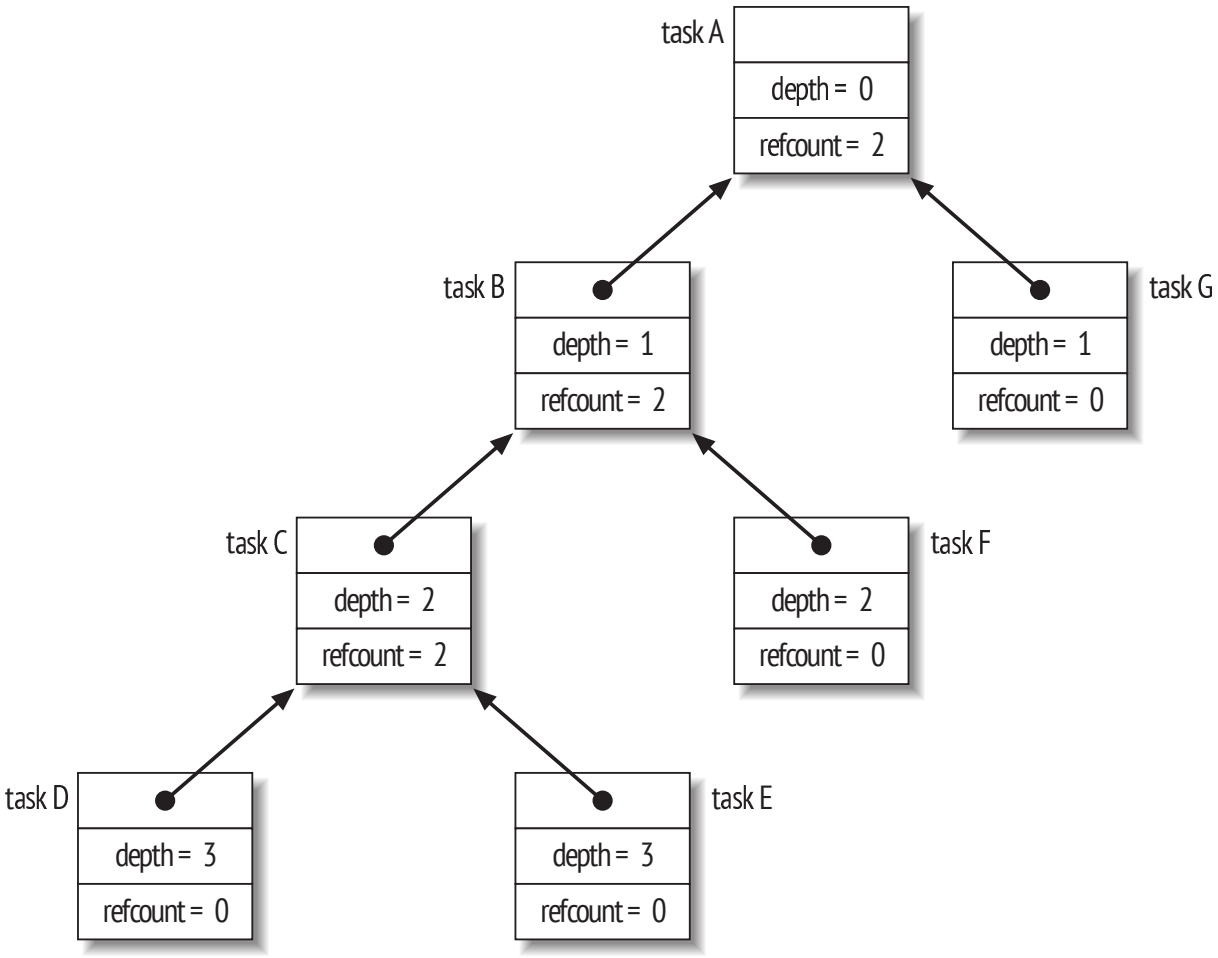

Figura 15 – Grafo de tarefas gerado por TBB. Cada tarefa aponta para uma tarefa sucessora e tem um contador de referências (refcount) que indica o número de tarefas que a tem como sucessora.

- As tarefas A, B e C criaram tarefas filhas que estão aguardando a execução, seus contadores de referência é o número de tarefas filhas mais 1. Este 1 é parte de uma convenção para espera explícita que será explicado posteriormente;
- A tarefa D está em execução, mas não criou nenhum tarefa, por isso não pode definir o seu contador de referências ainda;

- As tarefas E, F e G foram criadas, mas não tiveram sua execução iniciada ainda.

O escalonador executa as tarefas em uma maneira para minimizar tanto o consumo de memória como a comunicação entre *threads*. O objetivo é encontrar um equilíbrio entre execução em profundidade e em largura. Assumindo que a árvore é finita, em profundidade é melhor para execução sequencial pelos seguintes motivos.

- Quando a *cache* está quente as chances de acerto são maiores. As tarefas mais profundas foram criadas recentemente e ainda estão em *cache*. Considerando a Figura 15 (INTEL, 2017d), se as tarefas mais profundas forem executadas a tarefa C pode ser executada, e embora não seja a mais recente as chances de *hit* na *cache* são maiores que as demais tarefas;

- Executar a tarefa mais rasa leva a um desdobramento da árvore em largura, criando um número exponencial de nós que coexistem simultaneamente. A primeira execução de profundidade cria o mesmo número de nós, mas apenas um número linear deve existir ao mesmo tempo, porque empilha as outras tarefas prontas (E, F e G), minimizando o consumo de memória.

Embora execução em largura tenha o problema de consumo de memória, ela maximiza o paralelismo caso haja um número infinito de *threads* de sistema. Como *threads* de sistema são limitadas, o melhor é usar a execução de tarefas em largura apenas o suficiente para manter os processadores ocupados.

O escalonador do TBB implementa uma execução híbrida entre profundidade e largura. Cada *thread* tem sua própria fila de tarefas que estão prontas para execução. Quando um *thread* cria uma nova tarefa, ele insere a tarefa no fim da fila. A Figura 16 mostra um exemplo de fila de tarefas que corresponde ao grafo de tarefas da Figura 15 (INTEL, 2017d).

Quando um *thread* vai executar uma tarefa no gráfico de tarefas, ele executa a tarefa obtida pela primeira regra das listadas a seguir:

1. Pega a tarefa retornada pelo método `execute` da tarefa anterior. Esta regra não se aplica se o retorno for `NULL`;

2. Retira um tarefa da sua própria lista de tarefas. Está regra só se aplica caso a lista não esteja vazia;

3. Uma tarefa que tenha sido definido com afinidade com *thread*;

4. Rouba uma tarefa do topo da fila de tarefas de outro *thread*, se a fila estiver vazia tenta esta regra novamente até obter sucesso.

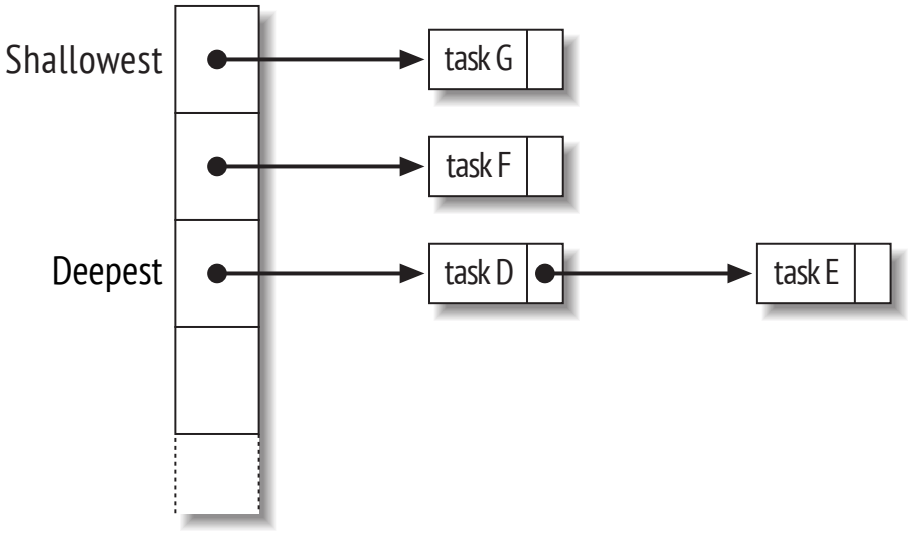

Figura 16 – Fila de tarefas no TBB. As tarefas criadas recentemente são as mais profundas, com chances maiores de *hit* na *cache*. As tarefas mais rasas possuem mais potencial de trabalho, possíveis vítimas para o roubo de trabalho.

O efeito da regra número 2 é executar a tarefa mais recentemente criada pelo *thread*, que ocasiona em um execução em profundidade até que o *thread* fique sem tarefas em sua lista. Quando isto ocorre ele rouba a tarefa mais antiga criada por outro *thread*, ocasionando em uma execução em largura temporária, convertendo um potencial paralelismo em um paralelismo de fato.

Para paralelismo aninhado, tarefas criadas tendem a seguir uma execução em profundidade, enquanto tarefas enfileiradas seguem uma execução em largura. Pelo espaço demandado pela execução em largura crescer de forma exponencial, tarefas enfileiradas devem ser utilizadas com cuidado.

De maneira geral utiliza-se tarefas criadas, a menos que haja uma razão para utilizar tarefas enfileiradas. Tarefas criadas geram um melhor balanceamento entre localidade, consumo de memória e paralelismo. O algoritmo para criação de tarefas é semelhante ao algoritmo de *work-stealing* do Cilk Plus (BLUMOFE, 1995).

Obter uma tarefa é sempre automático, é parte da avaliação do grafo de tarefas. Colocar uma tarefa em uma lista de tarefas pode ser explícito ou implícito. Um *thread* sempre coloca uma tarefa no fim de sua lista, nunca na lista de outro *thread*. Apenas pelo roubo de tarefa é possível transferir uma tarefa criada por um *thread* para outro.

Estas são as três condições que fazem com que um *thread* insira uma tarefa em sua lista:

- A tarefa é explicitamente criada pelo *thread*, por exemplo, pelo método `spawn`;
- Uma tarefa foi marcada para reexecução pelo método `task::recycle_to_reexecute`;

- O *thread* completou a execução da última tarefa e implicitamente decrementou seu contador de referências para 0. Caso isso ocorra, o *thread* coloca a tarefa sucessora no fim de sua lista.

Em essência, a estratégia do escalonador de tarefas é executar tarefas em profundidade e roubar tarefas em largura, assim como Cilk Plus. O roubo de tarefas em largura aumenta o paralelismo o suficiente para manter os *threads* ocupados. A execução de tarefas em profundidade mantém cada *thread* executando eficientemente uma vez que haja trabalho suficiente para ser executado. A diferença em relação à Cilk Plus é que TBB oferece em sua API diversas operações para manipular diretamente o DAG, expondo esta estrutura ao programador e, consequentemente, aumentando a complexidade de uso da ferramenta por exigir mesclar no algoritmo da aplicação a interação com o núcleo de escalonamento.

O escalonador de tarefas funciona mais eficientemente para paralelismo do tipo *fork-join* com muitos *forks*, dessa maneira o roubo de tarefas em largura ocupa os *threads* e estes executam em profundidade, até que precisem roubar mais tarefas.

2.4 Modelo de Escalonamento do OpenMP

Execução paralela requer a utilização de *threads*, e o desenvolvimento de um programa *multithread* muitas vezes pode ser algo complexo. O OpenMP, com a utilização de *pragmas*, consegue paralelizar laços que não contenham dependências entre os dados com uma única declaração, este tipo de paralelismo é chamado de *work-sharing* em OpenMP (*Open Multi-Processing*) (INTEL, 2018a) e será explicado nesta Seção.

A grande maioria dos laços podem ser paralelizados adicionando um *pragma* antes do laço. O termo *work-sharing* é usado para descrever a distribuição de trabalho entre os *threads* (CHAPMAN; JOST; PAS, 2007). Quando *work-sharing* é utilizado em um laço, as iterações do laço são distribuídas entre múltiplos *threads*, de modo que cada iteração do laço seja executada uma única vez, com diferentes posições dos dados de entrada caso hajam mais de um *thread*. Em OpenMP não há uma cláusula para declaração explícita do número de *threads*, o OpenMP determina o número de *threads* a criar, assim como a melhor maneira de sincroniza-los e destruí-los. Para que um laço possa ser paralelizado é necessário obedecer cinco restrições:

- A variável de controle do laço deve ser do tipo `int` ou `unsigned int`, um iterador de acesso aleatório ou um ponteiro;
- A operação de comparação para controle do laço deve ser dos tipos $<$, $<=$, $>$ ou $>=$;
- A parte de incremento do laço deve ser uma adição ou subtração por um valor fixo;

- Se o operador de comparação for $<$ ou $<=$, a variável de controle deve ser incrementada em cada iteração, de maneira análoga caso o operador de comparação seja $>$ ou $>=$, a variável de controle deve ser decrementada em cada iteração;
- Não são permitidos `jumps` para fora do laço, com exceção da declaração `exit` que termina toda a aplicação. Se declarações `goto` ou `break` forem utilizadas, os `jumps` devem ocorrer dentro do próprio laço. Da mesma forma para lançamento de exceções, o tratamento da exceção deve ocorrer dentro do laço.

