Cristiane de Andrade

# Modelo unificado de padrões paralelos elásticos para implementação de aplicações

Cascavel-PR

2023

Cristiane de Andrade

# Modelo unificado de padrões paralelos elásticos para implementação de aplicações

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre pelo Programa de Pós-Graduação em Ciência da Computação (PPGComp) da Universidade Estadual do Oeste do Paraná – Unioeste, campus de Cascavel.

Universidade Estadual do Oeste do Paraná – Unioeste – Cascavel

Centro de Ciências Exatas e Tecnológicas – CCET

Programa de Pós-Graduação em Ciência da Computação – PPGComp

Orientador: Dr. Guilherme Galante

Cascavel-PR

2023

Cristiane de AndradeAndrade, Cristiane
Modelo unificado de padrões paralelos elásticos para implementação de aplicações/ Cristiane de Andrade. – Cascavel-PR, 2023-
97p. : il. (algumas color.) ; 30 cm.

Orientador: Dr. Guilherme Galante

Dissertação (Mestrado)– Universidade Estadual do Oeste do Paraná – Unioeste – Cascavel
Centro de Ciências Exatas e Tecnológicas – CCET
Programa de Pós-Graduação em Ciência da Computação – PPGComp, 2023.

1. Programação paralela. 2. Padrões de programação paralela. 2. Elasticidade. I. Guilherme Galante. II. Universidade Estadual do Oeste do Paraná. III. Faculdade de Ciência da Compputação. IV. Modelo unificado de padrões paralelos elásticos para implementação de aplicações

Cristiane de Andrade

# Modelo unificado de padrões paralelos elásticos para implementação de aplicações

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre pelo Programa de Pós-Graduação em Ciência da Computação (PPGComp) da Universidade Estadual do Oeste do Paraná – Unioeste, campus de Cascavel.

Trabalho aprovado. Cascavel-PR, 07 de dezembro de 2023:

---

**Dr. Guilherme Galante**
Orientador(a)

---

**Dr. Luiz Antonio Rodrigues**
Universidade Estadual do Oeste do Paraná

---

**Dr. Marcio Seiji Oyamada**
Universidade Estadual do Oeste do Paraná

---

**Dr. Rodrigo da Rosa Righi**
Universidade do Vale do Rio dos Sinos

Cascavel-PR
2023

*Dedico este trabalho ao meu filho Pietro*

# Agradecimentos

Agradeço primeiramente a Deus por me prover força e coragem para não desistir diante das dificuldades.

Agradeço ao meu filho Pietro, por ser a luz da minha vida, fonte de força e coragem. Agradeço também aos meus pais por todo o apoio e por toda ajuda.

Agradeço ao meu padrinho César, por não me deixar desistir e me incentivar a sempre fazer o que precisava ser feito.

Agradeço ao meu orientador pelos ensinamentos, pela paciência, pelo respeito e compreensão, por entender todas as dificuldades que enfrentei em fazer esta tese sendo mãe solo, trabalhando e sem rede de apoio.

Agradeço as minhas amigas, Alexandra, Eloísa e Jessica por revisarem meus textos, por não me deixarem desistir. Agradecimentos também a minha psicologa Elaine, que acreditou em mim, me incentivou e esteve presente na minha defesa.

Agradeço também aos professores presentes na minha banca, por todas as trocas e ensinamentos durante a minha avaliação.

*“Um trabalho de amor atinge sua plenitude na colheita,*
*e esta chega sempre no tempo certo"*

# Resumo

ANDRADE, Cristiane de.. **Modelo unificado de padrões paralelos elásticos para implementação de aplicações**. Orientador: Dr. Guilherme Galante. 2023. 97f. Dissertação (Mestrado em Ciência da Computação) – Universidade Estadual do Oeste do Paraná, Cascavel – Paraná, 2023.

Atualmente, todos os computadores possuem algum nível de paralelismo utilizável. Os sistemas modernos são explicitamente equipados com suporte de hardware para essa funcionalidade, incluindo vários nós, núcleos, CPUs e aceleradores. No entanto, o desenvolvimento de software para computadores paralelos é um desafio, devido à variedade de considerações que os programadores devem ter em conta durante o processo de criação. Além dos desafios relacionados ao hardware, a dinâmica das aplicações, sujeitas a variações inesperadas de carga, é comum no contexto da Computação de Alto Desempenho (HPC). Neste sentido, padrões paralelos foram propostos para mitigar algumas complexidades. No entanto, há uma notável ausência de padrões que abordem o projeto e a construção de aplicações elásticas. Assim, este trabalho busca expandir os padrões existentes na literatura, propondo um modelo de desenvolvimento de aplicações e de padrões que incorpore suporte à elasticidade. O objetivo é auxiliar o desenvolvedor em todas as fases de projeto e implementação de aplicações paralelas. Adicionalmente, o trabalho abrange uma revisão de alguns *frameworks* que podem ser empregados para implementar aplicações elásticas. Por fim, este trabalho demonstra a aplicação deste modelo e os padrões elásticos propostos na definição da arquitetura de aplicações utilizadas em HPC. Este estudo avança em direção a uma mentalidade de programação que reconhece a importância de lidar com diferentes ofertas e variações de hardware e software, um aspecto crucial para a próxima geração de aplicações HPC.

**Palavras-chave**: programação paralela; padrões de programação paralela; elasticidade.

# Abstract

ANDRADE, Cristiane de.. **Unified patterns model for parallel programming and elasticity**. Orientador: Dr. Guilherme Galante. 2023. 97f. Dissertação (Mestrado em Ciência da Computação) – Universidade Estadual do Oeste do Paraná, Cascavel – Paraná, 2023.

Currently, all computers feature some level of usable parallelism. Modern systems are explicitly equipped with hardware support for this functionality, including multiple nodes, cores, CPUs, and accelerators. However, software development for parallel computers is challenging due to the variety of considerations programmers must address during the creation process. In addition to hardware-related challenges, the dynamic nature of applications, subject to unexpected load variations, is common in the context of High-Performance Computing (HPC). In this regard, parallel patterns have been proposed to mitigate some complexities. Nevertheless, there is a notable absence of standards addressing the design and construction of elastic applications. Thus, this work aims to expand upon existing standards in the literature by proposing a model for application development and patterns that incorporate support for elasticity. The objective is to assist the developer in all phases of designing and implementing parallel applications. Additionally, the work encompasses a review of some frameworks that can be employed to implement elastic applications. Finally, this work demonstrates the application of this model and the proposed elastic patterns in defining the architecture of applications used in HPC. This study advances towards a programming mindset that recognizes the importance of dealing with different offerings and variations of hardware and software, a crucial aspect for the next generation of HPC applications.

**Keywords**: parallel computing; patterns of papallel programming; elasticity

# Lista de ilustrações

# Lista de tabelas

# Lista de abreviaturas e siglas

| | |
|---|---|
| API | *Application Programming Interface* |
| CPU | *Central Processing Unit* |
| EBS | *Elastic Batch Scheduler* |
| ERS | *Elastic Runtime Schedule* |
| E/S | Unidade de Entrada ou Saída |
| GPU | *Graphics Processing Unit* |
| HPC | *High Performance Computing* |
| MDF | Método das diferenças finitas |
| MPI | *Message Passing Interface* |
| MV | Máquina virtual |
| PDE | Equações diferenciais parciais |
| PE | Elemento de processamento |
| RAM | *Random Access Memory* |
| RMS | *Resource Management System* |
| SLA | *Service Level Agreement* |
| SPMD | *Single Program, Multiple Data* |
| UE | Unidade de execução é um termo genérico para uma coleção de entidades em execução concorrente, geralmente processos ou *threads* |
| URL | *Uniform Resource Locator* |
| VCPU | *Virtual Centralized Processing Unit* |

# Sumário

# 1 Introdução

Atualmente todos os computadores possuem algum grau de paralelismo e são explicitamente equipados com algum suporte de hardware, incluindo instruções vetoriais, núcleos *multithreaded*, processadores *multicore*, processadores múltiplos, GPUs ou coprocessadores paralelos. Nesse cenário, o paralelismo tornou-se uma estratégia imperativa para se explorar de forma criteriosa e eficiente os abundantes recursos de computação disponíveis nas arquiteturas modernas.

Anteriormente os programas eram sempre pensados e projetados de forma sequencial, já que o desempenho dependia de um único núcleo de processamento, o que torna programar de forma paralela um desafio (SUTTER et al., 2005).

Nos últimos anos, os desenvolvedores de compiladores buscaram responder à necessidade de paralelizar as aplicações implementando, por exemplo, a detecção de paralelismo em *loops* simples. Alguns compiladores de última geração, como o compilador Intel, podem detectar automaticamente tal paralelismo (INTEL, 2022). Porém, esses recursos são limitados e nem todos os programas sequencias conseguem algum beneficio apenas com esse paralelismo provisionado pelos compiladores, pois acaba se perdendo o paralelismo de granularidade mais fina (PROGRAMMIERUNG, 2021). Isso, então, tem levado os programadores a pensar e projetar aplicações que executem de forma paralela, a fim de atingir um aumento de desempenho, além de utilizar efetivamente as arquiteturas. Contudo, programar essas aplicações é uma tarefa desafiadora e suscetível a erros (PACHECO, 2011).

Isso, então, tem levado os programadores a pensar e projetar aplicações que executem de forma paralela, a fim de atingir um aumento de desempenho, além de utilizar efetivamente as arquiteturas. Contudo, programar essas aplicações é uma tarefa desafiadora e suscetível a erros (PACHECO, 2011). Ela permite, por exemplo, explorar o poder de processamento de sistemas com múltiplos núcleos, aceleradores gráficos e *clusters* de computadores. Além disso, a programação paralela também desempenha um papel crucial na computação de alto desempenho, pois permite a distribuição das tarefas entre vários nós de um *cluster* de computadores, possibilitando a execução rápida e eficiente desses cálculos (PACHECO, 2011).

Atualmente existem muitas aplicações que necessitam extrair o máximo desempenho dos processadores. Exemplos dessas aplicações são: aplicações de modelagem climáticas, dobramento de proteínas, descoberta de novas drogas e pesquisa de energia (PACHECO, 2011). Com o crescimento exponencial dos dados e a necessidade de processamento rápido em áreas como inteligência artificial, simulação computacional e renderização gráfica, a

programação paralela tornou-se um componente vital para atender às demandas modernas (BRAWER, 2014).

Com o crescimento exponencial dos dados e a necessidade de processamento rápido em áreas como inteligência artificial, simulação computacional e renderização gráfica, a programação paralela tornou-se um componente vital para atender às demandas modernas (MARTÍN-ÁLVAREZ et al., 2023), devido às inúmeras considerações que os programadores devem levar em conta durante o processo de desenvolvimento. Isso inclui a implementação da sincronização de dados, a distribuição e coordenação de tarefas computacionais em unidades de processamento e a garantia de que elas serão executadas corretamente, o aproveitamento da simultaneidade, as sessões críticas, tipos distintos de acesso à memória, tolerância a falhas e otimização de hardware de baixo nível (MCCOOL; REINDERS; ROBISON, 2012), (SENA et al., 2013). Problemas como condições de corrida, *deadlock* e coerência de cache tornam-se mais complexos quando múltiplos *threads* estão envolvidos (KIESSLING, 2009).

Considerando todos esses desafios, metodologias têm sido propostas para o projeto e desenvolvimento de aplicações paralelas. Um dos métodos mais conhecidos para projetar algoritmos paralelos é a metodologia de Foster (1995). Essa metodologia permite que o programador se concentre inicialmente nos aspectos independentes da arquitetura, como a concorrência e a escalabilidade. Depois deve-se considerar os aspectos dependentes da arquitetura, como aumentar a localidade e diminuir a comunicação da computação.

A abordagem de Foster (1995) embora auxilie no desenvolvimento de aplicações paralelas, trazendo um direcionamento muito importante, não define padrões para serem utilizados. Neste contexto, diante dos desafios de desenvolver aplicações paralelas, a comunidade têm se dedicado a desenvolver padrões para facilitar o desenvolvimento dessas aplicações.

Abordando tal cenário e visando abstrair tais complexidades, foram propostas abordagens de design de desenvolvimento baseadas em padrões paralelos. Padrões de programação fornecem soluções reutilizáveis para problemas comuns e ajudam a evitar *bugs* de simultaneidade, como *deadlocks* e data *races*, que são muito difíceis de localizar (PROGRAMMIERUNG, 2021). Outra vantagem dos padrões de programação paralela, segundo (MCCOOL; REINDERS; ROBISON, 2012), é o fato de que os padrões evitam o não determinismo dos programas paralelos. Programas paralelos determinísticos são programas que obtém o mesmo resultado independentemente da ordem de escalonamento das tarefas. Um padrão paralelo é um princípio orientador que ajuda os desenvolvedores a paralelizar um aplicativo sugerindo como implementar computação concorrente. Usando essa abordagem, os desenvolvedores podem selecionar facilmente um padrão paralelo apropriado e seguir um padrão bem estabelecido para dividir um programa em blocos e

aproveitar de paralelismo (CZAPPA et al., 2021).

Nesse contexto, destacam-se as iniciativas Mattson, Sanders e Massingill (2004) e McCool, Reinders e Robison (2012), que propuseram padrões paralelos para estruturar o desenvolvimento de programas paralelos. O primeiro é voltado principalmente para o projeto arquitetônico de aplicativos, enquanto o segundo apresenta uma abordagem que se concentra na estratégia de algoritmo (ou esqueletos). Posteriormente essas duas obras serviram de base para o surgimento de outras propostas para padrões de aplicações paralelas (DOOLEY; DOOLEY, 2017), (DANELUTTO et al., 2021) e (KEUTZER; MATTSON, 2009).

A complexidade do desenvolvimento pode aumentar ainda mais ao considerar a elasticidade para aplicações paralelas, ou seja, aplicações cujos recursos ou configurações podem variar durante a execução. Uma compreensão profunda da elasticidade, seus mecanismos e das soluções relacionadas, será essencial à medida que se avança em direção à próxima geração de arquiteturas paralelas e de sistemas de alto desempenho (HPC), de forma que seja possível considerar a computação em nuvem como um ambiente virtualizado viável para a execução de aplicativos exigentes.

Esse recurso pode melhorar significativamente o desempenho e eficiência, reduzir custos, incorporar tolerância a falhas e balanceamento de carga e permitir melhor utilização de recursos. Considerando esses cenários, os benefícios da elasticidade para computação paralela tornam-se claros e desejáveis.

Atualmente não há na literatura padrões de programação paralelas unificados e que abordem a implementação da elasticidade. Este trabalho apresenta um modelo de programação paralela baseado em padrões, voltado principalmente para o projeto arquitetônico de aplicativos, com a definição de padrões elástico de programação paralela. Para isso, inicialmente propõem-se um modelo de projeto arquitetônico, o qual apresenta as fases de implementação de aplicações, as avaliações necessárias em cada fase, a combinação de padrões paralelos disponíveis no estado da arte para atender aos requisitos das aplicações e, por fim, propõem-se formas de adicionar a elasticidade apresentando padrões elásticos para implementação da elasticidade. Ademais, para cada padrão apresentado propõem-se *frameworks* presentes no estado da arte para a implementação. Em contraste com trabalhos relacionados, este trabalho adiciona à literatura novas formas de pensar e organizar aplicações paralelas diante de *softwares* cada vez mais dinâmicos que rodam em infraestruturas cada vez mais heterogêneas.

O restante do documento está estruturado da seguinte forma. O Capítulo 2 traz uma visão geral sobre programação paralela, abordando os principais conceitos envolvidos e também apresenta os padrões de programação paralela propostos por Mattson, Sanders e Massingill (2004) e McCool (2010). O Capítulo 4 apresenta um modelo de padrões de

programação paralela unificado, além de uma revisão do estado da arte sobre *Frameworks* propostos para a implementação da elasticidade nestes padrões. O 3 apresenta uma revisão do estado da arte sobre elasticidade e os *frameworks* para implementação de padrões paralelos. O Capítulo 5 aborda a aplicação do modelo proposto e destes *Frameworks*. O Capítulo 6 apresenta a conclusão deste trabalho.

# 2 Padrões de projeto para desenvolvimento de aplicações paralelas

Com o objetivo de auxiliar no desenvolvimento de aplicações paralelas, Mattson, Sanders e Massingill (2004) e McCool, Reinders e Robison (2012) trazem uma valiosa contribuição em seus trabalhos, com o intuito de trazer uma abstração em alto nível para os desenvolvedores de aplicações. Esses trabalhos trazem um direcionamento para o desenvolvedor em relação à análise de requisitos e aos cuidados no que tange a programabilidade e manutenção dos algoritmos paralelos.

Nas próximas seções apresentam-se os padrões de projeto propostos por Mattson, Sanders e Massingill (2004) e por McCool, Reinders e Robison (2012) que visam facilitar o desenvolvimento de aplicações paralelas.

## 2.1 Padrões de projeto propostos por Mattson, Sanders e Massingil (2004)

Em sua obra, Mattson, Sanders e Massingill (2004) propõem um padrão para desenvolver aplicações paralelas, organizando o projeto de desenvolvimento de aplicações paralelas em quatro fases:

1. Encontrando a concorrência;
2. Estrutura do algoritmo;
3. Estrutura de suporte;
4. Mecanismos de implementação.

Cada uma destas fases é detalhada nas próximas seções.

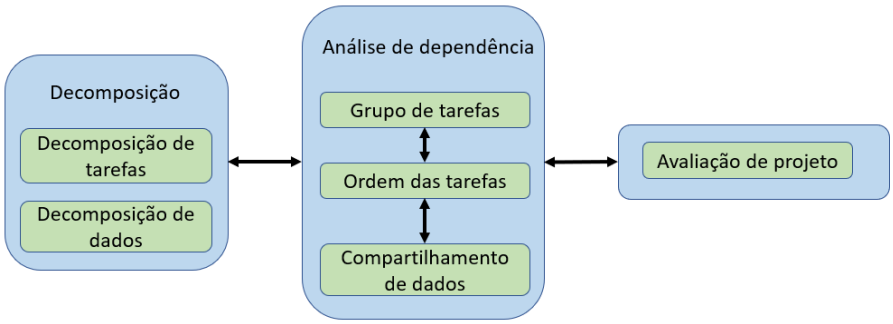
### 2.1.1 Encontrando a Concorrência

A concorrência foi explorada pela primeira vez na computação com o objetivo de melhorar a utilização ou compartilhar os recursos de hardware de um computador. Os sistemas operacionais suportam a troca de contexto para permitir que várias tarefas pareçam ser executadas concorrentemente, permitindo assim que um determinado trabalho útil ocorra enquanto o processador está ocupado executando uma outra tarefa.

A primeira fase de projeto "encontrando a concorrência", tem por objetivo definir, em alto nível, quais são os pontos da aplicação em que se pode explorar a concorrência, definir o tipo de concorrência e também como a implementar. Nesta fase, o programador precisa definir se o problema analisado é grande o suficiente, além de se os resultados são significativos o bastante para justificar o esforço despendido na criação de um algoritmo paralelo.

Mattson, Sanders e Massingill (2004) definem três espaços de projeto a serem considerados nesta fase de projeto: decomposição, análise de dependências e avaliação do projeto. Esses espaços de projeto são mostrados na Figura 1 detalhados nas próximas seções.

Figura 1 – Encontrando a Concorrência.




Padrões de decomposição: na fase de padrões de decomposição, é importante definir qual decomposição deve ser feita primeiro: decomposição de dados ou de tarefas. No padrão de decomposição de dados, os dados utilizados pelas tarefas são decompostos em pedaços distintos uma vez que esse padrão permite a escalabilidade. Alguns exemplos de aplicação de decomposição de dados são: cálculo envolvendo *arrays*, além de estruturas de dados recursivas, como árvores. Ao decompor as estruturas de dados, é necessário considerar as seguintes questões:

1. 1. Flexibilidade. O tamanho e o número dos blocos de dados devem ser flexíveis para suportar uma ampla gama de *hardware* paralelos. Uma abordagem possível é definir blocos que são controlados por um pequeno número de parâmetros. Esses parâmetros definem “ajustes” de granularidade que podem ser utilizados para modificar o tamanho dos blocos de dados, para que esses blocos correspondam ao *hardware* onde está sendo executada a aplicação. Porém, muitas aplicações não possuem granularidade sempre adaptáveis. O *overhead*, necessário para gerenciar as dependências entre as partes, é onde se pode analisar o impacto da granularidade na decomposição de dados. Quando a decomposição de dados for eficiente, as dependências dos dados escalam em uma dimensão menor do que o esforço computacional associado a cada pedaço.

2. Eficiência. Os blocos de dados devem ser grandes o suficiente para que a quantidade de trabalho realizada ao atualizar o bloco compense a sobrecarga do gerenciamento de dependências. Outra questão que deve ser considerada é como os blocos são mapeados nas Unidades de Execução, uma vez que um algoritmo paralelo eficaz deve equilibrar a carga entre Unidades de Execução.

3. Simplicidade. Decomposições de dados muito complexas podem ser muito difíceis de depurar. Um dado a ser decomposto geralmente exigirá um mapeamento de um índice global para um índice local.

No padrão de decomposição de tarefas, o programador identifica quais são os pontos do algoritmo que geram as tarefas que poderão ser executadas em paralelo, além de saber quais dessas tarefas possuem uma computação mais intensiva, as principais estruturas de dados utilizadas, e como esses dados são utilizados. É necessário levar em conta os seguintes aspectos: garantir que não ocorra *overhead* através do gerenciamento de dependência das tarefas; permitir que o número e o tamanho de tarefas possa ser migrado para diferentes tipos e números de processadores e arquiteturas de hardware. O padrão de decomposição de tarefas vê o problema como um fluxo de instruções que podem ser quebrados em tarefas, que são executadas de forma concorrente. Para realizar uma decomposição de tarefas de forma eficaz, é necessário garantir que as tarefas sejam independentes para que o gerenciamento destas dependências leve apenas uma pequena fração do tempo de execução geral do programa. É também importante garantir que, a execução das tarefas possa ser distribuída uniformemente entre o conjunto de unidades de processamento (balanceamento de carga).

No padrão de decomposição de tarefas, o programador identifica quais são os pontos do algoritmo que geram as tarefas que poderão ser executadas em paralelo, além de saber quais dessas tarefas possuem uma computação mais intensiva, as principais estruturas de dados utilizadas, e também como esses dados são utilizados. É necessário levar em conta os seguintes aspectos: garantir que não ocorra *overhead* através do gerenciamento de dependência das tarefas, além de permitir que o número e o tamanho de tarefas possam ser migrados para diferentes tipos, números de processadores e arquiteturas de hardware. O padrão de decomposição de tarefas vê o problema como um fluxo de instruções que pode ser quebrado em tarefas, as quais são executadas de forma concorrente. Para realizar uma decomposição de tarefas de forma eficaz, é necessário garantir que as tarefas sejam independentes para que o gerenciamento dessas dependências leve apenas uma pequena fração do tempo de execução geral do programa. Além disso, também é importante garantir que a execução das tarefas possa ser distribuída uniformemente entre o conjunto de unidades de processamento (balanceamento de carga).

Análise de dependência: a segunda etapa é a análise de dependência, na qual o programador analisa se as dependências na execução ocorrem por grupo de tarefas, ordem de tarefas ou compartilhamento de dados. Essa análise tem como objetivo avaliar se a decomposição de tarefas e dados realizada permite a paralelização do algoritmo, e também avaliar as dependências entre essas tarefas e dados.

Padrão de grupo de tarefas: o padrão de grupo de tarefas tem como objetivo agrupar as tarefas para simplificar o trabalho de gerenciamento de dependências. Esse padrão pode ser aplicado após a decomposição de tarefas e a decomposição de dados terem sido identificadas. Este é o primeiro passo na análise de dependências entre as tarefas dentro de um problema. Ao decompor as tarefas, pode parecer inicialmente que elas não estão relacionadas, mas durante a avaliação do grupo de tarefas pode-se perceber que elas possuem restrições semelhantes em sua execução simultânea e, portanto, podem ser agrupadas. Em alguns casos as tarefas de um grupo são realmente independentes umas das outras. Existem dependências temporais e também de ordem de execução.

Padrão de ordem de tarefas: de acordo com os autores, o objetivo desse padrão é encontrar e contabilizar corretamente dependências resultantes de restrições na ordem de execução de um grupo de tarefas. O livro cita que há três principais restrições em relação à ordem de execução de um grupo de tarefas:

1. Dependências temporais: relativo à ordem de execução do grupo de tarefas;
2. Ordem das tarefas: exigência de que tarefas particulares devam ser executadas ao mesmo tempo;
3. Ausência de ordem na execução de um grupo de tarefas: embora não seja uma restrição, conforme os autores, isso é uma característica importante a ser considerada.

Padrão de compartilhamento de dados: esse padrão analisa como os dados são compartilhados pelas tarefas e pelos grupos de tarefas, com o objetivo de gerenciar corretamente o acesso aos dados. O padrão analisa questões de sobrecarga no compartilhamento dos dados pelos grupos de tarefas. Garantir que os dados compartilhados estejam prontos para uso pode levar à sobrecarga de sincronização excessiva. Outro ponto que pode trazer sobrecarga é a comunicação.

Padrão de avaliação do projeto: nesta fase de projeto são avaliados os seguintes itens:

1. Adequação para plataforma alvo: são avaliadas questões como quantidade de processadores ou, ainda, se as tarefas da aplicação se ajustam ao número de processadores;

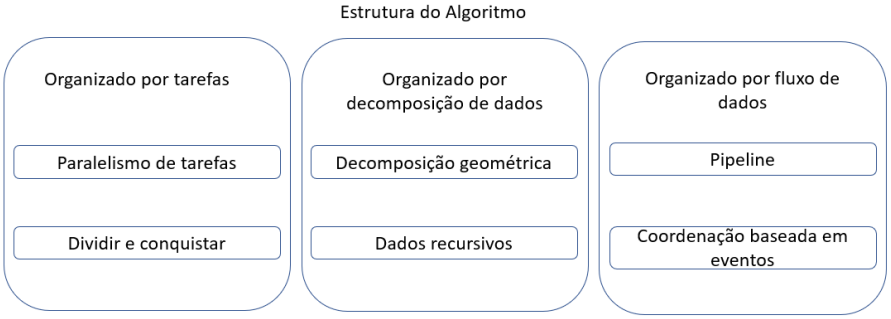
2. Qualidade de projeto: neste ponto é avaliado se, a aplicação dos padrões anteriores trouxe os atributos de simplicidade, flexibilidade e eficiência;

3. Preparação para próxima fase de projeto: são avaliadas se as tarefas e dependências são regulares ou irregulares, ou seja, se possuem tamanhos iguais ou não. Também é avaliado se a interação entre as tarefas é síncrona ou assíncrona, etc.

### 2.1.2 Estrutura do Algoritmo

A fase de estrutura do algoritmo traz a definição de três estruturas de algoritmos, ou seja, a forma de organização do algoritmo, quais sejam: organização por tarefas, organização por decomposição de dados e organização por fluxo de dados. Essas estruturas são mostradas na Figura 2.

Figura 2 – Estrutura do Algoritmo.




Nesta fase é importante considerar as seguintes questões:

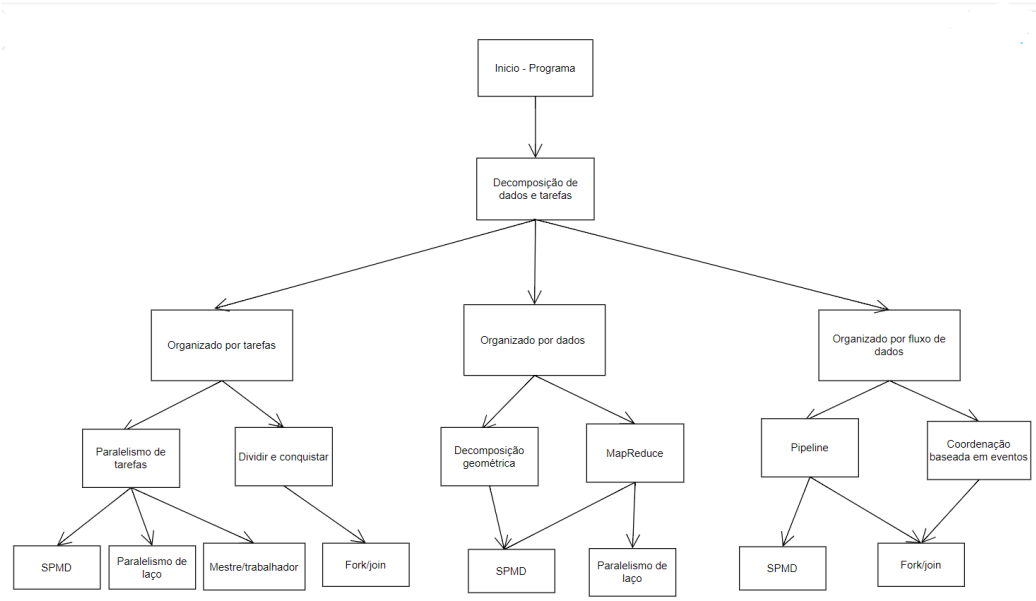
1. Quantas unidades de processamento o sistema possui efetivamente?.