Embora todas estas restrições possam parecer limitantes, a maioria dos laços podem ser reescritos para atender estas restrições. Mesmo que estas cinco restrições sejam atendidas e o compilador paralelize o laço, o funcionamento pode não ser o esperado caso haja dependência entre os dados. A dependência de dados existe quando diferentes iterações do laço leem ou escrevem em uma mesma posição da memória compartilhada. Esta situação é chamada de condição de corrida.

Condições de corrida são difíceis de detectar, pois, para um determinado caso ou sistema, os *threads* podem executar na ordem que fará com que o programa funcione corretamente. O fato de um programa funcionar uma vez, não significa que ele irá funcionar sobre todas as condições.

Praticamente todos os laços leem e escrevem na memória, é de responsabilidade do programador explicitar para o compilador quais áreas de memória devem ser compartilhadas entre os *threads* e quais devem ser mantidas como cópias privadas. Quando uma área de memória é definida como compartilhada, todos os *threads* podem acessar a mesma área de memória. Quando é definida como privada, uma cópia local é criada para cada *thread* acessar em privado.

Uma região de memória pode ser declarada como privada de duas maneiras:

- Declarando a variável dentro do laço sem a utilização da palavra reservada `static`;
- Especificando quais variáveis serão privadas na cláusula `private`.

Balanceamento de carga é a divisão igualitária de trabalho entre os *threads*. Este é um dos atributos mais importantes para o desempenho de aplicações paralelas. O balanceamento de carga garante que os processadores estarão ocupados senão toda, pelo menos a maior parte do tempo. Sem o balanceamento de carga, alguns *threads* podem terminar antes dos outros, deixando recursos do processador ociosos, desperdiçando oportunidades de ganho de performance.

Podem ocorrer casos onde a primeira metade do laço consuma tanto tempo quanto a segunda metade. Assim como também podem ocorrer casos onde seja impossível definir conjuntos de iterações que tenha um tempo de execução uniforme. Independente do caso, o programador deveria prover informações, fazendo uso da cláusula

`schedule`, ao compilador para melhor distribuir as iterações entre os *threads* para um melhor balanceamento de carga.

Em laços, o balanceamento de carga é afetado pelas variações nos tempos de computação entre as iterações do laço. Normalmente é fácil analisar o código para determinar a variabilidade entre as iterações do laço. Na maioria dos casos, as iterações consomem uma quantidade de tempo uniforme. Quando isto não ocorre, pode ser possível observar um conjunto de iterações que consomem aproximadamente a mesma fatia de tempo.

Se todas as iterações do laço consomem aproximadamente a mesma quantidade de tempo, a cláusula `schedule` deve ser utilizada para distribuir as iterações do laço entre os *threads* em quantidades aproximadamente iguais, de acordo com a politica de escalonamento. É necessário diminuir as chances de conflito de memória que podem surgir utilizando-se grandes tamanhos de *chunks*. Este comportamento pode acontecer pois os laços geralmente acessam a memória sequencialmente, então dividir o laço em grandes tamanhos de *chunks*, como primeira metade e segunda metade utilizando dois *threads*, resultará em menores chances de sobreposição de memória. Enquanto essa pode ser uma melhor escolha em questão de memória, esta escolha pode dificultar o balanceamento de carga. O contrário também é verdadeiro, o melhor para balanceamento de carga irá afetar a utilização de memória, um equilíbrio entre os dois deve ser buscado para otimizar a performance.

Quando um *thread* encontra um ponto de escalonamento de tarefas, ele pode fazer uma troca de contexto, começando a execução de uma nova tarefa, ou retomando a execução de uma outra tarefa que esteja vinculada ao time de tarefas. São pontos de escalonamento de tarefas os localizados nas seguintes situações:

- O ponto imediatamente seguinte à geração explícita de uma tarefa;
- Após a última instrução de uma região de tarefas;
- Em uma região *taskwait*;
- Em regiões de barreiras, implícitas ou explícitas.

Quando um *thread* encontra um ponto de escalonamento de tarefa, ele pode tomar uma das seguintes decisões:

- Começar a execução de uma tarefa vinculada ao time de tarefas;
- Retomar qualquer região de tarefas que tenha sido suspensa, que esteja vinculada ao time de tarefas que a tarefa esteja associada;
- Começar a execução de uma tarefa que não esteja mapeada para nenhum *thread*, que esteja no mesmo time de tarefas.

- Retomar qualquer região de tarefa não mapeada, que esteja vinculado ao time de tarefas.

O pragma `omp taskwait` especifica uma barreira para aguardar a execução de todas as tarefas criadas desde o começo da execução da tarefa atual. Um *taskwait* é associado a região de tarefas atual.

A região *taskwait* inclui um ponto de escalonamento de tarefas implícito na região de tarefas atual. A região de tarefas é suspensa no ponto de escalonamento de tarefas até que a execução de todas as tarefas criadas antes da região *taskwait* sejam terminadas.

A cláusula `schedule` é suportada na construção do laço. Ela é utilizada para controlar a maneira como as iterações do laço serão distribuídas entre os *threads*, podendo alterar a performance do programa. A cláusula `schedule` especifica como as iterações do laço serão mapeadas para os *threads* em um time (OPENMP, 2015). A granularidade desta distribuição de carga de trabalho é o `chunk`, um subconjunto contíguo não vazio do espaço da iteração.

static As iterações são divididas em *chunks* de tamanho do `chunck size`. Estes *chunks* são divididos entre os *threads* estaticamente em uma maneira `round-robin`, de acordo com o número de *threads*. O último *chunk* a ser mapeado pode ter um número menor de iterações. Quando o valor do *chunk size* não for especificado, o espaço de iterações é dividido em *chunk* que sejam aproximadamente iguais em tamanho. Para cada *thread* é associado pelo menos um *chunk*;

dynamic As iterações são mapeadas para os *threads* confirme os *threads* requisitam elas. O thread executa a porção de iterações, de acordo com o parâmetro que define o tamanho do *chunck*, então requere outro *chunk* até que não hajam mais *chunks* para dividir. Quando o tamanho do *chunk* não é especificado o valor padrão é 1;

guided As iterações são associadas aos *threads* conforme são requeridas, o *thread* executa as iterações do *chunk*, então solicita um novo *chunk*, até que não hajam mais *chunks*. Para um tamanho de *chunk* igual a 1, o tamanho de cada *chunk* é proporcional ao número de iterações não associadas a nenhum *thread*, dividido pelo número de *threads*, menos 1. Para tamanhos de *chunks* maiores que 1, o tamanho do *chunk* é determinado do mesmo modo, com a restrição de que os *chunks* não contenham menos iterações que o tamanho do *chunk* informado, com excessão do último *chunk* que pode conter menos iterações que o definido. Quando um tamanho não é definido, por padrão esse valor é igual a 1;

auto A decisão de qual politica de escalonamento a ser escolhida é tomada pelo compilador e/ou em tempo de execução. O programador deixa para a ferramenta a liberdade de escolha de qualquer maneira possível de mapear as iterações para os *threads* em um time;

runtime Se esta opção de escalonamento for escolhida, a decisão é feita em tempo de execução. O tipo de escalonamento e opcionalmente o tamanho do *chunk* são definidos pela variável de ambiente `OMP_SCHEDULE`.

Todos os três tipos de algoritmos de distribuição de carga suportam um parâmetro de tamanho de *chunk* opcional. Como visto na listagem acima, a interpretação deste parâmetro depende do tipo de escalonamento escolhido.

Por exemplo, um *chunk* de tamanho maior que 1 no tipo de escalonamento `static` pode gerar uma politica de escalonamento *round-robin*, onde cada *thread* executa as iterações nas sequencias de *chunks* cujo tamanho foi definido. Nem sempre é fácil selecionar a politica de escalonamento apropriada e o tamanho do *chunk*, a escolha depende não apenas do código do laço, mas também do tamanho do problema e do número de *threads* utilizados.

OpenMP, diferente de Cilk Plus e TBB, oferece ao programador duas estratégias distintas de escalonamento, conforme a estrutura do algoritmo. Uma destas estratégias considera um uma estrutura do tipo *map*, na qual é aplicada uma mesma operação sobre um espaço de dados que pode ser iterado. A estratégia de escalonamento, em qualquer de suas variantes (*static*, *dynamic* ou *guided*), consiste em dividir o espaço de iteração entre tarefas independentes. A segunda estratégia também busca privilegiar estruturas aninhadas-recursivas dos algoritmos, sendo explorada quando o programa implementado faz uso das diretivas *task* e *taskwait*.

2.5 Considerações Sobre o Capítulo

As ferramentas apresentadas neste capítulo oferecem recursos para a criação de tarefas e também recursos para a utilização de padrões concorrentes comumente encontrados na paralelização de problemas reais. Estas ferramentas tornam o escalonamento de tarefas transparente para o usuário, delegando ao usuário apenas as partes de identificação das tarefas e controle das dependências dos dados. No entanto, observa-se que esta transparência na atuação do escalonamento tem como custo a necessidade do programador adaptar o algoritmo de sua aplicação para atender os requisitos de escalonamento apresentados pelo núcleo. Programadores experientes no desenvolvimento de programas paralelos e conhecedores das estratégias de escalonamento podem obter o máximo de desempenho nas suas implementações. O custo, na prática, associada ao uso destas ferramentas é o da adaptação do algo-

ritmo da aplicação a uma estrutura de concorrência que seja adaptada as regras de escalonamento adotadas.

Observa-se que a adaptação do algoritmo à estratégia de escalonamento é uma etapa adicionada ao desenvolvimento de programas. Sendo assim, nesta etapa, o projetista sujeita-se ao risco de inserir novos erros de programação, reduzindo sua robustez. Soma-se a esta observação, o fato de que nem todos os programadores são especialistas nestas ferramentas de programação, tão pouco nas técnicas de escalonamento por elas empregadas. É, portanto, razoável concluir que haverá um maior tempo gasto no desenvolvimento de novas aplicações.

Não estando presente nas ferramentas de programação *multithread* comerciais, o Capítulo 3 busca na literatura ocorrência de trabalhos que permitam definir a estratégia de escalonamento a ser usada em um programa de forma ortogonal ao seu código.

3 TRABALHOS RELACIONADOS

Alguns trabalhos relacionados utilizam uma organização de *software* modular para permitir a alteração de alguns componentes da ferramenta para melhor se adaptarem a arquitetura onde estão sendo executadas. Esses trabalhos mostram que uma única abordagem não obtém o melhor desempenho em todos os casos, justificando a utilização de módulos diferentes em diferentes máquinas. Outros trabalhos apresentados neste capítulo focam na melhoria de algum aspecto dos escalonadores pré-existentes em busca de melhores resultados.

Neste capítulo é descrito as principais contribuições de cada trabalho relacionado, descrevendo brevemente a abordagem utilizada e os resultados obtidos.

3.1 Blaze-Tasks

Blaze-Tasks é um *framework* para escalonamento de tarefas e redução de tarefas em arquiteturas de memória compartilhada (PIRKELBAUER et al., 2019). A implementação do *framework Blaze-Tasks* utiliza técnicas de filas não bloqueantes e programação genérica (meta-programação em C++, através do uso de *templates*) e é totalmente implementado em C++17 garantindo assim a portabilidade de código.

A decomposição recursiva de tarefas em sub-tarefas menores é um cenário comum em programação paralela utilizando o modelo de tarefas. Dependendo do tipo de problema a ser resolvido o número de sub-tarefas criadas pode não ser uniforme, levando a problemas de balanceamento.

Para obter um desempenho ótimo no escalonamento das tarefas, *Blaze-Tasks* utiliza a técnica de *work-stealing* que, como podemos observar nos demais trabalhos relacionados, é a política de escalonamento mais utilizada para o gerenciamento de tarefas em bibliotecas e ferramentas para programação paralela, obtendo o melhor desempenho na grande maioria dos casos.