2. Qual o custo para a comunicação entre as unidades de execução?

Para definir a estrutura do algoritmo, os autores propõem a utilização de uma árvore de decisão, apresentada na Figura 3, para auxiliar na hora de escolher a estrutura a ser implementada. Porém, eles destacam que alguns algoritmos podem necessitar da implementação de mais de uma estrutura em uma mesma aplicação.

Organizado por tarefas: a estrutura de organização por tarefas deve ser escolhida na árvore de decisão quando a forma como as tarefas interagem se torna a principal característica do paralelismo. A estrutura denominada paralelismo de tarefas pode ser escolhida quando as tarefas forem completamente independentes umas das outras, o que é definido como algoritmos embaraçosamente paralelos (ALVENTOSA; MARTÍNEZ; QUILIS, 2020), e também em situações em que exista alguma dependência entre as tarefas. Por outro lado, a estrutura dividir e conquistar é mais indicada se as tarefas forem executadas por um procedimento recursivo.

Figura 3 – Árvore de decisão.




Paralelismo de Tarefas: os algoritmos paralelos são, fundamentalmente, uma coleção de tarefas executadas simultaneamente. Para os autores, a chave para uma decomposição de tarefas eficaz é garantir que as tarefas sejam suficientemente independentes, de forma que o gerenciamento de dependências consuma somente uma pequena fração do tempo de execução geral do programa.

Nesta fase de projeto é preciso identificar como estão definidas as tarefas, se é necessário avaliar as dependências entre as tarefas, se a quantidade de tarefas criadas para o algoritmo é suficiente para manter as unidades de execução ocupadas para um melhor escopo de projeto. Inicialmente, busca-se identificar o maior número de tarefas possível, pois é mais fácil mescla-las do que depois de iniciada a execução tentar dividi-las.

Além desses dois passos, é necessário definir como essas tarefas serão atribuídas para cada unidade de execução. As tarefas podem ser definidas todas no início da computação ou, em alguns casos, elas surgem dinamicamente, à medida que a computação é executada.

#### 2.1.2.1 Dividir e conquistar

Na estrutura de dividir e conquistar, os algoritmos são divididos em subproblemas menores, de forma recursiva, e as soluções desses subproblemas são combinadas até gerarem a solução final. Para aplicar essa estrutura de forma a tirar proveito de seu paralelismo intrínseco, é necessário avaliar algumas questões. De acordo com (MATTSON; SANDERS; MASSINGILL, 2004) as partes seriais de um programa podem restringir significativamente a aceleração que pode ser alcançada ao adicionar mais processadores. Portanto, se a divisão e a fusão dos cálculos não são triviais em comparação com a quantidade de cálculo para os casos base, um programa usando esse padrão pode não ser capaz de tirar proveito

de um grande número de processadores (CZARNUL, 2018). Além disso, se houver muitos níveis de recursão, o número de tarefas pode crescer demais, podendo atingir um ponto em que a sobrecarga de gerenciar as tarefas supere qualquer benefício de executá-las de forma paralela (SILVA; BUYYA, 1999). Outras questões a serem consideradas ao aplicarmos essa estrutura são: o custo de comunicação e o tratamento de dependências.

Nessa estrutura, ao mapear as tarefas para as unidades de processamento, existem dois padrões que podem ser utilizados para a sua implementação, o *fork/join* e o padrão mestre/trabalhador. O padrão *fork/join* pode ser utilizado mapeando cada tarefa em uma unidade de execução, e parando a recursão quando o número de sub tarefas ativas seja o mesmo que o número de unidades de processamento. Já quando o problema não não é regular, é indicado criar tarefas mais refinadas usando o padrão mestre/trabalhador para mapear as tarefas em unidades de execução, mantendo uma fila de tarefas e um *pool* de unidades de execução por unidade de processamento (MATTSON; SANDERS; MASSINGILL, 2004). Esses padrões serão abordados mais adiante com maiores detalhes.

### 2.1.3 Organizado por decomposição de dados

De acordo com (MATTSON; SANDERS; MASSINGILL, 2004), o padrão de organização por decomposição de dados deve ser selecionado quando o principal organizador da simultaneidade for a decomposição de dados. Quando o problema é decomposto em subespaços discretos, a estrutura denominada decomposição geométrica seria a melhor escolha. Já a estrutura de dados recursivos deve ser escolhida quando o problema for definido por uma estrutura de dados que seja recursiva, como uma árvore binária, por exemplo.

#### 2.1.3.1 Decomposição Geométrica

Este padrão é muito utilizado na modelagem e resolução de fenômenos que têm representação natural e geométrica no mundo real. Na Figura 4 temos um exemplo do padrão de decomposição geométrica aplicado para um domínio com duas dimensões, onde as caixas verdes são as tarefas. No quadro da esquerda há o agrupamento total das tarefas, e no quadro da direita, após a aplicação da decomposição, as tarefas estão agrupadas em número reduzido (porém, na prática, pode haver condições de contorno onde são necessárias secções parciais ao longo do domínio).

Para a aplicação desse padrão, algumas considerações se fazem necessárias. Se o processamento dos dados é estritamente local, ou seja, todas as informações necessárias estão dentro dos blocos, a simultaneidade é embaraçosamente paralela e o padrão de paralelismo de tarefas deve ser utilizado. Em muitos casos, no entanto, a computação requer informações de pontos em outros blocos, blocos contendo dados que estavam próximos na

estrutura de dados global. Nesses casos, as informações devem ser compartilhadas entre os blocos para concluir a atualização (MATTSON; SANDERS; MASSINGILL, 2004).

Para explorar o paralelismo potencial na aplicação desse padrão, devemos atribuir partes da estrutura de dados decomposta para UE's. Também, devemos garantir que os dados necessários para a atualização de cada bloco estejam disponíveis quando necessário. Projetos para problemas que se encaixam nesse padrão envolvem os seguintes elementos-chave:

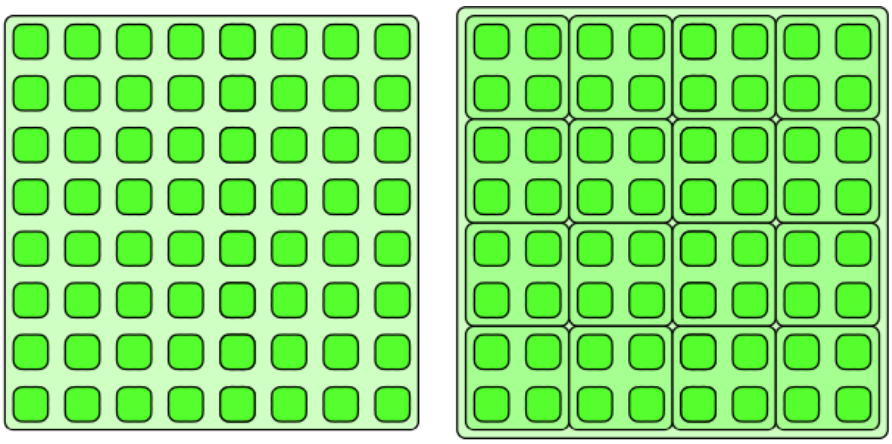
1. Decomposição de dados: particionar a estrutura de dados globais em subdomínios ou blocos. A aplicação deve ser implementada de forma que a granularidade dos dados seja controlada por parâmetros que possam ser facilmente alterados na compilação ou no tempo de execução.

2. Operações de trocas: garantir que cada tarefa tenha acesso a todos os dados necessários para a execução de atualização do seu bloco de dados. É necessário garantir que os dados não locais necessários para a atualização operação sejam obtidos antes de serem necessários

3. Operação de atualização: a atualização da estrutura de dados é feita executando, de forma simultânea, as tarefas responsáveis pela atualização de um bloco.

4. Distribuição de dados e agendamento de tarefas: essa etapa consiste em mapear os blocos para as UE's de forma que melhore o seu desempenho.

Figura 4 – Decomposição Geométrica 2D.



### 2.1.4 Padrão de dados recursivo

O padrão de dados recursivo pode ser utilizado para explorar o paralelismo de problemas que utilizam estruturas de dados de listas, árvores ou grafos. O objetivo

desse padrão é reformular as operações, com a finalidade de que o programa possa operar paralelamente em todos os elementos da estrutura de dados e também operar recursivamente. Um dos principais desafios desse padrão é aplicar a mudança do algoritmo original para explorar a concorrência, já que isso implica que em todos os níveis da estrutura exista concorrência entre os processos. Para isso, cada nó se comunica com o raiz ou com o pai dele. Embora seja recursivo como o padrão de Divisão e Conquista, o padrão de dados recursivos não possui o paralelismo de forma inerente (MATTSON; SANDERS; MASSINGILL, 2004).

### 2.1.5 Organizado por fluxo de dados

Essa estrutura é selecionada se a simultaneidade for obtida quando a ordenação do grupo de tarefas for organizada pelo fluxo de dados. Quando a ordenação das tarefas é regular, unilateral e estática, a estrutura que melhor se aplica seria a de pipeline. Quando o fluxo de dados for irregular, dinâmico e imprevisível, a melhor estrutura seria a coordenação baseada em eventos, que é baseada em eventos assíncronos (MATTSON; SANDERS; MASSINGILL, 2004).

#### 2.1.5.1 Padrão Pipeline

O padrão paralelo *pipeline* é aplicável a uma ampla gama de problemas que são parcialmente sequenciais, ou seja, é possível usar essa técnica para resolver um problema dividindo-o em uma série de tarefas sucessivas. Um *pipeline* pode ser definido como um conjunto de tarefas, ou estágios, simultaneamente ativos que se comunicam em uma relação produtor-consumidor (MCCOOL, 2010) (CZARNUL, 2018). Nesse padrão, cada tarefa é executada por um processador ou processo. Cada processo ou processador compreende um estágio do *pipeline* (ROSSAINZ-LÓPEZ et al., 2017),e cada estágio passa informações para o próximo estágio na sequência. Um algoritmo pode ser implementado utilizando esse padrão se puder ser dividido em uma série de funções que podem ser executadas por estágios do *pipeline* (SILVA; BUYYA, 1999).

Cada etapa contribui para a solução do problema geral. Um dos requisitos fundamentais para obter um bom desempenho desse padrão é a capacidade de enviar mensagens entre processos adjacentes do *pipeline*. Para isso são necessários *links* de comunicação direta entre os PE's, onde os processos adjacentes estão mapeados (WILKINSON, 2006). No entanto, este padrão pode sofrer de gargalos de desempenho se alguns dos estágios de execução que processam pacotes de dados forem muito mais longo que os outros.

O paralelismo implementado pelo padrão *pipeline*, pode ser visto como uma forma de decomposição funcional. Para (MATTSON; SANDERS; MASSINGILL, 2004), este padrão é aplicado em algoritmos organizados por fluxo de dados, e quando a ordenação

desse fluxo for regular, unilateral e estática, ou seja, os estágios do *pipeline* são conhecidos no início da computação e não se alteram durante a execução.

#### 2.1.5.2 Padrão baseado em eventos

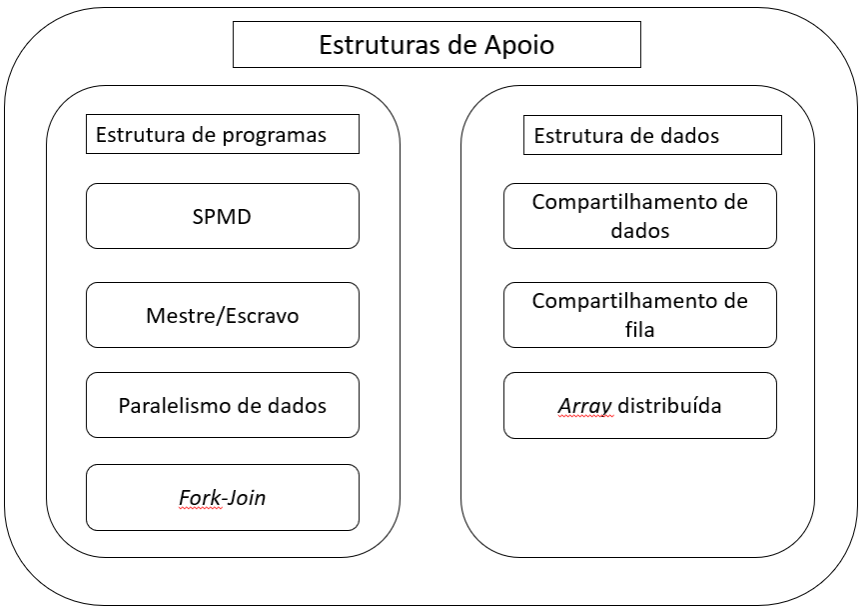
No padrão baseado em eventos, as interações paralelas são modeladas como eventos assíncronos processados por um único *thread* de execução, onde cada evento contém uma tarefa que gera o evento e uma tarefa que processa o evento (GRIEBLER; FERNANDES, 2011).

Esse padrão pode ser utilizado para fluxos de dados. Os eventos são geralmente gerados por entidades externas, um exemplo disso é um sistema de E/S em tempo de execução, que geralmente fornece um retorno de chamada associado ao evento. Neste caso, o desenvolvedor escreve um conjunto de manipuladores de eventos, sendo esses ativados por eventos de entrada (BONETTA, 2014). Outro exemplo seria o *feed* de redes sociais como o Twitter, entre outros (VOSS; ASENJO; REINDERS, 2019).

### 2.1.6 Estrutura de apoio

Na seção anterior, no espaço de projeto de estrutura do algoritmo, é definido se o algoritmo tem uma estrutura majoritariamente baseada na divisão por tarefas, por dados ou por eventos. O espaço de projeto de estruturas de apoio representa, no projeto de implementação de uma aplicação paralela, um estágio intermediário entre a estrutura do algoritmo e os mecanismos de implementação. A Figura 5 apresenta os dois tipos de estrutura de apoio.

Figura 5 – Estrutura de apoio.



De acordo com Mattson, Sanders e Massingill (2004), os padrões neste grupo descrevem abordagens para estruturar o código-fonte da aplicação, ou seja, os padrões que serão efetivamente aplicados para a implementação do algoritmo.

#### 2.1.6.1 *Single Program, Multiple Data*

Em um programa *Single Program, Multiple Data* (SPMD), todas as unidades de execução, *threads* ou processos, executam o mesmo programa em paralelo, mas cada um tem seu próprio conjunto de dados. Diferentes Unidades de Execução (UE's), podem seguir diferentes caminhos pelo programa. A lógica para controle disso é expressa usando um parâmetro que rotula exclusivamente cada UE (SILVA; BUYYA, 1999). Ao contrário do paralelismo dinâmico de tarefas, o conjunto de *threads* ou processos é fixo durante toda a execução do programa (KAMIL, 2012). Este padrão também é conhecido o padrão geométrico, ou padrão de decomposição de domínio, ou padrão de paralelismo de dados (SILVA; BUYYA, 1999).

Aplicações SPMD podem ser muito eficientes se os dados forem bem distribuídos pelos processos e o sistema for homogêneo. Se o sistema apresenta cargas de trabalho ou capacidades diferentes, então este padrão requer o suporte de algum balanceamento de carga capaz de adaptar a distribuição de dados em tempo de execução (CZARNUL, 2018).

Os dados são lidos individualmente por cada processo ou um dos processos é o responsável por ler todos os dados e depois distribui-los pelos processos restantes. Os processos se comunicam normalmente com processos vizinhos e, esporadicamente, existem pontos de sincronização globais (FURQUIM, 2006). Este padrão alcança bom desempenho e um elevado grau de escalabilidade por distribuir bem os dados e ter um modelo de comunicação bem definido.

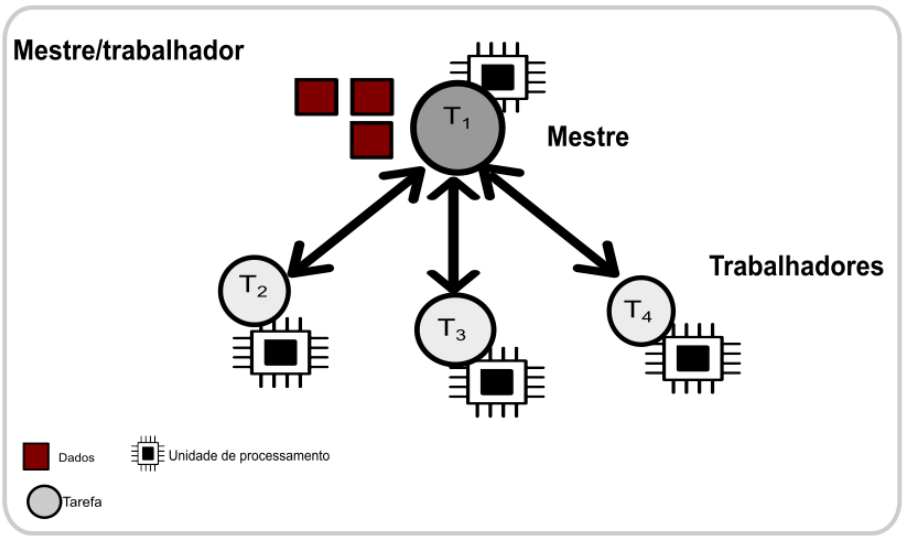
Uma característica do modelo SPMD é a existência frequente de um número de subconjuntos de dados ou tarefas maior que o número de processos rodando. Este padrão de programação paralela é dominante para máquinas de memória distribuída em larga escala, onde as máquinas pertencentes ao sistema possuem diferentes potências computacional e arquitetura (KAMIL, 2012), como apresentada pela computação em nuvem.

De acordo com (MATTSON; SANDERS; MASSINGILL, 2004), o padrão SPMD é a estrutura de apoio ideal para a implementação dos padrões de paralelismo de tarefas, decomposição geométrica, dividir e conquistar, mapreduce e *pipeline*. E de forma satisfatória o padrão coordenação baseado em eventos.

#### 2.1.6.2 Mestre/trabalhador

O padrão mestre/trabalhador é um padrão de projeto comumente usado em sistemas de computação distribuídos para coordenar e gerenciar várias tarefas ou processos. Ele fornece uma maneira de distribuir o trabalho entre um grupo de nós de computação, conhecidos como trabalhadores, enquanto um nó central, conhecido como mestre, coordena e controla suas ações. A Figura 6 mostra o padrão mestre/trabalhador.

Figura 6 – Mestre/trabalhador.




Nesse padrão, o nó mestre é responsável por dividir a carga de trabalho geral em tarefas menores e mais gerenciáveis. Além de atribuí-las aos nós trabalhadores, o nó mestre atua como autoridade central, tomando decisões e coordenando a execução de tarefas. Ele coleta os resultados dos nós trabalhadores e os agrega em uma saída final.

A vantagem deste padrão, de acordo com (HUANG; WANG, 1997) é sua separação controle e computação em dois programas distintos, o que leva a uma melhor estrutura de software. Contudo, nem sempre há um processo ou *thread* explícito no programa.

O referido padrão pode ser utilizado para implementar tanto paralelismo de tarefas quanto paralelismos de dados. O paralelismo de dados é explorado através de várias instâncias de um único programa trabalhador rodando em paralelo (SILVA; BUYYA, 1999).

Já o paralelismo de tarefas pode ser realizado de duas formas. Os programas mestre e trabalhador podem ser executados simultaneamente e cooperar em um pipeline ou diferentes processos trabalhadores podem realmente executar diferentes programas determinados pelo programa mestre para modelar o paralelismo de tarefas (HUANG; WANG, 1997). Neste modelo é criado um *pool* de tarefas de acordo com a análise e restrições avaliadas nas fases anteriores. Um importante ponto a considerar na aplicação desse padrão é o balanceamento de carga (MCCOOL; REINDERS; ROBISON, 2012).O

*pool* de tarefas pode usar balanceamento de carga estático ou balanceamento de carga dinâmico. No balanceamento de carga estático, a distribuição das tarefas é realizada no início da computação, permitindo ao mestre participar da computação após cada trabalhador ser alocado para executar uma tarefa do *pool* de tarefas. A atribuição de tarefas pode ser feita de forma cíclica. Já o balanceamento de carga dinâmico pode ser mais adequado quando o número de tarefas excede o número de processadores, quando os tempos de execução não são previsíveis, quando o número de tarefas é desconhecido no início da execução da aplicação ou quando as PE's tiverem capacidade de execução diferentes (CZARNUL, 2018).

Assumindo que o tempo de execução é consideravelmente mais alto que o tempo de comunicação, este padrão pode atingir alta velocidade de execução e um alto grau de escalabilidade. No entanto, para um grande número de processadores, o controle centralizado do processo mestre pode se tornar um gargalo para as aplicações. É, porém, possível melhorar a capacidade de escala do padrão, estendendo o único mestre a um conjunto de mestres, cada um deles controlando um grupo diferente de trabalhadores (CZARNUL, 2018).

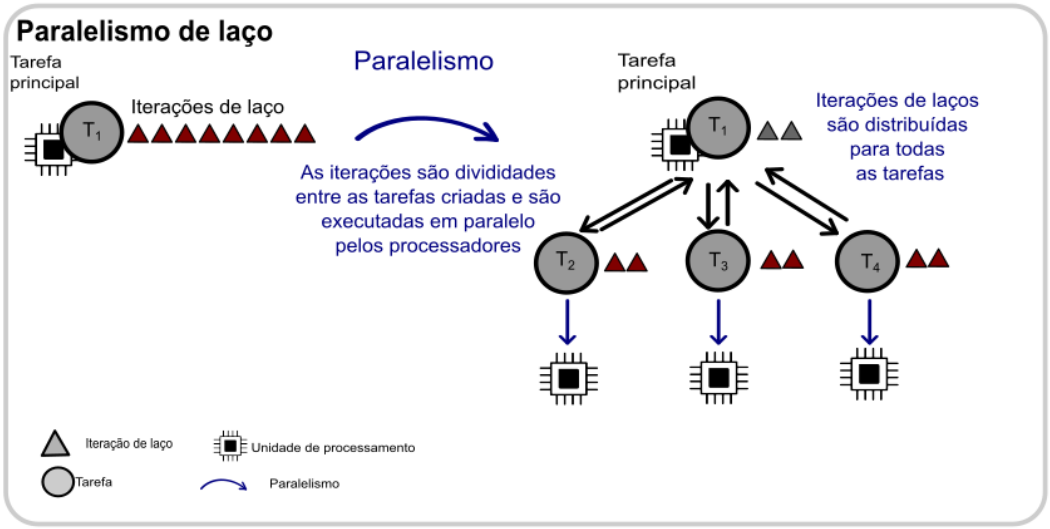
De acordo com Mattson, Sanders e Massingill (2004), , esse padrão é indicado para implementar os padrões de paralelismo de tarefas. O padrão dividir e conquistar pode ser implementado utilizando essa estrutura de apoio, mas com menos desempenho, já que o número de tarefas não costuma ser conhecido no início da execução, pois isso traria questões a mais a serem tratadas, como balanceamento de carga e dependências entre as tarefas.

#### 2.1.6.3 Paralelismo de Laço

A utilização deste padrão tem por objetivo "evoluir" um programa sequencial que executa iterações sobre uma estrutura de dados operando um índice de cada vez em um programa paralelo realizando uma série de transformações nos laços de repetição. São realizadas transformações nos laços que removam as dependências e deixem a semântica geral do programa inalterada. Um programa que explora o paralelismo de laço utilizará vários *threads* ou processos que operam em alguns ou todos os índices ao mesmo tempo a Figura 7 mostra o padrão paralelismo de laço.

Nem todos os problemas podem ser abordados pelo padrão de paralelismo de laço. Isso só funcionará quando a estrutura do algoritmo tiver a maior parte, se não todo o trabalho computacionalmente intensivo, concentrado em um número de laços distintos. Estruturar um algoritmo paralelo utilizando o padrão de paralelismo de laço se torna vantajoso em problemas para os quais já existem programas bem aceitos. Em muitos casos, não é prático reestruturar um programa existente para obter desempenho paralelo, principalmente quando tais programas contém código complicado e algoritmos

Figura 7 – Paralelismo de laço.




mal compreendidos. Essa abordagem só é eficaz quando os tempos de computação para as iterações de laços são grandes o suficiente para compensar a sobrecarga. A aceleração fornecida por esse padrão, no tempo de execução geral do programa, normalmente segue de acordo com a lei de Amdahl. [1]

Uma das questões a serem consideradas na implementação dessa estrutura de apoio é o balanceamento de carga. Existem dois tipos básicos de balanceamento que podem ser aplicados, o estático e o dinâmico. O balanceamento estático consiste em atribuir as iterações do laço paralelo em tempo de compilação, no código da aplicação, para que antes do início do processamento as iterações sejam distribuídas entre as UE's. Já no balanceamento dinâmico, as iterações do laço paralelizado são atribuídas para os recursos em tempo de execução, assim, as iterações são distribuídas entre as UE's somente durante a execução (LUZ, 2018).

De acordo com Mattson, Sanders e Massingill (2004), os padrões de paralelismo de tarefas e decomposição geométrica podem se beneficiar da implementação de tal estrutura. O padrão dividir e conquistar, embora possa ser implementado utilizando esta estrutura de apoio, não é o mais indicado, já que o número de tarefas não costuma ser conhecido no início da execução

#### 2.1.6.4 *Fork/Join*

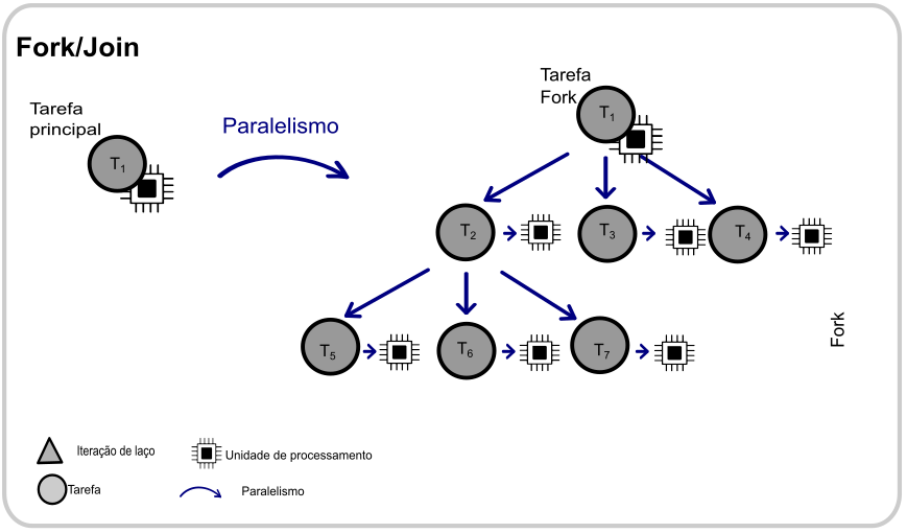
O padrão *fork/join* pode ser aplicado quando o fator mais importante for como as tarefas são gerenciadas pelo algoritmo. Quando as tarefas são geradas recursivamente,

[1] A lei de Amdahl é uma fórmula usada para encontrar a melhoria máxima possível, melhorando uma parte específica de um sistema. Na computação paralela, a lei de Amdahl é usada principalmente para prever a aceleração máxima teórica para o processamento de programas usando vários processadores (AMDAHL, 1967).

possuindo um conjunto de tarefas conectadas de forma irregular, e cuja a carga de trabalho em cada nó varie de maneira imprevisível durante a execução, essa estrutura de apoio pode ser utilizada (MOR, 2010). A Figura 8 mostra como fica o padrão *fork-join* executado de forma paralela.

Problemas que utilizam uma estrutura de algoritmo com padrões de Dividir e Conquistar e Dados Recursivo podem se beneficiar desta estrutura de apoio. Quando as tarefas são geradas recursivamente, conforme abordado na Seção 2.1.6.4, o padrão de estrutura de apoio *fork/join* pode ser aplicado. *Fork/join* permite que o fluxo de controle se bifurque em vários fluxos paralelos até que uma certa granularidade seja atingida. Em seguida, esses fluxos paralelos se unem. Tal abordagem pode ser implementada com um mecanismo eficiente que estende o mecanismo de chamada/retorno orientado a pilha usado para chamadas de funções seriais.

Figura 8 – Fork-Join.




A função*fork*origina sub-tarefas que podem ser executadas em paralelo por não possuírem dependências entre si. Essas sub-tarefas possuem uma instrução *join* no final e, somente quando todos os processos concorrentes tiverem sido executados, as instruções subsequentes do processo principal podem ser executadas. Normalmente, um contador é usado para manter um registro de processos não concluídos. Cada fluxo é independente e eles não são restritos a fazer cálculos semelhantes. Após a junção, apenas um fluxo continua (WILKINSON, 2006) (MOR, 2010).

De acordo com Mattson, Sanders e Massingill (2004), a aplicação de tal padrão de estrutura de apoio pode ser implementada, inicialmente, de duas formas:

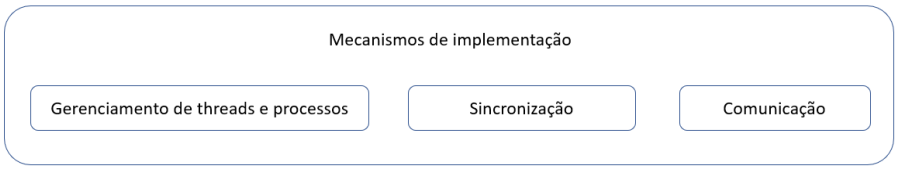
1. Mapeamento direto: Onde cada tarefa é mapeada para uma UE.
2. Mapeamento indireto: onde um *pool* de UE's executa um grupo de tarefas.