Para controlar a contenção na fila de tarefas o número de tentativas não-bloqueantes de roubar as tarefas é limitado pela distância entre o *worker*[1] que está

[1] *Worker* em *work-stealing* é como é denominado o *thread* que executa as tarefas.

roubando a tarefa e o *worker* que é o dono da fila de tarefas e o número estimado de vitimas na lista de tarefas. O *work-stealing* é empregado para garantir um balanceamento de carga. Paralelamente a essa abordagem, as novas tarefas criadas são distribuídas entre os *threads* disponíveis em uma forma de *work-sharing*, na expectativa de manter todos os recursos de processamento com disponibilidade de trabalho.

A implementação de *Blaze-Tasks* usa listas não bloqueantes para manter a descrição das tarefas prontas. Cada *thread* mantém sua própria lista de tarefas. Para suportar o *work-stealing*, as listas são implementas como *single-producer* e *multi-consumer*. Estas listas usam uma implementação assimétrica que delega diferentes responsabilidades para o *thread* proprietário da lista e o *thread* que irá roubar as tarefas. Separando estas responsabilidades a implementação utiliza operações atômicas que reduzem a necessidade de sincronização entre os *threads*.

A utilização de programação genérica (*templates*), permite que o alocador e gerenciador de memória seja alterado. O gerenciador de memória padrão do *Blaze-Task* usa *epochs* por padrão. *Epochs* mantém registro de *threads* iniciando ou terminando operações em dados compartilhados. A abordagem utilizada para otimizar o desempenho é o mínimo de *overhead* no caso comum, onde adicionar e remover tarefas das filas tem o mínimo de contenção, levando em consideração que os *threads* terão trabalho suficiente para não precisar roubar tarefas de outros *threads*.

Outro critério adotado para aumentar o desempenho é a localidade de *cache*, o escalonamento leva em consideração a proximidade entre as tarefas para saber de qual *thread* roubar as tarefas.

Os testes apresentados comparando *Blaze-Tasks* com *OpenMP*, *Cilk* e *TBB* mostram que nenhuma das abordagens conseguem obter os melhores resultados em todos os casos de testes. Nos testes realizados no sistema IBM o *Blaze-Tasks* obteve o melhor desempenho na grande maioria dos casos.

As grandes contribuições do trabalho é mostrar que diferentes abordagens podem gerar melhores desempenhos em máquinas diferentes e a possibilidade de mudar o gerenciador de memória, permitindo que o *framework* se adapte melhor a arquitetura que vai ser executada.

3.2 QThreads

O paralelismo de tarefas é a principal funcionalidade da linguagem de programação Chapel, desenvolvida pela Cray (EVANS et al., 2017). A arquitetura do sistema de execução da linguagem Chapel utiliza uma arquitetura modular e provê uma camada de tarefas que possui uma variedade de interfaces para execuções de tarefas como por exemplo, MassiveThreads, QThreads, POSIX Threads e, em sistemas Cray, uma impletação própria da Cray de tarefas *lightweight*.

A camada de tarefas usada por padrão no Chapel é QThreads, que provê abstrações para execução e sincronização de tarefas que mapeia o *hardware* paralelo diretamente.

Este trabalho apresenta um novo escalonador, chamado **distrib**, que combina os melhores aspectos dos outros escalonadores de QThreads, o *work-stealing* e o uso de filas não bloqueantes, para implementar um novo escalonador com melhor desempenho que os predecessores.

O escalonamento em QThreads é cooperativo, quando uma tarefa não consegue obter mais progressos o controle é então passado para outra tarefa que esteja aguardando. Essa transferência de controle é feita por sincronização ou pela chamada explícita de um comando *yield*. A troca de contexto é feita no espaço de usuário, evitando assim a chamada de funções de sistema normalmente mais demoradas.

O escalonador Sherwood foi o primeiro escalonador *work-stealing* para QThreads, desenvolvido para suportar tarefas do OpenMP em QThreads. Sherwood utiliza um balanceador de carga em dois níveis, que combina *work-stealing* e filas compartilhadas LIFO (*Last-In First-Out*), em uma topologia de *threads* próximas. Este arranjo permite o compartilhamento de *cache* e recursos de memória. O escalonamento das tarefas em LIFO faz com que tarefas criadas recentemente sejam executadas primeiro, explorando a localidade. Quando uma fila de tarefas está vazia ela examina as outras filas de tarefas em uma maneira *round-robin*.

O outro escalonador de QThreads é o Nemesis, baseado no subsistema de enfileiramento do MPICH2 Nemesis, este é o modelo de gerenciamento de filas. É utilizado um FIFO (*First-In First-Out*) para as tarefas, projetado para ser otimizado para arquitetura de memória compartilhada.

O novo escalonador proposto neste trabalho é o **distrib**. Ele combina os principais aspectos de cada escalonador, utilizando a eficiência em arquiteturas NUMA (*Non-Uniform Memory Access*) e o balanceamento de carga do Sherwood e a eficiência das listas do Nemesis. A principal contribuição é a reimplementação das filas do Nemesis, adicionando *work-stealing* e mantendo as filas não bloqueantes. Todas as estruturas das filas tem alinhamento de *cache* e as tarefas são escalonadas em LIFO para explorar a localidade de *cache*.

A avaliação dos resultados é feita comparando o escalonador **distrib** com o Nemesis e o Sherwood para analisar os desempenhos e verificar os efeitos do uso de *work-stealing* e filas não-bloqueantes separadamente. Os escalonadores **distrib** e Nemesis se mostraram mais eficientes em arquiteturas *manycore*. O escalonador Sherwood obteve maior desempenho em aplicações em que o acesso aos dados é regular em contraste com o **distrib** e o Nemesis que obtiveram maior desempenho em aplicações com acesso irregular.

3.3 Almost Deterministic Work Stealing

Este trabalho propõe um algoritmo de *work-stealing* quase determinístico (SHIINA; TAURA, 2019). O processo consiste em duas partes a alocação determinística das tarefas e o *work-stealing* hierárquico. A alocação de tarefas para determinados processadores exige o conhecimento prévio da quantidade de trabalho que cada tarefa gera e executa. Esse tipo de escalonador é chamado *offline*. Esse tipo de abordagem é difícil, e a alternativa utilizada é o uso de anotações no código para indicar uma estimativa de trabalho que a tarefa irá gerar. É difícil essa previsão, mas o balanceador de carga dinâmico corrige eventuais problemas na distribuição de carga.

Work-stealing tem uma boa localidade em memória compartilhada, porém em arquiteturas NUMA essa localidade tende a não existir no roubo de tarefas e o desempenho acaba não sendo o mesmo comparado a outras arquiteturas.

A ordem de execução da tarefas nessa política de escalonamento é praticamente a mesma da versão sequencial, por isso o algoritmo precisa ser escrito para ser eficiente na versão sequencial. Dessa forma os *workers* acessam os mesmos dados para iterações diferentes, explorando assim a localidade.

Essa forma de divisão de tarefas de forma quase determinística entre nodos de máquinas NUMA mostram resultados melhores em quase todos os caso de testes, comparados com a política de escalonamento *dynamic* do OpenMP e de *work-stealing* com roubo de tarefas de forma aleatório.

3.4 Tuning the victim selection policy of Intel TBB

Os experimentos deste trabalho mostram que o *overhead* da paralelização pode gerar *speedup* sub-lineares e aumentar o consumo de energia para aplicações paralelas (IORDAN; JAHRE; NATVIG, 2015).

Este trabalho propõe melhorias na seleção da vítima no *work-stealing* para mitigar a contenção de recursos de *hardware* por parte do *thread*.

Uma forma é pela seleção pseudo-aleatória da vítima, na primeira tentativa é feita uma busca aleatória, caso a busca não tenha resultados é feita uma busca em todas as listas de tarefas excluindo a lista da primeira tentativa antes que retorne para a busca aleatória novamente. Outro método de seleção da vítima pela taxa de ocupação, esse método procura uma vítima de uma das listas que tenha a maior taxa de ocupação, ou seja o maior número de tarefas.

O método pseudo-aleatório não escala bem, causando um aumento exponencial no *overhead*. Já no método de seleção pela taxa da ocupação a busca pela melhor vítima gera um *overhead* em cada roubo de tarefas, porém em alguns casos mostrou um tempo de execução menor e um menor consumo de energia de acordo com os resultados apresentados.

3.5 Considerações Sobre o Capítulo

Os trabalhos relacionados convergem para um ponto em comum, a necessidade de políticas de escalonamento que atenda o tipo de algoritmo a ser executado ou mesmo a arquitetura onde as aplicações serão executadas. Como mostram os resultados destes trabalhos, embora seja consenso que *work-stealing* obtenha na grande maioria dos casos um desempenho melhor se comparado com outras abordagens, casos específicos necessitam de alterações no próprio *work-stealing* ou até mesmo um algoritmo novo para extrair o máximo de desempenho.

4 FERRAMENTAL CONCEITUAL

Neste trabalho o principal objeto de estudo está associado à execução eficiente de tarefas concorrentes geradas em laços paralelizáveis. O estudo de caso é conduzido avaliando alternativas de escalonamento aplicáveis a algoritmos iterativos que reflitam o modelo de paralelismo espresso pelo padrão *map*. Neste capítulo são apresentados conceitos associados ao ferramental utilizado para o desenvolvimento do caso de estudo.

4.1 C++ Thread

A linguagem C++ em sua origem não oferecia suporte nativo para a programação concorrente. Com o crescimento dos *hardwares* paralelos e a necessidade por sistemas e aplicações que explorassem esses *hardwares* paralelos de maneira eficiente os programadores faziam uso de recursos de terceiros para implementar esse paralelismo. O que era utilizado era *PThreads* (THE OPEN GROUP, 2018), sua primeira especificação é de 1995, uma época onde nem a orientação a objetos nem os *hardwares* paralelos era amplamente utilizados, dessa forma muito dos recursos oferecidos pelo *PThreads* foram planejados focando a linguagem C, devido a sua compatibilidade com a linguagem C, C++ automaticamente suporta *PThreads*. Como C++ é uma linguagem orientada a objetos e *PThreads* foi planejado com o paradigma estruturado, os programadores não conseguiam explorar todo o potencial da orientação a objetos ao utilizar *PThreads*, como por exemplo o encapsulamento e a checagem de tipos. Essa necessidade foi observada e foi desenvolvida uma ferramenta, chamada *Boost* (BOOST, 2019) que é um conjunto de bibliotecas para estender a biblioteca padrão do C++. Entre os recursos oferecido pela biblioteca podemos destacar os *threads*, mecanismos de sincronização e *future*/*promises*. Essa biblioteca fornece uma forma de descrever a concorrência nas aplicações utilizando a orientação a objetos, porém ainda é necessário a utilização de bibliotecas de terceiros, e com isso dificultando a portabilidade e reuso do código.

Essa crescente necessidade por uma interface de descrição da concorrência foi

explorada nas especificações do C++11, onde verificou-se que a linguagem tinha uma lacuna em diversos itens, sendo a concorrência um desses itens. Muitos dos conceitos e abordagens utilizados pela biblioteca *Boost* foram adotados na especificação dos *threads* em C++11 (ISO/IEC, 2008), facilitando o uso de *threads* em C++ sem a necessidade da utilização destas bibliotecas, garantindo assim o reuso e a portabilidade dos códigos em C++.

Na especificação do C++11, pela primeira vez em 13 anos desde a primeira especificação de C++, são apresentados recursos para programação *multithread*, dentre os quais a criação de *threads* (`std::thread`), tarefas (`std::packaged_task`) e formas de sincronização (`std::thread::join`, `std::mutex`, `std::future`, `std::promise`). Esses recursos permitem que a concorrência da aplicação seja descrita sem a utilização de extensões, muitas vezes específicas de uma determinada plataforma, permitindo reuso e portabilidade dos códigos com a garantia de que o comportamento será o mesmo em todas as plataformas.

Portabilidade de código é a capacidade de aproveitar um código desenvolvido para um programa em outro. Eventualmente uma linguagem é portável, como C++, por estar amplamente disponível em várias plataformas. Portabilidade de desempenho é quando um programa pode ser executado em diferentes plataformas (com diferentes numero de processadores) e o desempenho escala de acordo com o aumento do número de processadores. Esta portabilidade de desempenho é obtida com o uso de tarefas. Como C++17 não oferece recursos para o escalonamento de tarefas dizemos que não possui portabilidade de desempenho.

O uso de *threads* pode não fornecer o ganho esperado, pois existe um custo inerente a criação dos *threads*. O sistema operacional precisa alocar recursos como a pilha de memória do *thread*. Caso a tarefa a ser executada pelo *thread* seja muito simples o custo de criar um *thread* pode ofuscar o tempo ganho pela divisão do problema em tarefas, levando o tempo total a ser maior que a execução sequencial por um único *thread*.