É necessário a avaliação da forma de implementação, pois a criação e destruição de UE's é caro computacionalmente. De acordo com (MATTSON; SANDERS; MASSINGILL, 2004), problemas que utilizam uma estrutura de algoritmo com padrões de dividir e conquistar, *pipeline*, coordenação baseada em eventos e mapreduce podem se beneficiar desta estrutura de apoio.

### 2.1.7 Mecanismos de implementação

Esta é a última etapa do espaço de projeto de algoritmos paralelos, a Figura 9 apresenta os três mecanismos de implementação.

Figura 9 – Mecanismos de implementação.




#### 2.1.7.1 Gerenciamento de threads e processos

Gerenciamento de unidades de execução é um mecanismo de implementação responsável pela criação, destruição e gerenciamento dos processos e *threads*.

O conceito de processo pode ser definido como sendo o conjunto de informações necessárias para que o sistema operacional implemente a concorrência de programas. Processos são objetos que carregam informações como memória, registradores e *buffers*. Em um sistema, diferentes processos pertencem a diferentes usuários. Um processo pode conter uma coleção de *threads*. As *threads* compartilham recursos como memória e a comunicação entre elas pertence ao mesmo processo. *Threads* e processos podem ser criados e destruídos, porém *threads* são mais simples e não exigem muitos ciclos de máquina para serem criadas. Por outro lado, os processos são mais custosos, pois quando um processo é criado, é necessário que todas as informações sejam carregadas para definir um lugar no sistema no qual ele irá atuar.

#### 2.1.7.2 Sincronização

O mecanismo de sincronização é a aplicação de restrições na ordenação de eventos que ocorrem em diferentes Unidades de Execução. Isso é utilizado principalmente para garantir que os recursos compartilhados sejam acessados por uma coleção de UE's de forma que o programa esteja correto, independentemente de como os UE's estejam agendados. Existem três tipos principais de sincronização:

- Sincronização de memória: no ambiente computacional de memória compartilhada, *threads* e processos podem executar uma sequência de instruções que são lidas e escritas na memória compartilhada atomicamente.
- Sincronização de barreiras: as barreiras são utilizadas para garantir que um conjunto de *threads* ou processos não prossiga antes que todos cheguem a um determinado ponto.
- Exclusão mutua: impede que dois ou mais processos acessem o mesmo recurso simultaneamente. Quando um processo estiver usando um recurso, todos os demais processos devem ser colocados no estado em espera.

#### 2.1.7.3 Comunicação

O mecanismo de comunicação trata a troca de informações entre Unidades de Execução. A comunicação pode ser realizada utilizando mensagens. Uma mensagem pode conter dados sobre a mensagem e outras informações. A troca de mensagens pode ser feita de duas formas: de uma origem específica para um destino específico e múltipla troca de mensagens entre processos ou *threads* em uma única comunicação. Nesse caso, estão envolvidas diferentes operações, como o mecanismo de *broadcast*, barreiras e redução.

## 2.2 Padrões de projeto propostos por McCool, Reinders e Robinson (2012)

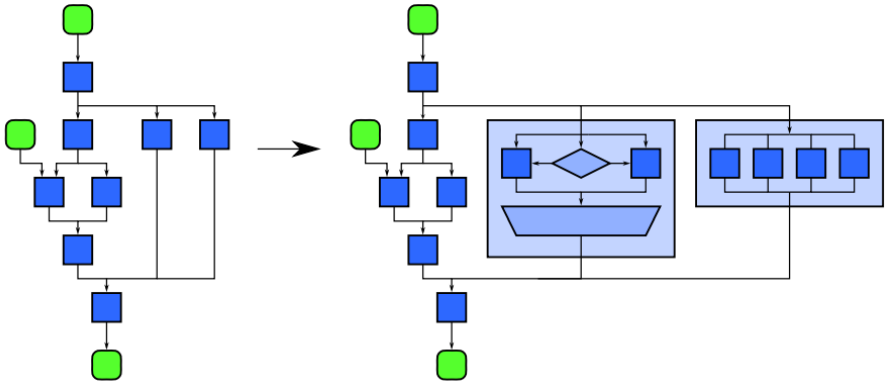
De acordo com McCool, Reinders e Robison (2012), o termo padrões paralelos define uma combinação recorrente de distribuição de tarefas e de acesso a dados que resolve um problema específico no projeto de algoritmos paralelos. Para os autores, um estudo de padrões fornece um "vocabulário" de alto nível para projetar algoritmos e para entender os algoritmos desenvolvidos por outras pessoas. Em seu livro, os autores propõem formas de como projetar e implementar programas em C e C++, baseados em padrões e que permitem o dimensionamento da aplicação. O livro traz exemplos de aplicação dos referidos padrões que são mais conhecidos e aplicados no estado da arte, os quais são fornecidos por bibliotecas e *Frameworks*, como OpenMP, MPI, etc. Ao final do livro, os autores apresentam exemplos de utilização dessas bibliotecas.

Na próxima seção, apresentaremos os seguintes padrões: padrão de aninhamento, padrão de fluxo de controle serial estruturado, padrão de controle paralelo, padrão de gerenciamento de dados seriais e padrão de gerenciamento de dados paralelos.

### 2.2.1 Padrão de Aninhamento

O Padrão de Aninhamento é o padrão de composição fundamental e aparece tanto em programas sequenciais quanto em programas paralelos. Aninhamento refere-se à capacidade de compor padrões hierarquicamente e recursivamente. O aninhamento pode ser estático (relacionado à estrutura do código) ou dinâmico (recursivo, relacionado com a pilha dinâmica de chamadas de função). O padrão de aninhamento significa que todos os "blocos de tarefas" nos diagramas de padrões apresentados na Figura 10são, na verdade, locais dentro dos quais o código pode ser inserido. Esse código pode, por sua vez, ser composto por outros padrões. Na Figura 10, as tarefas que descrevem os cálculos são mostradas como caixas, enquanto os dados são indicados por caixas de cantos arredondados. Os dados agrupados são indicados por caixas de cantos arredondados e tarefas agrupadas são indicadas por caixas poligonais de cantos pontiagudos. Para alguns padrões são introduzidos símbolos adicionais na forma de várias formas poligonais. As dependências de ordenação são dadas por setas. O tempo vai de cima para baixo e, exceto ao representar a iteração, evitamos que as setas subam e, portanto, "recuem" no tempo. Na ausência dessas setas para cima, a altura de um diagrama padrão é uma indicação aproximada de um padrão.

Figura 10 – Padrão de aninhamento (MCCOOL; REINDERS; ROBISON, 2012)



#### 2.2.1.1 Padrões de fluxo de controle serial estruturado

McCool, Reinders e Robison (2012) apresentam um conjunto de padrões sequenciais, já que os padrões paralelos são frequentemente compostos ou generalizados a partir desses padrões sequenciais. Os padrões sequenciais apresentados são, atualmente, a base do que é conhecido como programação estruturada. São os padrões: sequência, seleção, iteração e recursão. Vários padrões paralelos são generalizações dos padrões seriais.

1. **Sequência**: Uma sequência é uma lista ordenada de tarefas que são executadas em uma ordem específica. Cada tarefa é concluída antes de outra começar.

2. **Seleção**: No padrão de seleção existe uma dependência de fluxo de controle entre a condição "c" e as tarefas que serão executadas.

3. **Iteração**: O padrão de iteração diz respeito a um laço de repetição comum. Vários padrões paralelos podem ser considerados paralelizações de formas específicas de laços, incluindo *map, reduce, scan, recurrence, scatter, gather e pack*.

4. **Recursão**: A recursão é uma forma dinâmica de aninhamento que permite que as funções chamem a si mesmas, direta ou indiretamente.

#### 2.2.1.2 Padrões de controle paralelo

Para McCool, Reinders e Robison (2012), cada padrão de controle paralelo está relacionado a um ou mais padrões de controle seriais.

1. ***fork/join***: O padrão *fork/join* foi definido na Seção 2.1.6.4.

2. **Map**: o padrão de map replica uma função, chamada de função elementar, sobre cada elemento de um conjunto de índices. Ela é chamada desta forma pois se aplica aos elementos de uma coleção de dados de entrada. O conjunto de índices pode ser abstrato ou associado aos elementos de uma coleção. O padrão map substitui um uso específico de iteração em programas seriais: um loop no qual cada iteração é independente, em que o número de iterações é conhecido e cada computação depende apenas da contagem de iteração e dos dados lidos como um índice em uma coleção. O padrão map é muitas vezes conhecido como paralelismo trivial e é implementado com outros padrões como *reduction, scan, stencil, recurrence, e etc.*

3. **Stencil**: o padrão de stencil é uma generalização do padrão de map, porém neste padrão a função elementar não acontece de maneira individual, mas sim em todos os dados vizinhos (MCCOOL, 2010). O padrão de stencil é frequentemente combinado com iteração. Ele é implementado quando as entradas para cada instância de um padrão de mapa são baseadas em acesso a uma coleção de dados de entrada usando um conjunto fixo de deslocamentos. Em outras palavras, cada saída de um stencil é uma função de alguma vizinhança de elementos em uma coleção de entrada. Para o padrão de stencil, as condições de contorno nos acessos aos dados vizinhos devem ser consideradas. Esse padrão pode ser implementado em aplicações como detecção de bordas no processamento de imagens, filtros de processamento de imagens e equações diferenciais parciais.

4. **Reduce**: a execução de um reduce combina todos os elementos de uma coleção em um único elemento usando um operador de associatividade. Esse operador é utilizado para todos os elementos de uma coleção reduzindo-a para um único elemento. Pode também ser utilizado de forma combinada com outros padrões para prover maior eficiência no desenvolvimento de programas paralelos. Um exemplo é a utilização do padrão redução associado ao padrão map na multiplicação de matrizes. Tal padrão pode ser aplicado em programas de cálculo da média de amostras de Monte Carlo (aleatórias) para integração, teste de convergência em solução iterativa de sistemas de equações lineares, métricas de comparação de imagens e na codificação de vídeo.

5. **Scan**: o padrão scan, assim como o padrão de redução, é um padrão categorizado como uma operação coletiva, pois é dependente de toda a coleção de dados de entrada (SCHMALFUSS, 2020).Esse padrão calcula todas as reduções parciais de uma coleção, ou seja, uma redução com todos os valores anteriores de cada elemento do conjunto indexado. A operação de Scan produz todas as reduções parciais de uma sequência de entrada, resultando em uma nova sequência de saída. Existem duas variantes: varredura exclusiva e varredura inclusiva. Na varredura exclusiva, o enésimo valor de saída é uma redução sobre o primeiro $n-1$ valores de entrada. Ou seja, a varredura exclusiva exclui o enésimo valor de saída. Na varredura inclusiva, o enésimo valor de saída é uma redução sobre os primeiros n valores de entrada (SAMADI et al., 2014). Aplicações que utilizam Scan incluem manipulação de estruturas de dados para que possam ser usados na computação de posições de objetos antes de um Gather ou Scatter. Alguns exemplos de aplicações que incluem o padrão scan são integração de funções, geração de números aleatórios e análise de séries temporais.

6. **Recurrence**: o padrão recurrence é uma variação do padrão map, que é utilizado para tratar casos mais complexos, nos quais as iterações do laço dependem umas das outras. No padrão de recurrence, os elementos podem usar as saídas de elementos adjacentes como entrada. Se a dependência entre os elementos for constante, o padrão permite a execução paralela de tarefas (como no padrão stencil). Esse padrão é normalmente utilizado em fatoração de matrizes, processamento de imagens, equações diferenciais parciais (PDE), etc. É possível a utilização combinada com outros padrões, por exemplo, com o padrão pipeline, e também com o padrão superscalar sequence.

#### 2.2.1.3 Padrões de gerenciamento de dados paralelos

Vários padrões são usados para organizar o acesso paralelo aos dados. Para evitar problemas de condição de corrida, é necessário definir quando os dados são compartilhados por vários processos e quando não são.

1. **Pack**: o padrão pack, de acordo com (MCCOOL; REINDERS; ROBISON, 2012),pode ser usado para eliminar o espaço não utilizado em uma coleção. Os elementos de uma coleção são marcados com um valor booleano. O padrão pack descarta elementos na coleta de dados que estão marcados como falsos. Os demais elementos marcados como verdadeiros são colocados juntos em uma sequência continua, na mesma ordem em que apareceram na coleta de dados de entrada. Isso pode ser feito para cada elemento da saída de um mapa ou de forma coletiva. Pack é útil quando fundido com o padrão map e outros padrões para evitar saídas inválidas.

2. **Pipeline**: como já citado na Seção 2.1.5.1, o padrão pipeline conecta tarefas em um relacionamento produtor-consumidor. Conceitualmente, todas as etapas do pipeline estão ativas de uma só vez, e cada estágio pode manter o estado que pode ser atualizado conforme os fluxos de dados. O padrão pipeline pode ser implementado de forma linear, ou pode ser implementado como um conjunto de estágios em um grafo acíclico direcionado. Também é possível haver etapas paralelas.

3. **Decomposição geométrica**: O padrão de decomposição geométrica foi definido na Seção 2.1.3.1.

4. **Gather**: o padrão de Gather lê uma coleção de dados de outra coleção de dados, dada uma coleção de índices. Gather pode ser considerado uma combinação do padrão map com operações de leitura do padrão Random read and write. Gather é apenas um conjunto de leituras aleatórias dentro de um mapa. Por exemplo, deslocar dados para a esquerda ou para a direita em uma matriz é um caso especial de implementação do padrão Gather, que pode ser acelerado utilizando operações vetoriais.