Além dos problemas relacionados a criação de *threads* temos a questão de limites de recursos, especialmente em sistemas 32-*bit*. A criação de muitos *threads* pode fazer com que o sistema como um todo fique muito lento, consumindo toda a memória, já que cada *thread* precisa de seu próprio espaço de memória. Um exemplo em uma máquina 32-*bit* onde cada *thread* tenha uma pilha de 1MB, o endereçamento total de memória nesse sistema é limitado a 4GB, levando a um máximo de 4096 *threads*, considerando apenas a memória alocada para a pilha. Embora em um sistema 64-*bit* não haja estes problemas, os recursos ainda são finitos, podendo levar a um consumo exagerado de recursos.

4.2 Tarefas

O conceito de tarefas tem sido adotado por praticamente todas as bibliotecas e ferramentas para solucionar problemas paralelos. A divisão do problema em pequenas tarefas facilita a solução do problema, uma vez que o programador deixa questões arquiteturais de lado e foca na solução do problema. Atualmente a especificação de C++ não tem um sistema de gerenciamento de tarefas, embora em sua especificação tenha de forma experimental a implementação de alguns algoritmos paralelos. O que os compiladores fazem na prática é a utilização de alguma biblioteca paralela para implementar esses algoritmos. No caso compilador GCC temos a utilização de OpenMP (FREE SOFTWARE FOUNDATION, 2020) como gerenciador de tarefas e no caso do ICPC é utilizado o TBB (INTEL, 2018b).

Embora ambas as abordagens obtenham bons resultados na grande maioria dos casos, temos alguns casos específicos onde elas não tem um desempenho ótimo. Alguns trabalhos relacionados discutidos no Capítulo 3 mostram como para determinados tipos de algoritmos a utilização de uma política de escalonamento direcionada ao problema obtém resultados melhores que as bibliotecas TBB e OpenMP.

Tarefas não sofrem dos mesmos problemas dos *threads*, não é necessário a alocação de recursos do sistema operacional para a criação de tarefas. Em um nível mais abstrato, tarefas normalmente são criadas é inseridas em uma lista de um *thread pool*, onde de acordo com alguma política de escalonamento serão executadas.

Esse modelo de paralelismo consome menos recursos e tem se mostrado mais eficiente, uma vez que as políticas de escalonamento conseguem manter os processadores ocupados na maior parte do tempo, garantindo um melhor balanceamento da carga, consequentemente gerando num ganho de desempenho em relação ao tempo de execução.

Embora C++ ofereça recursos para a criação e execução de tarefas, a especificação não detalha um sistema de gerenciamento destas tarefas, ficando a cargo do programador definir e implementar a politica de escalonamento das tarefas a ser empregada.

Nas novas especificações do C++ foi introduzida um forma de encapsularmos uma tarefa para a execução futura. O método `packaged_task` permite que qualquer invocável da linguagem C++ seja encapsulado e utilizando o método `bind` podemos atribuir parâmetros para essa tarefa.

O escalonamento das tarefas nas bibliotecas disponíveis comercialmente no momento é executado de forma transparente para o programador. As bibliotecas utilizam principalmente dois tipos de politicas de escalonamento, o *work-stealing*, adotado por TBB e o *work-sharing* adotado pelo OpenMP. Ambas as bibliotecas não oferecem uma interface para a troca dessa política de escalonamento. Essa limitação na troca da po-

lítica de escalonamento faz com que o programador tenha que adaptar a solução do problema para a biblioteca a ser utilizada, diminuindo a expressividade e o reuso de código.

4.3 Estudo de caso: map

Laços são um dos conceitos mais utilizados para expressar o paralelismo em linguagens e bibliotecas paralelas (WISMÜLLER, 2011). Existem vários tipos de laços, um particularmente especial em programação paralela é o *map*, onde todas as iterações são completamente independentes.

Duas formas principais de expressar o paralelismo é especificar diferentes regiões de código que devem ser executadas em paralelo, como ocorre na definição de tarefas. Esta forma oferece uma baixa escalabilidade uma vez que o grau paralelismo é determinado pelo número de regiões paralelas. A outra forma é o uso de laços paralelos, nesse tipo de paralelismo a mesma região de código é executada com diferentes dados SPMD (FLYNN, 2011; DAREMA, 2011).

Laços paralelos são tipicamente utilizados em modelos de programação de memória compartilhada, como OpenMP e TBB.

No OpenMP o laço é paralelizado pela diretiva `#pragma omp parallel for` e tem algumas restrições. É necessário que seja um laço do tipo `for`, com uma variável de iteração do tipo `int`, um incremento constante e um teste de terminação simples. O laço não pode terminar com um `break` ou `return` ou exceção.

No TBB o laço é dividido recursivamente, dividindo o *range* em *subranges*, onde cada subdivisão é uma tarefa, que é escalonada dinamicamente para um dos *threads* (chamados de *workers* em TBB) disponíveis. O número de *workers* é normalmente menor que o número de *cores*, reservando um *core* para o *thread* que gerencia o escalonamento.

Devido a importância de laços em problemas paralelos, nosso estudo de caso escolhido foi o padrão paralelo *map*. Duas formas de se dividir um laço em tarefas é dividir recursivamente o laço em laço menores, até um ponto de controle, normalmente este ponto de controle é definido como um número mínimo de iterações. Em TBB o laço é dividido na metade, o *thread* atual executa a metade do laço e a outra metade é criada como uma tarefa, eventualmente novas tarefas serão criadas a partir dessas subdivisões. A outra forma de dividir os laços em tarefas é dividir o conjunto de dados do laço em pedaços e criar tarefas para cada divisão. Estas tarefas então são executadas de acordo com alguma política de execução.

4.4 Escalonamento de tarefas map

O OpenMP quebra em *chunks*. Os *chunks* são “fatias” do laço e correspondem às tarefas a serem executas. O programador define o tamanho (em iterações) dos *chunks*, podendo considerar o número total de iterações e CPUs disponíveis. O programador também define o tipo de escalonamento: estático (adequado aos casos em que o custo computacional das iterações não varia), dinâmico (adequado quando o custo computacional das iterações varia, havendo uma lista de tarefas compartilhada sendo consumida pelos *threads* - *overhead* de acesso à lista), tem também o *guided* (semelhante ao *dynamic*, mas é usado associado a cláusula *nowait*, que permite que um *thread* inicie a execução antes das outras. Assim, a que chegou primeiro, pega um *chunk* “maior” (em número de iterações), a segunda pega um *chunk* menor e assim por diante.

O escalonamento de TBB, assim como o de Cilk Plus, divide de forma recursiva: divide no meio, deixa a primeira metade em espera e executa a segunda. Isso ele faz de forma recursiva. Eventualmente, por *work-stealing*, a primeira metade do laço pode ser roubada e executada em outra CPU: da mesma forma, ele divide e vai executar a segunda metade e gerando subdivisões do laço.

4.5 Considerações Sobre o Capítulo

As estratégias de escalonamento de OpenMP e TBB são fixas na ferramenta, havendo pouca possibilidade de adequação da estratégia de escalonamento em função da necessidade do algoritmo. Seria interessante poder separar a especificação da concorrência da aplicação do mecanismo de escalonamento. Essa separação permite que o problema seja visto sob novas perspectivas, permitindo que a solução não fique dependente da ferramenta utilizada para explorar o paralelismo.

5 IMPLEMENTAÇÃO

Existem diversos tipos de algoritmos iterativos: *map*, *reduce*, *stencil*, *scan*, entre outros. Podendo, estes, ter tarefas com custos homogêneos ou não. A execução eficiente destes diferentes algoritmos depende de como distribuí-los sobre os recursos de processamento. Necessidades distintas de escalonamento, portanto, são necessárias. Neste capítulo é apresentado um estudo de caso sobre o escalonamento de algoritmos iterativos, aplicado ao *map*. Neste estudo de caso, sobre a linguagem C++17, é construída uma interface de programação, sobre a API (*Application Programming Interface*) de *threads* nela disponibilizada, que permite separar a especificação da concorrência do mecanismo de escalonamento.

O *map* pela sua definição não tem dependência de dados, ou seja, todas suas operações podem ser executadas concorrentemente sem a utilização de bloqueios. Na abordagem do TBB é utilizado o método recursivo-aninhado, onde é gerado um grafo acíclico direcionado (DAG), dessa forma a política de escalonamento do TBB garante que uma tarefa que possui dependências em outra tarefa não seja executada antes que a tarefa com as dependências seja concluída. Garantindo assim uma execução *lock-free*. O OpenMP por sua vez utiliza os conceitos de váriaveis *private* e *shared* nas anotações do código, criando cópias locais das variáveis definidas como privadas e por padrão todas as variáveis que não forem especificadas são definidas como compartilhadas. Caso exista alguma variável global que precise ser escrita por mais de um *thread* existe um mecanismo de sincronização dos *threads*, o código especificado no bloco `#pragma omp critical` irá executar sequencialmente.

Na implementação do algoritmo *map* assumimos que o algoritmo não tem dependência de dados, porém, caso seja necessária a sincronização de algum trecho de código é possível a utilização dos recursos da linguagem C++ como por exemplo o `mutex`, pois a ferramenta é totalmente compatível com a API de *threads* de C++17.

5.1 Interface de Programação

No Algoritmo 1 temos um exemplo de utilização da interface. O objetivo da implementação é a transparência para o programador, dessa forma utilizamos a inferência dos tipos dos templates, para que o programador não precise se preocupar com a tipagem na hora de utilizar a interface. A interface por ser próxima a C++ é facilmente adaptada para soluções já existentes.

```
1 algorithms::map(❶[&](const int &begin, const int &end) {
2     for (int i = begin; i < end; i++) {
3         ...
4     }
5 },❷ 0,❸ vec.size(),❹ new scheduler());
```

Algoritmo 1 – Exemplo de utilização da interface de programação, o padrão *map* recebe quatro parâmetros: (1) a função lambda contendo o corpo do laço; (2) a posição inicial da coleção de dados a ser iterada; (3) a posição final da coleção de dados a ser iterada; (4) uma instância opcional do escalonador a ser utilizado.

O uso de funções lambdas de C++ facilita o uso da concorrência, a assinatura do algoritmo *map* espera como parâmetros uma função com retorno *void* e que recebe três parâmetros o início e fim do vetor a ser iterado e uma instância opcional do algoritmo de escalonamento.

Variáveis externas ao bloco paralelo não precisam ser passadas como parâmetros, já que é possível especificar na função lambda a forma de capturar as variáveis, no caso do exemplo acima as variáveis foram capturadas por referência, permitindo que elas sejam modificadas dentro do corpo do algoritmo *map*.

5.2 Descrição da Concorrência

No Algoritmo 2 temos a implementação do padrão paralelo *map*, primeiramente uma instância do escalonador é obtida para que as tarefas sejam repassadas para que ele faça o escalonamento das tarefas. O tamanho do *chunk* é definido pelo tamanho do *container* a ser iterado dividido pelo número de núcleos do processador da máquina onde está sendo executado. O número de núcleos é obtido pelo método `thread::hardware_concurrency()`, permitindo que a implementação se adapte aos diversos tipos de hardware.

Uma vez que o tamanho dos *chunks* é definido são criadas tarefas usando o `packaged_task`, amarrando a função a ser executada como os parâmetros passados, pelo método `bind`. As tarefas criadas são então passadas para o escalonador para aplicar a heurística de escalonamento. O *join* do escalonador é um ponto de sincronização, um barreira, onde a implementação do *map* aguarda a execução de todas as tarefas para continuar a execução da aplicação.

```
using namespace std;

template<typename TF, typename TB, typename TE>
static void map(TF f, TB b, TE e, scheduler *sch = nullptr)
{
  if(sch == nullptr) {
    sch = new scheduler();
  }
  int cs = std::max((e - b) / thread::hardware_concurrency(), 1);

  for (auto i = b; i < e; i = i + cs) {
    packaged_task<void()> _task(bind(f, i, std::min(e, i + cs)));
    sch->detach(_task);
  }
  sch->join();
}
```

Algoritmo 2 – Implementação do padrão paralelo *map*, caso não seja passado o escalonador é utilizado um escalonador padrão. No trecho de código 'f' é a função aplicada, 'b' e 'e' são o ínicio e fim respectivamente e 'cs' é o *chunk size*.

5.3 Especificação do Mecanismo de Escalonamento

Para garantir que houvesse apenas uma única instância do escalonador foi utilizado um *singleton* no método de criação do escalonador como podemos ver no Algoritmo 3.

```
static scheduler *getInstance() {
  static std::mutex instance_mutex;
  volatile static scheduler *_instance;

  if(_instance == nullptr){
    instance_mutex.lock();
    if(_instance == nullptr) {
      _instance = new scheduler();
    }
    instance_mutex.unlock();
  }
  return const_cast<scheduler *>(_instance);
}
```

Algoritmo 3 – Uso de *singleton* para garantir que a instância do escalonador será a mesma sempre que uma instância for requisitada.

Um problema na implementação de um *singleton* (GAMMA et al., 1995) em uma aplicação paralela é a sincronização do acesso ao construtor. Uma abordagem para solucionar esse problema é o *double-checked locking* (MEYERS; ALEXANDRESCU,

2004) do *singleton*. Nessa abordagem é feito primeiramente uma verificação se a instância é nula e ai então é adquirido o *lock*, verificado novamente se a instância é nula e então é criado uma nova instância. Esta solução garante que a instância do escalonador só seja bloqueante no primeiro acesso na hora da criação, e caso alguma outro *thread* tenha criado a instância antes que o *thread* atual adquira o *lock* é feita uma nova checagem se a instância é nula. Esta implementação garante que a instância será única e que não haverá *locks* desnecessários.

```
std::vector<task> task_queue;
std::vector<std::thread> workers;
scheduler() {
  terminated = false;
  max_threads = std::thread::hardware_concurrency();
  for (unsigned int i = 0; i < this->max_threads; i++) {
    std::thread thr(
    [this]() {
      while (true) {
        this->task_queue_mutex.lock();
        if (!this->task_queue.empty()) {
          auto task = std::move(this->task_queue.back());
          this->task_queue.pop_back();
          this->task_queue_mutex.unlock();
          task();
        } else if (terminated) {
          this->task_queue_mutex.unlock();
          break;
        } else {
          std::this_thread::yield();
          this->task_queue_mutex.unlock();
        }
      }
    });
    workers.push_back(std::move(thr));
  }
}
```

Algoritmo 4 – Descrição dos *workers*, o *thread* consome as tarefas da lista de tarefas compartilhadas até que a lista esteja vazia, terminando o *thread* caso a execução seja marcada como terminada ou colocando-se no fim da lista de prioridades caso não tenha nenhuma tarefa para executar no momento.

O algoritmo do escalonador é exibido no Algoritmo 4, inicialmente é criado um número de *threads*, chamados *workers*, igual ao número de núcleos do processador. O escalonador possuí uma lista de tarefas e os *threads* consomem as tarefas dessa lista. O acesso a lista é sincronizado para que cada *thread* execute as tarefas uma única vez.

No caso de uma thread não conseguir uma tarefa para executar ela sinaliza que ela não tem nada para executar e vai para o fim da fila de prioridades chamando o método `this_thread::yeld()`. Esse controle evita que um *thread* fique tentando acesso a lista enquanto ainda não há tarefas para serem consumidas.

Essa abordagem é semelhante a utilizada em OpenMP no tipo de escalonamento *dynamic*. Os *threads* vão solicitando novas tarefas para executarem conforme as tarefas vão sendo completadas. Eventualmente as tarefas acabam e o escalonador é sinalizado que os *threads* podem ser sincronizados.

Para mover as tarefas para a lista de tarefas o escalonador adquire o *lock* da lista de tarefas e adiciona a tarefa na lista como mostrado no Algoritmo 5, o detalhe a ser observado aqui é a utilização do `std::move` que move o objeto, para que não exista duas referências para o mesmo objeto, garantindo assim que a tarefa seja executada uma única vez.

```
void detach(std::packaged_task<void()> &_task) {
    this->task_queue_mutex.lock();
    task_queue.push_back(std::move(task(_task)));
    this->task_queue_mutex.unlock();
}
```

Algoritmo 5 – Criação das tarefas nas listas de tarefas dos *workers*, o método *detach* recebe uma tarefa e a implementação define qual lista de tarefas receberá a tarefa.

E por fim é feita a sincronização dos *threads*. No Algoritmo 6 é mostrado a sincronização dos *threads*. Após a criação de todas as tarefas e elas terem sido passadas para a lista de tarefas o `join` informa que a criação de tarefas foi finalizada definindo do parâmetro *terminated* como *true*.

```
void join() {
    terminated = true;
    for (auto &work : this->workers) {
        if (work.joinable()) {
            work.join();
        }
    }
}
```

Algoritmo 6 – Mecanismo de sincronização das tarefas, a execução é bloqueada até que todas os *threads* sejam sincronizados.

Quando um *thread* busca por uma tarefa e a lista está vazia ele verifica se a criação de tarefas foi terminada, caso o valor seja *true* o *thread* é terminado pela chamada do `break` e esse *thread* pode então ser sincronizado pelo `join`. Isso garante que todos os *threads* tenham terminado antes da continuação da execução da aplicação.

5.4 Implementação de Novas Heurísticas de Escalonamento

Para a implementação de uma nova heurística de escalonamento é necessário que a nova classe herde a interface `scheduler`, essa interface contém três métodos que são necessários para o novo escalonador.

No construtor é definido o número de *workers* a serem criados e também o tipo de algoritmo a ser implementado para a execução das tarefas que forem criadas. As estruturas de dados para armazenamento dos *workers* é de escolha do programador, podendo escolher diferentes estruturas e até mesmo a implementação de novas estruturas caso seja necessário o armazenamento de dados que auxiliem o *worker* nas tomadas de decisões.

O método `detach` recebe as tarefas encapsuladas pelo `packaged_task` e distribui entre as filas de tarefas dos *workers*. Nas listas de tarefas o programador também pode escolher o tipo de estrutura de dados que melhor se adapte a implementação, podendo inclusive caso seja necessário para otimizar o desempenho, a implementação de listas não bloqueantes, para diminuir a contenção de dados.

O outro método que deve ser implementado para o novo escalonador é o `join`. O `join` serve como uma barreira de sincronização dos *threads*.

```
1 class scheduler {
2 public:
3   scheduler(); ←❶
4
5   virtual void detach(std::packaged_task<void()> &_task); ←❷
6
7   virtual void join(); ←❸
8 };
```

Algoritmo 7 – Interface da classe scheduler. Contém os métodos básicos para a criação de um novo escalonador: (1) o método construtor, onde é definido o algoritmo executado por cada *worker*; (2) o método detach, responsável por alocar as tarefas nas listas de tarefas dos *workers*; (3) o método join, ponto de sincronização dos *workers*.

O novo algoritmo de escalonamento pode então ser passado para o padrão implementado através do parâmetro opcional *sch*, como exemplificado no Algoritmo 8.

```
1 template<typename TFunc, typename TBegin, typename TEnd>
2 static void map(TFunc f, TBegin b, TEnd e, scheduler *sch = nullptr)
```

Algoritmo 8 – Assinatura do padrão parelelo *map*, uma instância de um novo escalonador pode ser passado pelo parâmetro opcional *sch*.

5.5 Exemplos de Utilização dos Padrões Paralelos

Uma ferramenta para ser amplamente adotada, precisa além do desempenho oferecer uma interface simples e uma forma de paralelizar algoritmos sequencias que seja simples sem a necessidade de alterar o código para se adaptar a ferramenta.

Nesta seção apresentamos alguns exemplos de códigos utilizando a ferramenta, comparando a versão sequencial e a sua forma paralela utilizando a biblioteca.

No Algoritmo 9 temos a função que cálcula as iterações do mandelbrot, utilizando um vetor unidimensional é feita as devidas transformações para coordenadas *x* e *y*;

```
using cmplx = complex<double>;

static auto mandelbrot_iterations(cmplx c) {
    cmplx z{};
    size_t iterations{0};
    const size_t max_iterations{1000};
    while (abs(z) < 2 && iterations < max_iterations) {
        ++iterations;
        z = pow(z, 2) + c;
    }
    return iterations;
}

auto i_to_xy([=](int i) { return scale(i % w, i / w); });

auto to_iteration_count([=](int i) {
    return mandelbrot_iterations(i_to_xy(i));
});
```

Algoritmo 9 – Trecho de código da implementação do algoritmo de Mandelbrot

Na versão sequencial apresentada no Algoritmo 10 é criado um vetor representando o plano de Mandelbrot com dimensões *w* (largura) e *h* (altura) e o método `transform` é aplicado. Este método tem o funcionamento semelhante ao *map* sequencial, ele aplica uma função em todos os elementos do vetor.

```
for (int j = 0; j < 1; j++) {
    vector<int> v(w * h);
    iota(begin(v), end(v), 0);

    transform(begin(v), end(v), begin(v), to_iteration_count);
}
```

Algoritmo 10 – Trecho de código da versão sequencial do algotimo de Mandelbrot.

Para paralelizar este código decompomos o algoritmo `transform` em um laço *for* e aplicamos o método em cada elemento do vetor como é possível observar no Algoritmo 11.

```
for (int j = 0; j < 1; j++) {
  vector<int> v(w * h);
  iota(begin(v), end(v), 0);

  algorithms::map([&](const int &begin, const int &end) {
    for (int i = begin; i < end; i++) {
      v[i] = to_iteration_count(v[i]);
    }
  }, 0, v.size());
}
```

Algoritmo 11 – Trecho de código da implementação da versão paralela do algoritmo de Mandelbrot.

Como a interface não exige que seja passado o *container* que será iterado, podemos usar tanto um *container* da STL (*Standard Template Library*), como também vetores *C-like*, neste outro exemplo mostramos o Método de Romberg, um método para estimar uma integral no intervalo *a*, *b*, como parâmetro é passado uma função representando a integral e os intervalos *a* e *b*. Um excerto do método em sua versão sequencial é mostrado no Algoritmo 12.

```
double romberg(double (*func)(double), double a, double b) {
  double h[N+1];
  double r[N+1][N+1];

  for (int i = 2; i < N + 1; ++i) {
    double coeff = 0;

    for (int k = 1; k <= pow(2, i - 2); ++k) {
      coeff += func(a + (2 * k - 1) * h[i]);
    }

    r[i][1] = 0.5 * (r[i - 1][1] + h[i - 1] * coeff);
  }

}
```

Algoritmo 12 – Trecho de código da implementação sequencial do método de Romberg.

No Método de Romberg precisamos iterar a variável de controle de *2* até *N + 1*, a interface nos permite controlar o laço pelos dois últimos parâmetros passados para o algoritmo *map*. Dessa forma garantimos o correto funcionamento da versão paralela. Novamente, com poucas alterações no algoritmo original podemos paralelizar o método, independente do tipo de estrutura de dados utilizada bem como os tipos de invocáveis. No Algoritmo 13 temos o trecho de código onde foi aplicado o padrão paralelo *map* no Método de Romberg.

```
double h[N+1];
double r[N+1][N+1];

algorithms::map(
  [&](const int &begin, const int &end) {
    for (int i = begin; i < end; ++i) {
      double coeff = 0;

      for (int k = 1; k <= pow(2, i - 2); ++k) {
        coeff += func(a + (2 * k - 1) * h[i]);
      }

      r[i][1] = 0.5 * (r[i - 1][1] + h[i - 1] * coeff);
    }
  }, 2, N + 1
);
```

Algoritmo 13 – Trecho de código da implementação paralela do método de Romberg.

É possível adaptar algoritmos sequenciais, mesmo aqueles que utilizam construções mais abstratas da linguagem, com poucas alterações no código. Utilizando alguma política de escalonamento do tipo *work-stealing* é possível obter resultados com ótimos desempenho, garantindo o correto funcionamento da aplicação.

5.6 Implementação de Work-Stealing

Nesta seção é descrita uma implementação de um escalonador work-stealing adaptado do livro (WILLIAMS, 2012).

```
typedef function_wrapper task_type; ←❶
std::atomic_bool done;
thread_safe_queue<task_type> pool_work_queue; ←❷
std::vector<std::unique_ptr<work_stealing_queue>> queues; ←❸
std::vector<std::thread> threads;
inline static thread_local work_stealing_queue *local_work_queue; ←❹
inline static thread_local unsigned my_index;
```

Algoritmo 14 – Atributos privados da implementação do escalonador work-stealing. É utilizado um wrapper para os invocáveis e listas de tarefas locais.

No Algoritmo 14 temos um trecho de código com os atributos privados da classe que implementa o algoritmo de *work-stealing*. Destacamos aqui quatro atributos: (1) Uma classe implementando um *wrapper* para os invocáveis de C/C++; (2) Uma fila *thread-safe* para o *pool* de *threads*; (3) Um vetor de filas de tarefas usado para o roubo de trabalho; (4) O especificador `thread_local` que especifica que o atributo será estático porém existirá um cópia privada por *thread*, estas filas são privadas à cada *thread*.

```
1  void worker_thread(unsigned my_index_) { ←❶
2    my_index = my_index_;
3    local_work_queue = queues[my_index].get();
4    while (!done) {
5      run_pending_task();
6    }
7  }
8
9  bool pop_task_from_local_queue(task_type &task) { ←❷
10   return local_work_queue && local_work_queue->try_pop(task);
11 }
12
13 bool pop_task_from_pool_queue(task_type &task) { ←❸
14   return pool_work_queue.try_pop(task);
15 }
16
17 bool pop_task_from_other_thread_queue(task_type &task) { ←❹
18   for (unsigned i = 0; i < queues.size(); ++i) {
19     unsigned const index = (my_index + i + 1) % queues.size();
20     if (queues[index]->try_steal(task)) {
21       return true;
22     }
23   }
24   return false;
25 }
```

Algoritmo 15 – Métodos privados da classe que implementa o *work-stealing*.