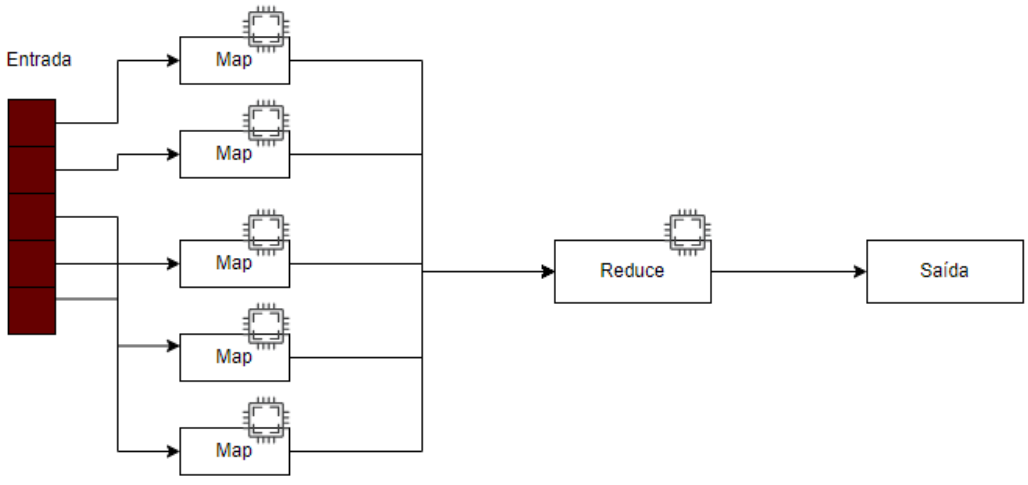
5. **Scatter**: o padrão Scatter é o inverso do padrão de Gather. Um conjunto de dados de entrada e um conjunto de índices são fornecidos, mas cada elemento da entrada é escrito no local determinado, não lido. A dispersão pode ser considerada equivalente a uma combinação do map e padrões de gravação serial aleatórios. Para obter um Scatter determinístico, precisamos de regras para resolver as colisões que podem ocorrer.

### 2.2.2 Padrões coletivos e suas combinações

Padrões coletivos são um conjunto de padrões que lidam com uma coleção de dados, e não com elementos separados. Alguns dos padrões coletivos são: Reduce, fusão dos padrões Map e Reduce, Scan, fusão dos padrões Map e Scan. Esses padrões foram abordados separadamente na Seção 2.2.1.2. Portanto, focaremos aqui na aplicação de suas combinações com o padrão Map.

No padrão mapreduce, as funções map e reduce são utilizadas em conjunto e, normalmente, as saídas produzidas pela execução das funções map são utilizadas como entrada para as funções reduce. Esse padrão pode ser utilizado para implementar problemas que podem ser particionados ou fragmentados em subproblemas. Isso ocorre porque é possível aplicar separadamente as funções map e reduce a um conjunto de dados. Se os dados forem suficientemente grandes, podem ainda ser particionados para a execução de diversas funções Map ao mesmo tempo, em paralelo. Mapreduce é um padrão de programação paralela, para grandes coleções de dados, utilizando computação distribuída em *clusters* ou computação em nuvem. Essas coleções de dados são distribuídas entre as PE's para que esses dados sejam processados em tempo aceitável, onde uma mesma função é executada em todas as PE's em conjuntos de dados diferentes. Ao invés de dividir as etapas de processamento para atingir o paralelismo, este padrão divide a carga de dados. Ou seja, cada PE's é responsável por processar completamente um grupo e dados ao invés de processar todos os dados em uma determinada etapa da computação. Uma ilustração simplificada deste padrão é apresentado na Figura 11, onde temos a entrada dos dados, a função map realizada nesses dados e, no final, é realizado uma função reduce que traz uma saída única. A função map é aplicada na entrada dos dados, seguida de uma função reduce, e outra função map que retorna o resultado (WHITE, 2012).

Figura 11 – MapReduce.



De acordo com os autores, o padrão mapscan, assim como o padrão reduce, pode ser otimizado ao ser implementado junto com outros padrões. É possível implementar o padrão mapscan em três fases: o scan é precedido pela aplicação do padrão map no conjunto de dados e, em seguida, por outra operação utilizando o padrão map. Então, considerando que os blocos de dados sejam do mesmo tamanho, os blocos no primeiro map podem ser combinados com as reduções seriais na primeira fase do padrão scan e, o scan lado a lado na terceira fase, pode ser combinado com o seguinte mapa lado a lado.

Esses padrões são um método eficaz para a implementação de programação paralela, uma vez que tanto map quanto reduce são funções sem estado associado e, portanto, facilmente paralelizáveis.

O foco para Mattson, Sanders e Massingill (2004) está na arquitetura da aplicação, em como avaliar a estrutura das aplicações, propondo um passo a passo para que os programadores consigam pensar em paralelo desde o início da implementação da aplicação. Os autores conseguem propor passos que detalham de forma clara a construção de uma aplicação paralela. Já McCool, Reinders e Robison (2012), têm por foco a definição e aplicação dos padrões para a implementação dos programas, analisando linguagens de programação e como elas disponibilizam esses padrões para implementação.

# 3 Elasticidade e Frameworks

## 3.1 Elasticidade

Nesta Seção iremos propor a inclusão da elasticidade nos padrões paralelos, permitindo assim extrairmos um melhor desempenho das aplicações paralelas. De acordo com Herbst, Kounev e Reussner (2013). elasticidade pode ser definida como a capacidade de um sistema se adaptar de forma dinâmica às mudanças na demanda de recursos computacionais, aumentando ou diminuindo recursos, de forma independente e em tempo de execução, de forma que, em cada ponto no tempo, os recursos disponíveis correspondam à demanda atual o mais próximo possível.

A elasticidade é considerada uma propriedade muito importante da computação em nuvem porque permite que os recursos sejam escalonados dinamicamente conforme sua carga de trabalho aumentar ou diminuir em tempo de execução (RIGHI, 2013) (KEHRER; BLOCHINGER, 2019a) (COUTINHO et al., 2015) (HERBST; KOUNEV; REUSSNER, 2013).

Existem duas modalidades principais para a implementação da elasticidade: horizontal ou vertical (HERBST; KOUNEV; REUSSNER, 2013). A elasticidade horizontal, também conhecida como expansão ou dimensionamento horizontal, envolve a adição ou remoção de instâncias de um nó ou de um sistema distribuído com base na carga de trabalho (GALANTE; BONA, 2012). Isso pode ser obtido provisionando ou desprovisionando dinamicamente máquinas virtuais, contêineres ou outras instâncias na nuvem para lidar com a carga de trabalho. A elasticidade horizontal propicia para as aplicações as seguintes características (RIGHI, 2013):

- Escalabilidade: a elasticidade horizontal permite escalabilidade fácil e rápida do sistema adicionando ou removendo instâncias, tornando-o adequado para cargas de trabalho que exigem alta escalabilidade.
- Tolerância a falhas: ao distribuir a carga de trabalho em várias instâncias, a elasticidade horizontal pode fornecer melhor tolerância a falhas, uma vez que as falhas em uma instância podem ser atenuadas pela redistribuição das tarefas para outras instâncias.
- Custo-benefício: a elasticidade horizontal pode ser econômica para cargas de trabalho com demanda variável e imprevisível, pois os recursos podem ser alocados ou desalocados dinamicamente com base na carga de trabalho.

A elasticidade vertical, também conhecida como dimensionamento vertical, envolve aumentar ou diminuir os recursos como CPU, memória ou armazenamento de um único nó em um sistema para lidar com a carga de trabalho (MOREIRA, 2018). Isso pode ser obtido ajustando dinamicamente a capacidade de uma máquina virtual ou instância de contêiner na nuvem (MELL; GRANCE et al., 2011). Algumas melhorias obtidas com a utilização da elasticidade vertical são:

- Desempenho: a elasticidade vertical pode fornecer melhor desempenho para determinadas cargas de trabalho que exigem muitos recursos de computação ou memória, pois envolve o aumento da capacidade de um único nó.
- Simplicidade: a elasticidade vertical pode ser mais simples de implementar e gerenciar em comparação com a elasticidade horizontal, pois envolve ajustar os recursos de uma única instância em vez de gerenciar várias instâncias.
- Custo: a elasticidade vertical pode ser mais econômica para cargas de trabalho com demanda previsível e estável, pois evita a sobrecarga de gerenciamento de várias instâncias.

A implementação da elasticidade pode ser realizada por meio de políticas, sejam elas manuais ou automáticas. Essas políticas procuram ajustar dinamicamente os recursos computacionais de um ambiente em nuvem para atender às demandas de carga de trabalho (HERBST; KOUNEV; REUSSNER, 2013).As políticas manuais de elasticidade exigem a intervenção humana para definir e ajustar os parâmetros de escalabilidade. Nesse caso, um administrador de nuvem monitora a carga de trabalho e toma decisões com base em indicadores como utilização de recursos, tráfego de rede ou métricas de desempenho. Essas decisões podem envolver aumentar ou reduzir a capacidade de computação disponível, adicionar ou remover instâncias de máquinas virtuais, dimensionar o armazenamento ou ajustar a alocação de recursos (RIGHI, 2013).

Por outro lado, as políticas automáticas de elasticidade empregam algoritmos e mecanismos de automação para realizar os ajustes necessários de forma contínua. Esses algoritmos podem levar em consideração métricas e critérios pré-definidos como, por exemplo, o uso da CPU, o tráfego de rede ou o tempo de resposta para então tomar decisões sobre o dimensionamento automático. Com base nessas métricas, o ambiente em nuvem pode se expandir ou contrair dinamicamente, provisionando ou liberando recursos automaticamente, conforme a necessidade (RODRIGUES, 2016).

Independentemente do tipo de política adotada, é essencial que haja um planejamento cuidadoso e uma definição clara dos objetivos de escalabilidade. As políticas devem levar em consideração os requisitos da aplicação, as demandas de desempenho, os custos envolvidos e as restrições operacionais. Além disso, é importante monitorar continuamente

o ambiente em nuvem para ajustar e otimizar as políticas ao longo do tempo, garantindo uma escalabilidade eficiente e adequada às necessidades da carga de trabalho

Além disso, o controle de elasticidade pode ser implementado de duas formas: proativo ou reativo (RODRIGUES et al., 2018). No método proativo os recursos são alocados ou liberados de acordo com predições da utilização de recursos pela aplicação, baseado no monitoramento da aplicação. Já na abordagem reativa é definido um limiar ou métrica que, quando atingido, os recursos são alocados ou liberados. Ambas as abordagens também podem ser combinadas em uma abordagem híbrida (VERGARA, 2017).

Mecanismos reativos são acionados em resposta à mudanças na carga de trabalho ou nas condições do sistema (KEHRER; BLOCHINGER, 2019c). Eles monitoram constantemente os indicadores de desempenho como a utilização da CPU, o uso da memória e a taxa de transferência de rede. Quando esses indicadores ultrapassam determinados limites predefinidos, os mecanismos reativos entram em ação realizando ajustes automáticos na infraestrutura da nuvem. Por exemplo, eles podem provisionar novas instâncias de máquinas virtuais para lidar com um aumento repentino na demanda ou desativar instâncias ociosas para economizar recursos quando a carga diminui. Esses mecanismos reativos permitem uma resposta rápida às alterações na carga de trabalho, garantindo um desempenho adequado e minimizando interrupções (LIMA, 2019).

Já os mecanismos proativos antecipam as mudanças na carga de trabalho com base em padrões históricos e tendências. Eles usam algoritmos de previsão para estimar a demanda futura e tomar ações preventivas com antecedência. Essas ações podem incluir o provisionamento antecipado de recursos adicionais ou a redistribuição da carga entre diferentes servidores. Ao antecipar as necessidades do sistema, os mecanismos proativos podem melhorar a eficiência operacional e evitar possíveis gargalos ou indisponibilidades (DAROLT; SOUZA; KOSLOVSKI, 2016).

A elasticidade pode ser utilizada para melhorar o desempenho de aplicações trazendo uma maior escalabilidade, flexibilidade e melhorias de custo. Além dos benefícios diretos, a elasticidade também facilita a implementação de padrões de computação paralela. Muitos desses padrões exigem a divisão de tarefas em unidades menores que podem ser executadas independentemente. Com a elasticidade, é possível alocar dinamicamente essas unidades de tarefa para diferentes recursos de processamento, otimizando a utilização de cada um deles. A elasticidade também simplifica a escalabilidade do sistema, permitindo que a capacidade seja facilmente aumentada ou reduzida conforme necessário. Nesse contexto, o foco será na aplicação da elasticidade nos padrões de estrutura de apoio para a implementação.

## 3.2 Frameworks para desenvolvimento de aplicações paralelas elásticas

Ferramentas e estruturas se fazem necessárias a fim de apoiar com eficiência a alocação dinâmica de recursos em infraestruturas paralelas. Para resolver este problema, existem soluções que permitem desenvolver aplicações elásticas de diversos modelos e visando diferentes arquiteturas. Alguns desses esforços serão descritos nesta Seção.

Para a implementação do padrão mestre/trabalhador, encontramos alguns *Frameworks* que possibilitam a sua implementação de forma elástica. O artigo apresentado por Rodrigues et al. (2017) aborda como tornar as aplicações mestre/trabalhador eficientes em nuvem. O trabalho propõe uma abordagem onde os limiares de atividade das aplicações mestre/trabalhador podem ser ajustados em tempo real com base em métricas de desempenho coletadas. Essa abordagem é chamada de *live thresholding* e permite que a aplicação se adapte dinamicamente às mudanças de carga. A arquitetura proposta consiste em três componentes principais: coleta de métricas de desempenho, análise de métricas e ajuste dinâmico dos limiares. A coleta de métricas envolve a monitoração contínua do desempenho dos componentes da aplicação. A análise das métricas permite identificar padrões de desempenho e determinar se os limiares precisam ser ajustados. O ajuste dinâmico dos limiares é realizado automaticamente para otimizar o desempenho da aplicação.

Righi et al. (2018) apresenta Helpar, um *Framework* apresenta Helpar, um *Framework* controlador de elasticidade híbrido para aplicativos paralelos iterativos, baseado em uma técnica chamada *live thresholding*, que implementa o padrão mestre/trabalhador. Desenvolvido para aplicações em *Cloud*, esse *Framework* possui um gerenciador capaz de prever um comportamento futuro para acionar ações elásticas e para alocar ou liberar novas VM's. Em relação ao aplicativo, há um mestre e uma coleção de processos trabalhadores onde, cada um, é instanciado por modelos de VM específicos: um para o mestre e outro para os trabalhadores. Cada nova aplicação requer a criação dos dois modelos mencionados.

Rodrigues et al. (2018) apresentam o *Framework SelfElastic*, um modelo de controle de elasticidade híbrido, pois aplica elasticidade reativa e proativa. Esse *Framework* possui um gerenciador que executa um *loop*no qual as métricas de monitoramento servem para otimizar e prever parâmetros internos, que podem disparar ou não ações de elasticidade para se adequar ao histórico de comportamento da aplicação. Os padrões de carga de trabalho são detectados e comparados aos dois últimos valores médios de carga, os quais foram calculados com base em dados de séries temporais monitorados, utilizando uma suavização exponencial simples. O gerenciador está separado em três módulos. O primeiro é o módulo sensor, que monitora a carga de CPU de cada VM periodicamente, passando

dados para o controlador depois. Por sua vez, o controlador aplica algoritmos para definir os valores de carga e os valores limites para disparar ações de elasticidade. Se a reorganização de recursos for necessária, o atuador executa ações de elasticidade usando a API da nuvem. Ao concluir as tarefas de todos os módulos, o gerenciador SelfElastic encerra o monitoramento de observação e aguarda o próximo ciclo de monitoramento. Os autores afirmam que SelfElastic é um *Framework* sem parâmetros, pois dispensa os programadores de definirem regras de elasticidade, condições ou limites, e não há a necessidade de uma execução prévia para a geração de dados, pois os dados, tanto de regras, condições ou limites, são gerados pelo *Framework* durante sua execução. SelfElastic também implementa o padrão mestre/trabalhador.