No Algoritmo 15 é listado os métodos privados da classe *work-stealing*. (1) Este método é o *kernel* de cada *thread*, ele busca na fila de tarefas local uma tarefa pendente para executar. Cada *worker* recebe um índice único para indexar as filas locais e as filas de tarefas para roubo de trabalho. Pelo índice o *worker* obtém um ponteiro para a fila e aponta para a fila local. Enquanto a execução não for marcada como terminada pelo atributo `done` o *worker* tenta rodar tarefas pendentes; (2) É o método que verifica se tem tarefas na fila local e retira uma da fila para execução. Internamente o método adquire o *lock* da fila e verifica se a mesma não está vazia e remove um elemento da início da fila; (3) Método para pegar uma tarefa do *pool* de tarefas, funcionamento análogo ao método anterior; (4) Este método faz uma busca nas filas dos outros *workers* em busca de trabalho para roubar. Para evitar que todos os *workers* tentem roubar tarefas da primeira fila é adicionado um *offset* a partir do índice do *worker* que está tentando roubar tarefas. O método `try_steal` internamente adquire o *lock* da fila de tarefas e verifica se a mesma não está vazia. Caso hajam tarefas ele então remove a tarefa do fim da fila de tarefas, na expectativa de que essa tarefa como sendo gerada a mais tempo contenha mais trabalho e mantenha o *worker* ocupado por mais tempo.

```
1  thread_pool_work_stealing() ←❶
2    : done(false) {
3    unsigned const thread_count = std::thread::hardware_concurrency(); ←❷
4    try {
5      for (unsigned i = 0; i < thread_count; ++i) {
6        queues.push_back( ←❸
7          std::unique_ptr<work_stealing_queue>(
8            new work_stealing_queue
9          )
10       );
11       threads.push_back(std::thread( ←❹
12         &thread_pool_work_stealing::worker_thread, this, i)
13       );
14     }
15   }
16   catch (...) {
17     done = true; ←❺
18     throw;
19   }
20 }
21
22 ~thread_pool_work_stealing() { ←❻
23   done = true;
24 }
```

Algoritmo 16 – Construtor e destrutor da classe que implementa o *work-stealing*.

No Algoritmo 16 temos as implementações do construtor e destrutor da classe *work-stealing*. (1) O construtor é responsável pela inicialização das filas de tarefas e dos *workers*. O parâmetro *done* é passado como *false* na lista de inicialização do construtor, para sinalizar que a execução não está finalizada e garantir que os *workers* irão consumir tarefas das filas; (2) O número de *workers* é definido pelo método `thread::hardware_concurrency`, que normalmente é o número de *cores* da máquina onde está executando; (3) Para cada *worker* é criado uma fila de tarefas e é inserida nas listas de filas do *pool*; (4) De forma análoga ao item anterior são criados os *threads* representando os *workers* e atribuído a eles o método `worker_thread`, passando o *this* e o índice como parâmetro. É necessário passar o *this* para que o *thread* tenha acesso ao objeto e possa acessar os atributos e métodos da classe; (5) Caso ocorra qualquer erro na criação dos *workers* ou das filas de tarefas a execução é marcada como terminada e é lançada uma exceção; (6) O destrutor marca a execução como terminada setando o atributo `done` como *true*.

```
template<typename FunctionType>
std::future<typename std::result_of<FunctionType()>::type>
detach(FunctionType f) {
  typedef typename std::result_of<FunctionType()>::type result_type;
  std::packaged_task<result_type()> task(f);
  std::future<result_type> res(task.get_future());
  if (local_work_queue) {
    local_work_queue->push(std::move(task));
  } else {
    pool_work_queue.push(std::move(task));
  }
  return res;
}
```

Algoritmo 17 – Método detach.

No Algoritmo 17 a implementação do método *detach* utiliza *type traits* para obter o tipo do retorno da função. O invocável passado como parâmetro para o método *detach* é encapsulado pelo `packaged_task`. Para que seja possível aguardar o retorno da execução da tarefa é criado um *future* que irá receber o resultado da tarefa. Caso a tarefa seja criada dentro do *worker* ele é adicionado na fila de tarefas local, caso contrário é adicionado na fila de tarefas do *pool*.

```
void run_pending_task() {
  task_type task;
  if (pop_task_from_local_queue(task) ||
      pop_task_from_pool_queue(task) ||
      pop_task_from_other_thread_queue(task)) {
    task();
  } else {
    std::this_thread::yield();
  }
}
```

Algoritmo 18 – Método para executar as tarefas.

No Algoritmo 18 por avaliação curto-circuito o método primeiramente tenta executar uma tarefa da fila local, caso esteja vazia tenta executar do *pool* de tarefas e finalmente caso nenhuma tarefa seja encontrada ele tenta roubar uma tarefa de outro *worker*. Caso não consiga encontrar nenhum tarefa em nenhuma das filas o *thread* chama o método *yield* que coloca o *thread* no fim da lista de prioridades.

5.7 Reduce

O algoritmo de escalonamento *work-stealing* facilita a implementação do padrão paralelo *reduce*, pois a solução é recursiva e o *work-stealing* consegue tirar melhor proveito desse tipo de estrutura, como mostra o Algoritmo 19. Para a implementação do *reduce* utilizamos a seguinte abordagem: (1) A assinatura do método recebe dois parâmetros template, o tipo do invocável e o tipo da estrutura de dados que contém os elementos a serem reduzidos; (2) O retorno do método é deduzido utilizando *type traits* do C++; (3) É instanciada a classe do escalonador *work-stealing*, detalhado na Seção anterior; (4) Caso o número de elementos do conjunto de dados seja menor que quatro não são feitas novas subdivisões e é retornado o valor da função redutora aplicada a estes elementos; (5) Caso ainda seja possível dividir o conjunto de dados é gerado dois novos conjuntos de dados contendo metade dos elementos cada; (6) Por fim o algoritmo é chamado recursivamente sobre esses dois conjuntos de dados;

```
1  template<typename FunctionType, typename DataType> ←❶
2  static typename std::result_of<FunctionType(DataType)>::type ←❷
3  reduce(FunctionType f, DataType data) {
4
5    static work_stealing thread_pool; ←❸
6
7    if(std::size(data) < 4) {
8      return thread_pool.detach(std::bind(f, data)).get(); ←❹
9    }
10
11   int half = std::size(data) / 2;
12
13   DataType first_half(std::begin(data), std::begin(data) + half); ←❺
14   DataType second_half(std::begin(data) + half, std::end(data));
15
16   DataType result{
17     thread_pool.submit(std::bind(f, first_half)).get(),
18     thread_pool.submit(std::bind(f, second_half)).get()
19   };
20
21   return thread_pool.submit(std::bind(f, result)).get();
22 }
```

Algoritmo 19 – Trecho de código da implementação do algoritmo *reduce*.

O Algoritmo 20 mostra um exemplo de uso do padrão *reduce*. A utilização é semelhante ao algoritmo *map*. Como primeiro parâmetros é esperado a função redutora que será aplicada aos elementos, é necessário iniciar a váriavel que receberá o elemento sumarizado com a identidade para a operação. O segundo parâmetro é o vetor que contendo os elementos a serem sumarizado pela operação *reduce*.

```
int main() {
  std::vector<int> vector(100);
  std::iota(std::begin(vector), std::end(vector), 0);

  int sum = algorithms::reduce(
    [](const std::vector<int>& vector) -> int {
      int result = 0;
      for (int it : vector) {
        result += it;
      }
    return result;
  }, vector);

  std::cout << sum << std::endl;
}
```

Algoritmo 20 – Trecho de código da utilização do padrão paralelo *reduce*.

Como o padrão paralelo divide o conjunto de dados de entrada recursivamente e os valores calculados nas iterações são locais a cada tarefa não é necessário a utilização de sincronização no acesso à variável que armazena o resultado.

5.8 Considerações Sobre o Capítulo

A interface de programação é bastante próxima à linguagem C++, facilitando a adaptação de algoritmos sequenciais já existentes. Isso é um fator que influencia a utilização de uma biblioteca em algum projeto de *software*, especialmente em códigos mais antigos, que não foram projetados para serem paralelos. A utilização de uma nova biblioteca que exija diversas modificações do código tende a introduzir erros na lógica da programação, dificultando o refatoramento destes códigos.

É importante destacar também que, como a biblioteca é completamente desenvolvida em C++, sem a utilização de bibliotecas externas, apenas alguns arquivos de *headers*, a compilação não necessita de parâmetros extras, apenas a adição da biblioteca *PThread* que é usada pela implementação do C++ em sistemas *Unix-like* para a criação dos *threads*, através da diretiva de compilação `-pthread`.

Embora uma implementação paralela que utilize uma política de escalonamento padrão possa obter tempos de execução melhores que as versões sequencias na grande maioria dos casos, alguns casos o desempenho pode ser otimizado fazendo

ajustes no escalonador. Para esses casos a ferramenta permite a criação de políticas de escalonamento especificas para cada algoritmo, permitindo assim uma maior flexibilidade e melhor adaptação aos sistemas onde forem ser executados. No caso de programas específicos para algum tipo de arquitetura ou sistema, podem se aproveitar desta funcionalidade e otimizar o escalonador para determinados casos específicos. Outro caso que pode ser aplicado é em sistemas embarcados onde o consumo de recursos seja limitado, ou até mesmo o consumo de energia, políticas de escalonamento que explore esses aspectos podem ser facilmente implementadas.

6 AVALIAÇÃO

Neste capítulo temos os casos de testes utilizados para a avaliação da interface e a aferição do desempenho. Para a aferição do desempenho foi realizada a combinação da execução de duas estratégias de escalonamento sobre três *benchmarks* bastante conhecidos, geração de fractal, multiplicação de matrizes e *Embarrassingly Parallel* (EP) do *benchmark* NAS.

6.1 Considerações Sobre a Interface

O ganho em relação a OpenMP e TBB está na possibilidade de implementar uma política de escalonamento específica para diferentes padrões de algoritmos iterativos. O desenvolvimento de uma nova estratégia de execução pode vir a ser um aspecto que aumente a complexidade do desenvolvimento, mas oferece a vantagem de permitir acesso a melhores índices de desempenho

A interface de descrição de concorrência proposta não pode ser comparada diretamente com OpenMP ou Cilk Plus, pois, ao contrário destas, a proposta parte de uma abordagem orientada a objetos. No entanto, ressalta-se que a proposta apresentada é bastante similar a TBB. Essa similaridade permite a migração de aplicações desenvolvidas para TBB para a ferramenta desenvolvida.

```
1  void matrix_mult(int M, double *a, double *b, double *c) {
2      tbb::parallel_for(0, M, [=](int i) {
3          for (int j = 0; j < M; ++j) {
4              int c_index = i*M+j;
5              for (int k = 0; k < M; ++k) {
6                  c[c_index] += a[i*M + k] * b[k*M+j];
7              }
8          }
9      });
10 }
```

Algoritmo 21 – Trecho de código da implementação de Multiplicação de Matrizes em TBB.

No Algoritmo 21, temos uma implementação de multiplicação de matrizes em TBB (VOSS; ASENJO; REINDERS, 2019), como podemos observar no Algoritmo 22, as interfaces das duas ferramentas são bastante semelhantes. O fato de ambas utilizarem funções lambdas para definição do invocável que será chamado nas tarefas simplifica a definição da concorrência, porém a nossa interface difere de TBB permitindo a alteração da política de escalonamento.

Outro ponto onde as interfaces divergem é na especificação do laço mais externo, em TBB esse laço é omitido do corpo da função. Essa limitação dificulta a compreensão do código, em uma primeira análise essa omissão pode prejudicar a leitura do código, exigindo conhecimento prévio da implementação de TBB para entender que o laço é omitido, e que o índice de cada iteração é passado como parâmetro para a função lambda.

```
1  void matrix_mult(int M, double *a, double *b, double *c) {
2      algorithms::map(
3          [&](const int &begin, const int &end) {
4              for(int i = begin; i < end; i++) {
5                  for (int j = 0; j < M; ++j) {
6                      int c_index = i * M + j;
7                      for (int k = 0; k < M; ++k) {
8                          c[c_index] += a[i * M + k] * b[k * M + j];
9                      }
10                 }
11             }
12         }, 0, M
13     );
14 }
```

Algoritmo 22 – Trecho de código da implementação da Multiplicação de Matrizes em C++.