Rodrigues (2016) implementa o padrão mestre/trabalhador para prover o paralelismo com elasticidade. AutoElastic é um *middleware* que explora o paralelismo de dados para lidar com aplicativos de transmissão de mensagens executados em um ambiente de nuvem. Permite adaptar o número de unidades de processamento no início de cada iteração com base em limites definidos. A adaptação dinâmica é baseada no gerenciamento dinâmico de processos fornecido pelo MPI-2, e a alocação de novas máquinas virtuais é controlada por limites de alta ou baixa utilização de CPU. O lançamento de uma nova VM envolve automaticamente a execução de um novo processo trabalhador, a qual solicita uma conexão com o mestre automaticamente. Autoelastic permite que aplicações HPC usufruam da elasticidade de uma infraestrutura de nuvem sem a necessidade de modificações no código fonte e sem que a aplicação pare enquanto as ações de elasticidade são executadas.

Em Moreira (2018) o autor apresenta um modelo que provê elasticidade vertical assíncrona e hibrida, adicionando um módulo ao trabalho de Rodrigues (2016), qual seja, a elasticidade vertical. A abordagem proposta consiste em incrementar ao modelo do AutoElastic um módulo decisório, o qual busca executar uma série de testes para saber se existe a disponibilidade de hardware necessária para que a elasticidade vertical seja executada. Deve-se levar em consideração que a capacidade total de computação do nó hospedeiro é uma restrição para a implementação de elasticidade vertical. Por isso, antes de o VertElastic executar a elasticidade vertical, ele verifica a disponibilidade de hardware e, caso seja insuficiente, o módulo de decisão entende a limitação e permite que o fluxo de elasticidade horizontal seja iniciado. Sendo possível, o módulo inicia a execução da elasticidade vertical a fim de que a quantidade de recurso necessária para que uma aplicação finalize seu ciclo de execução, dentro de um SLA esperado, seja alcançada com a utilização da elasticidade vertical, sempre que o nó computacional permitir.

Ainda no contexto mestre/trabalhador, o *framework work queue* (RAJAN et al., 2013) visa a implementação de aplicações paralelas para ambientes de memória distribuída. Aplicativos desenvolvidos com *Work Queue* permitem que o número de trabalhadores possa

ser adaptado dinamicamente em tempo de execução para permitir o dimensionamento elástico. Os trabalhadores são implementados como arquivos executáveis que podem ser instanciados pelo usuário em diferentes máquinas. Quando executados, os trabalhadores se comunicam com o mestre, o qual coordena a execução da tarefa e a troca de dados conforme necessário.

Já para a implementação do padrão de paralelismo de laço, há alguns poucos *frameworks* para sua implementação de forma elástica. Em (DANELUTTO; TORQUATI, 2014), apresentam o FastFlow, implementado em C++, que tem por objetivo facilitar a implementação de algoritmos que utilizam o padrão paralelismo de laço. Os autores argumentam que, com a utilização do *Skeleton* apresentado, os programadores só precisam se preocupar em identificar os laços sem a necessidade de reescrever o corpo do laço. Os *Skeletons* são estruturas pré-definidas que encapsulam os detalhes de implementação do paralelismo e permitem que os programadores expressem o paralelismo em um nível mais alto de abstração. Os *Skeletons* podem ser combinados e compostos para construir algoritmos paralelos complexos de forma modular

Bu et al. (2012) apresentam Haloop, um modelo de programação e arquitetura para programas iterativos. Tal modelo oferece uma interface de programação para desenvolver aplicações iterativas de análise de dados e permite que os programadores reutilizem mapeadores e redutores existentes de aplicações Hadoop convencionais. De acordo com os autores, o agendador do HaLoop garante que as tarefas sejam atribuídas aos mesmos nós em múltiplas iterações, permitindo o uso de caches locais para melhorar o desempenho.

Para a implementação do padrão SPMD existem alguns *Frameworks* como o apresentado por Fatéma et al. (2017). Os autores exploram um novo modelo de equipe de microsserviços em nuvem baseado em máquinas virtuais (MVs) para aplicações SPMD de alto desempenho. O modelo proposto visa abordar os desafios enfrentados na execução de aplicações SPMD em ambientes em nuvem, especialmente quando se trata de escalabilidade e eficiência de recursos. A proposta é usar uma abordagem distribuída e altamente escalável baseada em MVs, permitindo que os recursos de computação sejam compartilhados e dimensionados dinamicamente para lidar com cargas de trabalho intensivas. O artigo descreve a arquitetura do modelo, incluindo a criação de “equipes"de microsserviços que colaboram entre si para executar aplicações SPMD. Essas equipes são compostas por: líder de equipe, time de trabalhadores, microsserviço de balanceamento, DF microsserviço e proxy provedor de microsserviços. A abordagem pode envolver a implantação de várias MVs em diferentes nós de uma infraestrutura em nuvem, e o uso de mecanismos de comunicação eficientes para troca de dados, além da sincronização entre os componentes da equipe. O objetivo principal é fornecer uma solução que seja capaz de lidar com grandes volumes de dados, distribuir o processamento em paralelo e oferecer elasticidade, adaptando os recursos computacionais às necessidades da aplicação.

O trabalho de Martín et al. (2015) se concentra em programas SPMD iterativos. Eles apresentam o Flex-MPI, uma biblioteca que permite que aplicativos MPI criem e removam processos de forma dinâmica em tempo de execução para otimizar seu desempenho, aumentando a eficiência e o custo-benefício. Para adicionar novos processadores, o Flex MPI envia uma solicitação ao sistema gerenciador de recursos do cluster (RMS), que envia de volta uma lista dos nós adicionais que podem ser usados. Em seguida, um módulo de gerenciamento coordena a adição e remoção de processos MPI. Novos processos são criados se houver processadores ociosos no sistema ou se o desempenho atual da aplicação não satisfizer o objetivo. Flex-MPI também fornece um mecanismo de redistribuição de dados que é acionado quando ocorre uma reconfiguração. O *Framework* implementa as operações necessárias para o desenvolvimento de aplicações adaptativas SPMD (alocação dinâmica de recursos e redistribuição de dados).

Uma proposta similar, o Elastic MPI, é apresentada por (MO-HELLENBRAND, 2019). Na citada estrutura, um gerenciador de recursos é responsável por iniciar e executar tarefas paralelas, tomando decisões para atribuição de recursos, além de iniciar o tempo de execução dos recursos (através da interação RMS do cluster). O foco são os modelos de execução SPMD e mestre/trabalhador.

Quando nos referimos ao modelo *fork-Join*, OpenMP vem em mente. Um programa OpenMP sempre começa a execução como um *thread*único (o *thread* mestre), e quando uma diretiva paralela é encontrada, a execução se bifurca e a região paralela é executada por uma equipe de *threads*. Quando a região paralela é encerrada, os *threads* da equipe juntam-se novamente em uma barreira implícita e o *thread* mestre continua a execução. A partir desse modelo de execução, surgem as possibilidades de explorar a adaptabilidade. Por exemplo, (GALANTE; BONA, 2014) desenvolveram uma versão elástica do OpenMP na qual a diretiva paralela foi modificada para adaptar o número de VCPUs de acordo com o número de *threads* bifurcadas na região paralela. Quando a região paralela termina, os *threads* são unidos, o número de VCPUs é redefinido para configuração anterior.

Os autores Nguyen et al. (2020) abordam em seu artigo o desenvolvimento de um modelo de previsão de latência de *fork-join* para aplicações intensivas em dados. O objetivo principal do modelo proposto é fornecer uma estimativa precisa e eficiente do tempo de execução de tarefas em aplicações que envolvem o processamento de grandes volumes de dados. A previsão de latência é uma informação valiosa para otimizar o desempenho e a eficiência de tais aplicações, permitindo o dimensionamento adequado dos recursos de computação e o agendamento eficiente de tarefas. O termo *Black-box*, utilizado pelos autores, indica que o modelo é projetado para ser aplicável a uma variedade de aplicações de processamento de dados, independentemente dos detalhes internos de implementação. O modelo se baseia na análise de características observáveis das tarefas como tamanho dos dados, complexidade computacional e padrões de acesso para fazer previsões de latência.

O modelo proposto envolve a coleta de dados de execuções anteriores das aplicações e a utilização de técnicas de aprendizado de máquina para treinar um modelo preditivo. Esse modelo pode então ser usado para prever a latência de tarefas futuras com base em suas características observáveis.

Zhao, Gao e Li (2022) implementaram uma solução para realizar a alocação dinâmica de recursos de nuvem durante a operação de um programa OpenMP baseado em um controlador difuso.

Abordando o aspecto recursivo do modelo *Fork-Join*, Wrzesinska et al. (2005) apresentam um sistema que suporta capacidade de adaptação para aplicações de divisão e conquista. Os mecanismos são implementados usando Satin, uma divisão e conquista baseada em Java, projetado para ambientes de rede. Em Satin, o trabalho é distribuído entre os processadores por roubo de trabalho, ou seja, ao ser adicionada uma nova máquina os trabalhos são roubados de outras máquinas. Satin também pode ajustar automaticamente os aplicativos para responder a alterações nas condições dos recursos, como CPUs sobrecarregadas ou links de comunicação lentos. A tabela 1 contém os *Frameworks* apresentados.

| **Padrão** | ***Frameworks*** | **Tipo de Elasticidade** |
|---|---|---|
| Mestre/Trabalhador | (RODRIGUES et al., 2017) | Vertical e Horizontal |
| | (RIGHI et al., 2018) | Vertical e Horizontal |
| | (RODRIGUES et al., 2018) | Vertical e Horizontal |
| | (MOREIRA, 2018) | Horizontal |
| | (RODRIGUES, 2016) | Vertical |
| | (RAJAN et al., 2013) | Horizontal |
| Paralelismo de laço | (DANELUTTO; TORQUATI, 2014) | Horizontal |
| | (HOUZEAUX et al., 2022) | Horizontal |
| SPMD | (FATÉMA et al., 2017) | Horizontal e Vertical |
| | (MARTÍN et al., 2015) | Horizontal e Vertical |
| | (MO-HELLENBRAND, 2019) | Horizontal e Vertical |
| Fork/Join | (GALANTE; BONA, 2014) | Vertical |
| | (NGUYEN et al., 2020) | Horizontal e Vertical |
| | (ZHAO; GAO; LI, 2022) | Horizontal e Vertical |

Tabela 1 – *Frameworks* para implementação de padrões paralelos.

# 4 Modelo unificado de padrões elásticos para programação paralela

O objetivo deste capítulo é realizar a intersecção dos modelos apresentados no Capítulo 2, propondo a unificação dos padrões de programação paralela apresentados pelos autores lá citados, visando auxiliar a implementação de aplicações paralelas a partir de uma definição de padrões, bem como fases de projeto de desenvolvimento para essas aplicações.

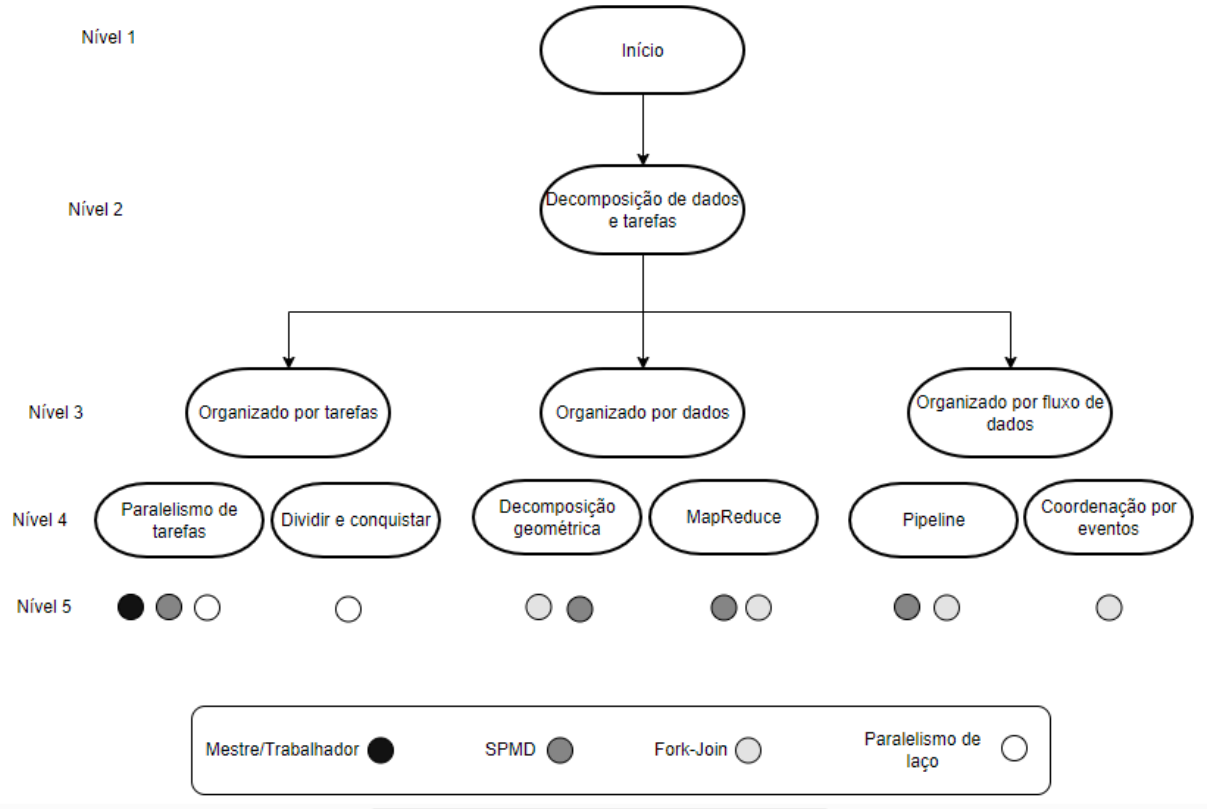
## 4.1 Modelo Unificado para o projeto de programas paralelos

Conforme apresentado no Capítulo 2, Mattson, Sanders e Massingill (2004) propõem fases de projeto que possibilitam o desenvolvimento de aplicações paralelas, separando essas fases em quatro espaços de projeto onde o foco dessas fases é a arquitetura da aplicação, trazendo uma orientação para o desenvolvimento das aplicações a partir do zero. Nessas fases, toda a estrutura da aplicação é pensada e definida. No modelo proposto, algumas das fases são utilizadas e combinadas para que seja possível analisar a concorrência, avaliar a forma de organização do algoritmo bem como a estrutura do mesmo, além da sua estrutura de apoio a fim de definir padrões, questões de arquitetura de *hardware* e balanceamento de carga. A estrutura da árvore presente na Figura 12, com as fases 2, 3, 4 e 5, foi proposta com base no modelo de Mattson, Sanders e Massingill (2004). Já o padrão mapreduce apresentado na árvore foi trazido como contribuição da obra de McCool, Reinders e Robison (2012), assim como a descrição dos demais padrões que são intersecção da obra dos autores.