Um dos aspectos relevantes da implementação é a fácil adaptação da interface para algoritmos sequencias já existentes. Exemplos de códigos mostrados no Capitulo 5 mostram alguns exemplos de paralelização de algoritmos.

6.2 Aferição de Desempenho

A implementação de uma política de escalonamento para o padrão paralelo *map* foi validada e o desempenho medido focando em três tipos de aplicações. Um aplicação onde o grão das tarefas não é uniforme, para esse caso escolhemos o algoritmo de Mandelbrot, adaptando uma implementação utilizando os recursos fornecidos pelo C++17 (GALOWICZ, 2017). A outra aplicação é uma multiplicação de matrizes adaptado de (VOSS; ASENJO; REINDERS, 2019), onde o grão das tarefas é uniforme. Um terceiro algoritmo do *benchmark* NAS (BAILEY et al., 1994) foi adaptado de uma

implementação do trabalho (GRIEBLER et al., 2018).

Foram desenvolvidos dois algoritmos de escalonamento para execução dos casos de testes. Um semelhante ao escalonamento *dynamic* do OpenMP e outro semelhante ao *static*. Nas legendas dos gráficos apresentando os desempenhos aferidos, OpenMP (*dynamic*) e OpenMP (*static*) referem-se respectivamente aos escalonadores *dynamic* e *static* do OpenMP. Task (*dynamic*) e Task (*static*) são as nossas implementações dos escalonadores. TBB (*auto*), TBB (*simple*) e TBB (*static*) são as formas como o TBB divide as tarefas, *auto* e *simple* dividem em diversas tarefas para gerar um melhor balanceamento de carga e o *static* divide as iterações entre os *workers* disponíveis sem a possibilidade de balanceamento de carga durante a execução, abordagem semelhante ao *static* do OpenMP.

Os testes foram executados em uma máquina com sistema operacional Fedora 30, 64-*bit*, processador Intel i5-8250U, com 8 núcleos de 1.60GHz cada e 8GB de memória RAM. Para coleta dos dados foram realizados 30 execuções para cada caso.

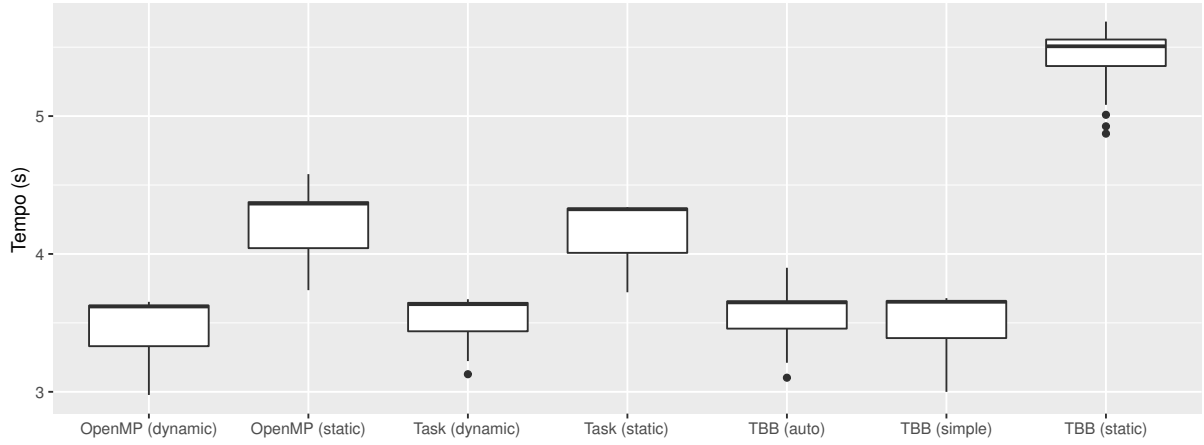

Figura 17 – Tempos de execução para o algoritmo de *Mandelbrot*. No eixo x temos as implementações nas ferramentas e suas variações na divisão dos dados entre dinâmico e estático. No eixo y temos o tempo de execução em segundos. As implementações apresentam dois comportamentos, um no estático onde a carga de trabalho não é homogênea todas as implementações tem um tempo de execução próximo e mais elevado e o outro comportamento é na divisão dinâmica onde o grande número de tarefas criadas consegue manter os processadores ocupados na maior parte do tempo resultando em um tempo de execução mais rápido nas três ferramentas.

Para o algortimo de *Mandelbrot* foram utilizadas matrizes de 1000×1000, para gerar mais carga para que o *overhead* na criação das tarefas nas ferramentas não fossem os tempos dominantes.

Os tempos de execução para a implementação do algoritmo de *Mandelbrot* são mostrados na Figura 17. Nestes resultados observamos dois comportamentos, quando as tarefas são divididas de forma estática, ou seja, dividindo a computação igualmente entre os *threads* que executaram, os tempos de execução foram mais ele-

vados. Isso se deve ao fato de que o algoritmo de *Mandelbrot* é irregular. Essa divisão fixa entre os *threads* não consegue balancear a carga eficientemente. Esse tipo de divisão em TBB demonstrou um desempenho abaixo das outras ferramentas, o *overhead* adicionado para o gerenciamento das tarefas juntamente com esse tipo tarefas irregulares gerou tempos de execução mais elevados.

Os tipos de divisões dinâmicas, onde as ferramentas geram mais tarefas que o número de *threads*, obtiveram os melhores resultados para esse caso. Como podemos observar, as três ferramentas apresentaram tempos de execução muito próximos.

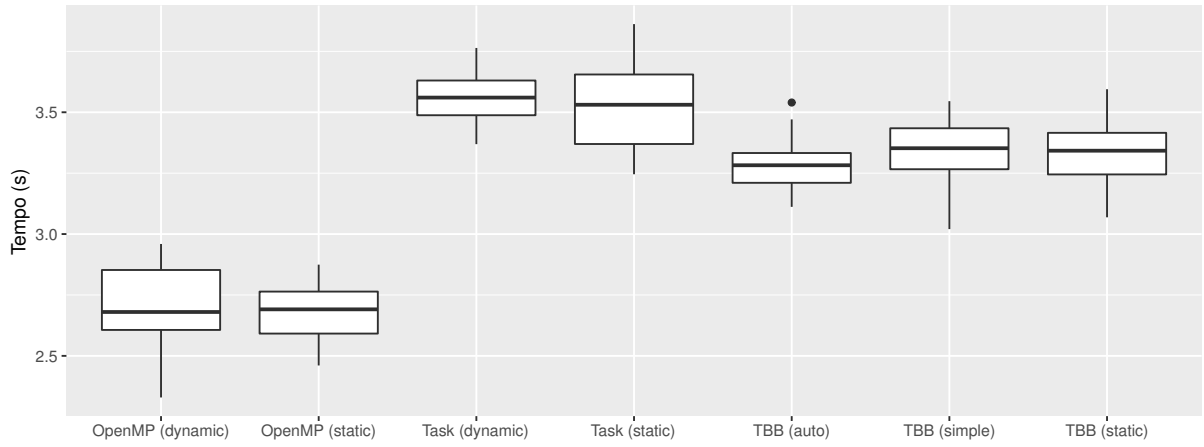

Figura 18 – Tempos de execução para o algoritmo de multiplicação de matrizes. No eixo x temos as implementações nas ferramentas e suas variações na divisão dos dados entre dinâmico e estático. No eixo y temos o tempo de execução em segundos. Nas duas abordagens de divisão dos dados (estática e dinâmica) as ferramentas apresentaram tempo de execução semelhantes. A carga de trabalho homogênea faz com que todas as tarefas executem aproximadamente na mesma fatia de tempo. OpenMP teve melhores resultados devido a otimizações para esse tipo de aplicação.

Na Figura 18 são apresentados os tempos de execução para a implementação do algoritmo de multiplicação de matrizes. Os resultados apresentados permitem comparar o desempenho da abordagem de escalonamento proposta com a oferecida por implementações análogas utilizando OpenMP e TBB. Para as execuções foram utilizadas matrizes de 1024×1024 inicializadas com os mesmos valores.

Este algoritmo gera tarefas com o mesmo grau de complexidade para a carga de trabalho. Podemos observar que as três ferramentas tiveram tempos de execução diferentes, mas mantendo tempos de execução próximos para as diferentes formas de divisão das tarefas de cada ferramenta.

Para algoritmos regulares a criação de muitas tarefas, como podemos observar, não melhora o desempenho da execução, podendo em determinados casos o *overhead* de criação e gerenciamento destas tarefas prejudicar o desempenho.

Neste caso o que observamos que OpenMP obteve um desempenho melhor do que o as outras ferramentas. Isso deve-se ao fato de que OpenMP é otimizado para

a execução de laços onde o carga de trabalho dos *chuncks* é homogênea. O custo de oferecer uma estrutura mais complexa de escalonamento, como a presente na nossa proposta e mesmo em TBB, não tem seus custos cobertos na execução de um algoritmo *map* em que as tarefas tenham custos homogêneos.

O algoritmo *Embarrassingly Parallel* (EP) é um dos kernels que compõem o *NAS Parallel Benchmark* (BAILEY et al., 1994). O *kernel* do EP gera um grande número de pares aleatórios gaussianos de acordo com um esquema específico. Sua computação intensiva está concentrada no laço principal, que processa independentemente cada grupo de pares de números aleatórios. Considerando que algumas iterações do laço têm mais números para gerar do que outras, cada iteração calcula seu *offset* para equilibrar a carga. Em seguida, números pseudo-aleatórios uniformes são enumerados para cada grupo para calcular o desvio gaussiano pelo método de aceitação-rejeição (GRIEBLER et al., 2018). O resultado é uma redução dos valores dos pares, para sincronização na redução foi utilizado `mutex` do C++.

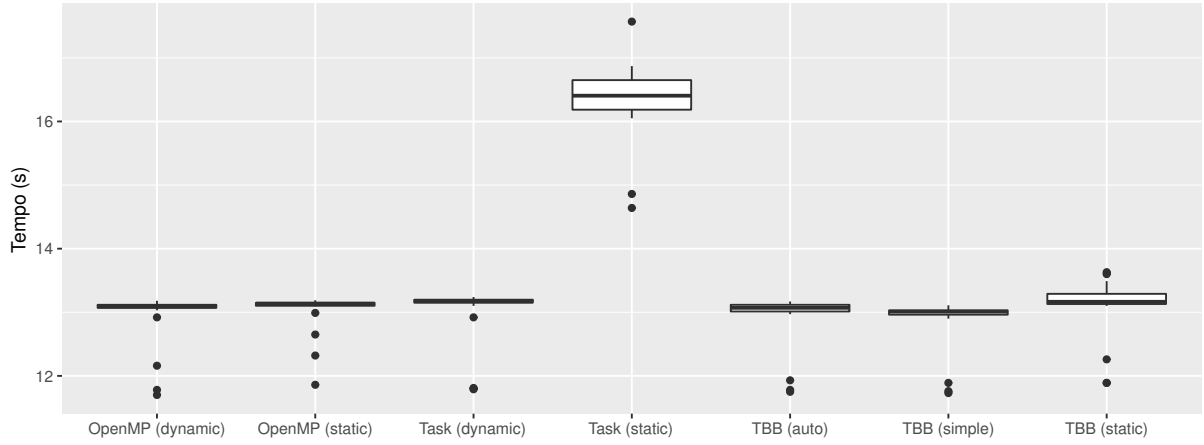

Figura 19 – Tempos de execução para o algoritmo *EP*. No eixo x temos as implementações nas ferramentas e suas variações na divisão dos dados entre dinâmico e estático. No eixo y temos o tempo de execução em segundos. Nas duas abordagens de divisão dos dados (estática e dinâmica) as ferramentas apresentaram tempo de execução semelhantes. A carga de trabalho homogênea faz com que todas as tarefas executem aproximadamente na mesma fatia de tempo.

Na Figura 19, são apresentados os tempos de execução para o algoritmo EP. Devido a implementação adaptada utilizar a mensuração do tempo de execução em segundos com apenas duas casas decimais após a vírgula, não foi possível analisar a variação do tempo de execução com precisão para cada variação do estudo de caso. Porém observamos que os tempos de execução das ferramentas ficaram muito próximos. Com exceção da divisão das iterações de forma estáticas na nossa ferramenta que obteve um tempo de execução ligeiramente maior que as demais variações.

6.3 Considerações Sobre o Capítulo

A política de escalonamento desenvolvida não obteve o melhor desempenho em todos os casos de testes, esse comportamento era esperado, pois as políticas de escalonamento não conseguem obter o melhor desempenho em todos os casos. Para os casos onde o a tarefa não era homogênea, como no caso do *Mandelbrot*, a abordagem utilizada para o escalonamento mostrou tempos de execução semelhantes as outras ferramentas.

7 CONSIDERAÇÕES FINAIS

As arquiteturas *multicore* consolidaram-se como a tecnologia padrão para a produção de processadores atualmente. Isso por que, nesse paradigma, o ganho de performance por parte dos programas aplicativos advém da exploração efetiva dos recursos paralelos desse tipo de arquitetura. As ferramentas que facilitam o uso do paralelismo do *hardware* muitas vezes não podem oferecer a melhor estratégia de escalonamento para cada algoritmo, limitando a expressividade do programador.

A identificação de padrões na programação paralela é importante para auxiliar o programador na solução do problema. Quando entendemos a natureza do problema conseguimos facilmente definir qual tipo de solução implementar. As ferramentas de programação paralela atuais, descritas nos capítulos anteriores, oferecem interfaces de alto-nível para os principais padrões paralelos. Outros padrões que não estão implementados como um recurso da ferramenta, podem ser facilmente implementados utilizando-se os recursos da ferramenta.

Conhecendo os padrões e entendendo como funciona o escalonador da ferramenta é possível obter resultados que favoreçam o desempenho da aplicação. Não existe um escalonador perfeito, como vimos diferentes ferramentas aplicam diferentes estratégia de escalonamento para tentar solucionar o problema da forma mais eficiente possível. Estudando os principais padrões paralelos utilizados e a forma como normalmente esses algoritmos são decompostos paralelamente em tarefas ajuda a elaborar novas estratégias de distribuição da carga entre os processadores, para melhorar o desempenho.

Neste trabalho apresentamos uma interface para permitir a adequação da política de escalonamento de acordo com o tipo de problema a ser resolvido. Exemplos de escalonadores apresentadas na avaliação da ferramenta mostram que essa abordagem pode obter melhores resultados que a utilização de uma única política de escalonamento para todos os algoritmos.

Os diferenciais da abordagem apresentada em relação às ferramentas comerciais, como as apresentadas no Capítulo 2, são decorrentes de três considerações iniciais:

1. Considerar que as implementações dos algoritmos que descrevem a solução do problema da implementação e a heurística de escalonamento são realizadas de forma independente;

2. Considerar que a mesma estratégia de escalonamento pode não ser ideal para todos os trechos concorrentes de um programa;

3. Considerar que o uso de uma linguagem de programação consolidada no mercado de desenvolvimento é mais indicado.

Assim, a proposta construída distingue claramente as implementações do algoritmo da aplicação e do algoritmo de escalonamento. Com isso, o programador pode optar em utilizar estratégias de escalonamento já desenvolvidas ou desenvolver novas, considerando particularidades de sua aplicação. É possível a coexistência, em um mesmo programa, de diferentes estratégias de escalonamento, sendo possível ativar, para cada trecho concorrente, aquela identificada como mais apropriada a sua execução. A materialização da prova de conceito se deu na linguagem C++, explorando recursos oferecidos em sua versão C++17.

7.1 Trabalhos Futuros

Como trabalhos futuros temos a realização de mais experimentos, mesclando um maior número de problemas e estratégias de escalonamento. A implementação dos problemas de *benchmarks* voltados para a usabilidade, como o Cowichan, e para o desempenho, como o NAS Parallel Benchmarks. Considerando o crescente número de arquiteturas heterogêneas, onde temos CPUs juntamente com GPGPUs e coprocessadores *manycore*, uma abordagem para trabalhos futuros é a introdução de estratégias de escalonamento que considerem essas arquiteturas heterogêneas.

REFERÊNCIAS

ASANOVIC, K. et al. **The Landscape of Parallel Computing Research**: A View from Berkeley. Berkeley, CA, USA: EECS Department, University of California, Berkeley, 2006. (UCB/EECS-2006-183).

ASANOVIC, K. et al. A View of the Parallel Computing Landscape. **Commun. ACM**, New York, NY, USA, v.52, n.10, p.56–67, Oct. 2009.

BAILEY, D. H.; BARSZCZ, E.; DAGUM, L.; SIMON, H. D. **NAS Parallel Benchmark Results 10-94**. Moffett Field, CA - USA: NASA Ames Research Center, 1994.

BLUMOFE, R. D. **Executing Multithreaded Programs Efficiently**. 1995. Tese (Doutorado em Ciência da Computação) — Massachusetts Institute of Technology, Cambridge, MA, USA.

BLUMOFE, R. D.; LEISERSON, C. E. Space-Efficient Scheduling of Multithreaded Computations. **SIAM J. Comput.**, Philadelphia, PA, USA, v.27, n.1, p.202–229, 1998.

BLUMOFE, R. D.; LEISERSON, C. E. Scheduling Multithreaded Computations by Work Stealing. **Journal of the ACM**, New York, NY, USA, v.46, n.5, p.720–748, 1999.

BOOST. **Boost 1.72.0 Library Documentation**. Disponível em: <https://www.boost.org/doc/libs/1_72_0/>. Acesso em: 26/02/2020.

BURTON, F. W.; SLEEP, M. R. Executing Functional Programs on a Virtual Tree of Processors. In: CONFERENCE ON FUNCTIONAL PROGRAMMING LANGUAGES AND COMPUTER ARCHITECTURE, 1981., 1981, New York, NY, USA. **Proceedings...** ACM, 1981. p.187–194. (FPCA '81).

BUTENHOF, D. R. **Programming with POSIX Threads**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

CARLINI, P. et al. **The GNU C++ Library Manual**. Disponível em: <https://gcc.gnu.org/onlinedocs/libstdc++/manual/index.html>. Acesso em: 01/03/2018.

CASAVANT, T. L.; KUHL, J. G. A taxonomy of scheduling in general-purpose distributed computing systems. **IEEE Transactions on Software Engineering**, [S.l.], v.14, n.2, p.141–154, Feb 1988.

CHAPMAN, B.; JOST, G.; PAS, R. v. d. **Using OpenMP**: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation). Cambridge, MA, USA: The MIT Press, 2007.

DAREMA, F. SPMD Computational Model. In: PADUA, D. (Ed.). **Encyclopedia of Parallel Computing**. Boston, MA: Springer US, 2011. p.1933–1943.

EVANS, N.; OLIVIER, S. L.; BARRETT, R.; STELLE, G. Scheduling Chapel Tasks with Qthreads on Manycore: A Tale of Two Schedulers. In: INTERNATIONAL WORKSHOP ON RUNTIME AND OPERATING SYSTEMS FOR SUPERCOMPUTERS ROSS 2017, 7., 2017, New York, NY, USA. **Proceedings...** Association for Computing Machinery, 2017. (ROSS '17).

FEITELSON, D. **A Survey of Scheduling in Multiprogrammed Parallel Systems**. [S.l.]: IBM T.J. Watson Research Center, 1994. (Research report).

FLYNN, M. Flynn's Taxonomy. In: PADUA, D. (Ed.). **Encyclopedia of Parallel Computing**. Boston, MA: Springer US, 2011. p.689–697.

FLYNN, M. J. Some Computer Organizations and Their Effectiveness. **IEEE Transactions on Computers**, [S.l.], v.C-21, n.9, p.948–960, Sept 1972.

FREE SOFTWARE FOUNDATION. **The GNU C++ Library**. Disponível em: <https://gcc.gnu.org/onlinedocs/libstdc++/>. Acesso em: 26/02/2020.

GALOWICZ, J. **C++17 STL Cookbook**. [S.l.]: Packt Publishing, 2017.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. **Design Patterns**: Elements of Reusable Object-Oriented Software. USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

GOLDBERG, D. What Every Computer Scientist Should Know About Floating-point Arithmetic. **ACM Comput. Surv.**, New York, NY, USA, v.23, n.1, p.5–48, Mar. 1991.

GRIEBLER, D. et al. Efficient NAS Benchmark Kernels with C++ Parallel Programming. In: EUROMICRO INTERNATIONAL CONFERENCE ON PARALLEL, DISTRIBUTED AND NETWORK-BASED PROCESSING (PDP), 2018., 2018. **Anais...** [S.l.: s.n.], 2018. p.733–740.

INTEL. **Getting Started with Parallel STL**. Disponível em: <https://software.intel.com/en-us/get-started-with-pstl>. Acesso em: 01/03/2018.

INTEL. **Cilk Plus Task Scheduler**. Disponível em: <https://www.cilkplus.org/faq/20>. Acesso em: 04/09/2017.

INTEL. **Migrate your application to use OpenMP or Intel(R) TBB instead of Intel(R) Cilk(TM) Plus**. Disponível em: <https://software.intel.com/en-us/articles/migrate-your-application-to-use-openmp-or-intelr-tbb-instead-of-intelr-cilktm-plus>. Acesso em: 18/03/2020.

INTEL. **Intel Threading Building Blocks Documentation**. Disponível em: <https://software.intel.com/en-us/tbb-documentation>. Acesso em: 26/02/2018.

INTEL. **Worksharing Using OpenMP**. Disponível em: <https://software.intel.com/en-us/cpp-compiler-18.0-developer-guide-and-reference-worksharing-using-openmp>. Acesso em: 27/02/2018.

INTEL. **Get Started with Parallel STL**. Disponível em: <https://software.intel.com/en-us/articles/get-started-with-parallel-stl>. Acesso em: 26/02/2020.

IORDAN, A. C.; JAHRE, M.; NATVIG, L. Tuning the victim selection policy of Intel TBB. **Journal of Systems Architecture**, [S.l.], v.61, n.10, p.584 – 591, 2015.

ISO/IEC. **Multi-threading Library for Standard C++ (Revision 1)**. [S.l.: s.n.], 2008.

ISO/IEC. **Working Draft, Standard for Programming Language C++**. [S.l.: s.n.], 2012.

ISO/IEC. **Working Draft, Standard for Programming Language C++**. [S.l.: s.n.], 2017.

MCCOOL, M.; REINDERS, J.; ROBISON, A. **Structured Parallel Programming**: Patterns for Efficient Computation. 1st.ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012.

MEYERS, S.; ALEXANDRESCU, A. C++ and the Perils of Double-Checked Locking. **Doctor Dobbs Journal**, [S.l.], v.29, 07 2004.

OPENMP. **OpenMP Application Programming Interface Version 4.5**. [S.l.: s.n.], 2015.

ORACLE. **JDK 1.1 for Solaris Developer's Guide**. Disponível em: <https://docs.oracle.com/cd/E19455-01/806-3461/ch2mt-41/index.html>. Acesso em: 01/03/2018.

PIRKELBAUER, P.; WILSON, A.; PETERSON, C.; DECHEV, D. Blaze-Tasks: A Framework for Computing Parallel Reductions over Tasks. **ACM Trans. Archit. Code Optim.**, New York, NY, USA, v.15, n.4, Jan. 2019.

REINDERS, J. **Intel Threading Building Blocks**. 1st.ed. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2007.

SCHARDL, T. B.; MOSES, W. S.; LEISERSON, C. E. Tapir: Embedding Fork-Join Parallelism into LLVM's Intermediate Representation. **SIGPLAN Not.**, New York, NY, USA, v.52, n.8, p.249–265, Jan. 2017.

SHIINA, S.; TAURA, K. Almost Deterministic Work Stealing. In: INTERNATIONAL CONFERENCE FOR HIGH PERFORMANCE COMPUTING, NETWORKING, STORAGE AND ANALYSIS, 2019, New York, NY, USA. **Proceedings...** Association for Computing Machinery, 2019. (SC '19).

SILBERSCHATZ, A.; GALVIN, P. B.; GAGNE, G. **Operating System Concepts**. 9th.ed. Hoboken, NJ, USA: Wiley Publishing, 2012.

THE OPEN GROUP. IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(R)) Base Specifications, Issue 7. **IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)**, [S.l.], p.1–3951, Jan 2018.

VOSS, M.; ASENJO, R.; REINDERS, J. **Pro TBB**: C++ Parallel Programming with Threading Building Blocks. Berkeley, CA: Apress, 2019. 754p.

WILLIAMS, A. **C++ Concurrency in Action**: Practical Multithreading. [S.l.]: Manning, 2012. (Manning Pubs Co Series).

WISMÜLLER, R. Loops, Parallel. In: PADUA, D. (Ed.). **Encyclopedia of Parallel Computing**. Boston, MA: Springer US, 2011. p.1079–1087.