A árvore de decisão apresentada Figura 12 tem como objetivo orientar a escolha dos padrões a serem implementados. No primeiro nível tem-se a forma como o algoritmo é estruturado, no segundo tem-se os padrões de estruturas de algoritmos propostos, e no terceiro estão os padrões de implementação desses algoritmos, sugeridos por Mattson, Sanders e Massingill (2004).

Para iniciar a análise de um algoritmo tem-se como entrada o código de qualquer aplicação, independente da linguagem de programação utilizada. Na primeira fase é feita uma análise do problema, avaliando a forma como o algoritmo está organizado, se está organizado por tarefas ou se a organização é feita por decomposição de dados ou por fluxo de dados. Embora esteja se tratando a organização dos algoritmos de forma separada para facilitar a escolha do padrão a ser aplicado, dificilmente encontram-se algoritmos que sejam

Figura 12 – Árvore de decisão proposta.




unicamente organizados por tarefas ou organizados por decomposição de dados. Isso porque mesmo que o algoritmo seja organizado por tarefas, é necessário tratar as dependências de dados e o fluxo da execução das tarefas. Diante disso, mesmo que o algoritmo seja majoritariamente organizado por tarefas, por exemplo, ainda se faz necessário avaliar a decomposição dos dados utilizados por essas tarefas.

Após realizar a avaliação a fim de determinar se o algoritmo é organizado majoritariamente por dados, tarefas ou fluxo de dados, avança-se para o terceiro nível da árvore, onde é necessário avaliar outros pontos. Por exemplo, se a aplicação apresentar uma estrutura organizada por tarefas, ao avançar para o nível 3 da árvore de decisão, deve-se avaliar os seguintes pontos:

- Se as tarefas são geradas de forma estática ou dinâmica;
- Se o número de tarefas é conhecido no início da computação do algoritmo ou geradas em tempo de execução;
- Qual a ordem de execução destas tarefas?
- Existe agrupamento entre as tarefas?

Por outro lado, ao termos um algoritmo majoritariamente organizado por decomposição de dados, avaliam-se os seguintes pontos:

- Há dependência entre os dados durante a execução do algoritmo, ou os dados podem ser manipulados de forma independente?
- Avaliar a decomposição destes dados;
- Quais tarefas compartilham estes dados?
- É possível manter cópias locais dos dados?
- Como será a escrita e leitura destes dados pelas tarefas? Será necessário implementar proteção?

Caso o algoritmo seja organizado por fluxo de dados, deve-se atentar a:

- Há dependências de fluxo de execução?
- As tarefas são executadas de forma independente a partir de algum evento gerado pelo algoritmo?
- O fluxo de dados é regular, unilateral e estático?
- O fluxo de dados é irregular, dinâmico e imprevisível?

A partir desta análise é possível determinar qual estrutura de algoritmo se aplica melhor. A estrutura de algoritmo propõe a utilização de alguns padrões, tais como: Paralelismo de tarefas, Dividir e conquistar, Decomposição geométrica, MapReduce, Pipeline e Coordenação baseada em eventos. Tais padrões estão presentes no nível 4 da árvore de decisão proposta e foram descritos no capítulo anterior. Eles visam implementar a organização do algoritmo com base nas questões apresentadas anteriormente e definem como o algoritmo está estruturado.

A fim de definir o melhor padrão da estrutura de apoio, deve-se avaliar todas as questões já citadas, além de considerar também os mecanismos de implementação onde são tratadas questões de gerenciamento de *threads* e processos, sincronização, comunicação, estrutura de memória e balanceamento de carga. Embora os padrões tentem abstrair ao máximo questões de *hardware*, é necessário realizar uma avaliação para definir se paralelizar um problema é realmente viável. Nas próximas Seções serão apresentadas formas de realizar a análise do algoritmo para definição do melhor padrão a ser aplicado

## 4.2 Elasticidade aplicada aos padrões de estrutura de apoio

A evolução das arquiteturas paralelas aponta para ambientes dinâmicos onde a disponibilidade de recursos ou as configurações podem variar durante a execução da aplicação. Esta propriedade é comum em arquiteturas dinâmicas como computadores em grade e nuvens, mas também pode ser adaptada e explorada em arquiteturas com múltiplos processadores ou núcleos. Para explorar plenamente essa propriedade não só a infraestrutura dinâmica é necessária, mas também necessita-se de aplicações elásticas.

Compreender e adotar a elasticidade e seus mecanismos será, portanto, essencial à medida que se avança em direção a esta próxima geração de arquiteturas paralelas. Mesmo considerando os esforços no desenvolvimento de software para alavancar o poder computacional das máquinas paralelas, ainda resta muito a ser feito para aproveitar efetivamente a elasticidade nesses sistemas e também para obter uma integração perfeita do código do aplicativo e dos recursos de hardware. Diante desse cenário, propõem-se várias iniciativas para estender os padrões propostos e para ajudar os desenvolvedores a projetar e implementar aplicações paralelas elásticas.

Aplicações elásticas são capazes de se adaptar às mudanças no número de unidades de processamento durante a execução, podendo tomar decisões por conta própria (aplicações evolutivas) ou necessitar do suporte de um sistema de gerenciamento de recursos externo ou sistema operacional (aplicações maleáveis) (ESTELLÉS et al., 2022). Do ponto de vista da aplicação, isso pode ser realizado adicionando/removendo ou duplicando tarefas (implementadas em processos ou *threads*) ou migrando essas tarefas para outra unidade de processamento com diferentes capacidades. Do ponto de vista arquitetônico, deve-se fornecer mecanismos (gerenciador de recursos ou tempo de execução) para a alocação dinâmica de unidades de processamento, que podem ser um núcleo de processador, um nó de *cluster*, uma máquina virtual ou contêiner na nuvem. A Figura 13 mostra como funcionaria a definição da estrutura de algoritmo elástico, com a adição de novas tarefas, a duplicação das tarefas ou migração das mesmas. Ao definir o padrão da estrutura do algoritmo, se faz necessário definir a estrutura de apoio para a implementação do algoritmo.

A Figura 14 mostra exemplos de como a elasticidade pode ser efetivamente aplicada nos padrões de estrutura de apoio, aumentando o número de processadores/core, migrando tarefas para outras unidades de processamento com mais poder computacional, adicionando nós em *clusters* ou alocando VMs adicionais, contêineres ou outros recursos para as tarefas que são criadas pela aplicação

As estruturas de algoritmo apresentadas são utilizadas para a implementação da aplicação, já os padrões da estrutura de apoio são utilizados para definir a forma que a aplicação irá se comunicar com o hardware utilizado.

Figura 13 – Elasticidade na perspectiva da aplicação.

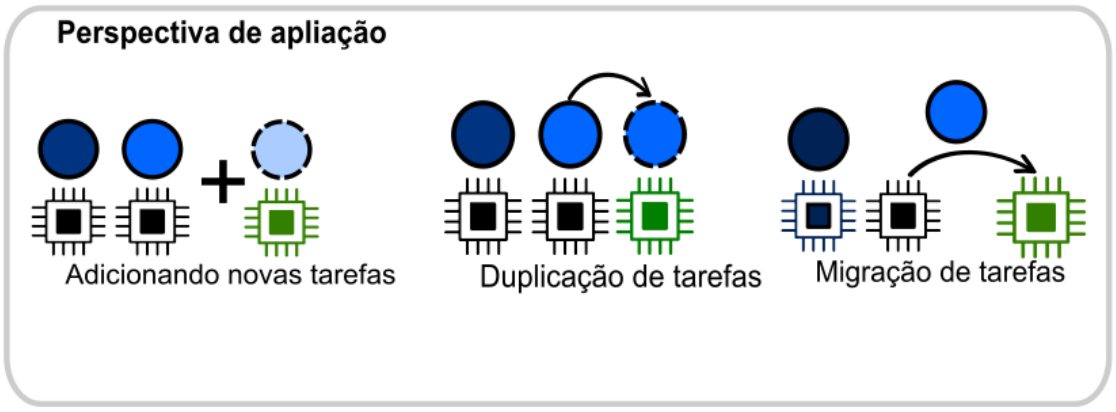
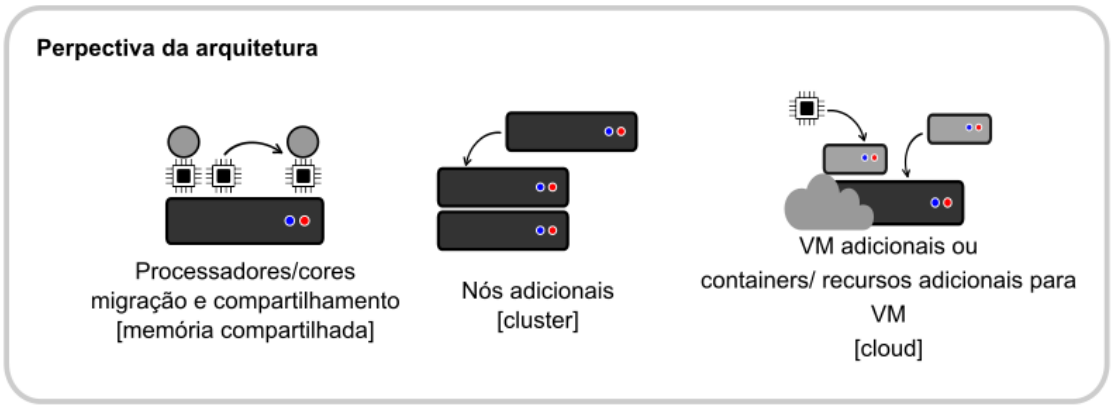



Figura 14 – Elasticidade na perspectiva dos padrões de estrutura de apoio.




Utilizando o modelo proposto na 4.1, ao se definir o tipo do mecanismo de implementação e também se o sistema será de memória distribuída ou compartilhada, é possível definir o tipo de elasticidade que será utilizada, se vertical ou horizontal, ou ambas. Grande parte dos padrões apresentados podem implementar os dois tipos de elasticidade, como o padrão mestre/trabalhador. Em um sistema de memória compartilhada, pode-se utilizar a elasticidade vertical para aumentar o número de CPU's alocadas ou a quantidade de memória. Para um sistema de memória distribuída, pode-se utilizar a elasticidade horizontal alocando/desalocando novos nós de processamento. A elasticidade deve ser aplicada após a escolha dos padrões de estrutura de apoio, pois é a estrutura de apoio que irá implementar de fato a elasticidade.

A estrutura de apoio descreve abordagens de estruturação do código-fonte da aplicação, pensando já nos mecanismos que serão utilizados para implementação efetiva do algoritmo, se serão *threads* ou processos, como será a sincronização e a comunicação e também como será realizado o balanceamento de carga. Os padrões que fazem parte da estrutura de apoio são: mestre/trabalhador, SPMD, *Fork/Join* e paralelismo de laço. Os demais padrões, abordados em estrutura de algoritmo, fazem uso da elasticidade através da implementação da mesma em padrões de estrutura de apoio. Para que tal afirmação

fique mais clara, usou-se como exemplo o padrão de estrutura de algoritmo paralelismo de tarefas, que é como o algoritmo está estruturado, ou seja, está organizado por tarefas geradas de forma independente, e que pode ser implementado por três padrões de estrutura de apoio, quais sejam, mestre/trabalhador, SPMD e paralelismo de laço, sendo que a elasticidade será aplicada nesses padrões.

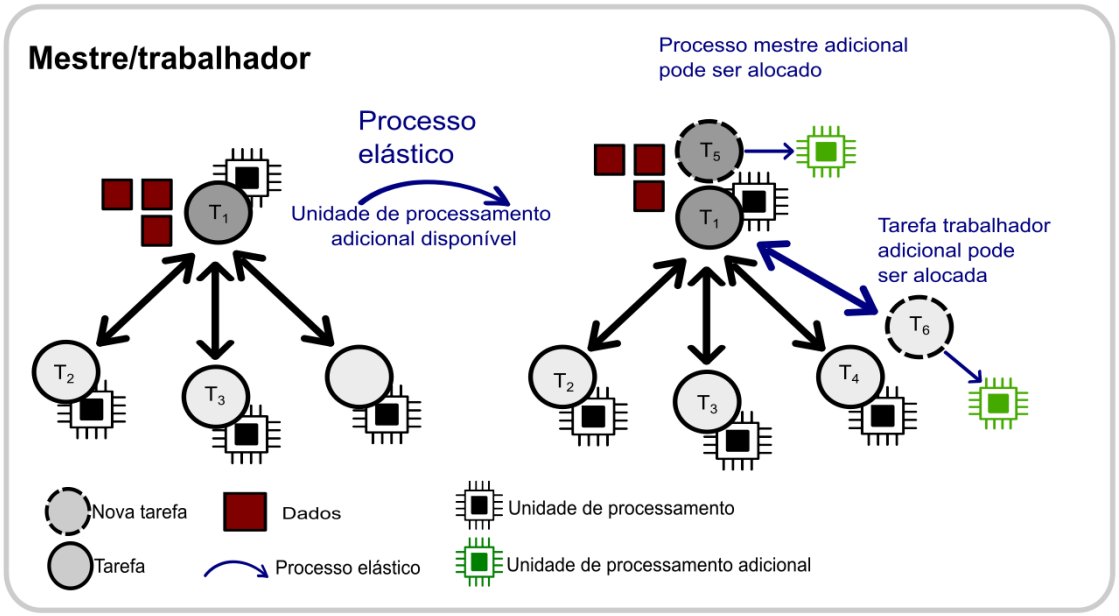
### 4.2.1 Padrão mestre/trabalhador

Nesse padrão tem-se dois papéis: o mestre e os trabalhadores. O mestre é responsável pela coordenação e distribuição de tarefas, já os trabalhadores executam as atividades atribuídas de maneira assíncrona. Nesse padrão, o nó mestre pode ser um grande gargalo ao se tornar o ponto único de falha, limitando a capacidade da aplicação lidar com um grande volume de solicitações. Se o nó mestre falhar ou ficar sobrecarregado, todo o sistema pode sofrer interrupção ou degradação no desempenho. A elasticidade pode aprimorar o padrão mestre/trabalhador possibilitando oferecer escalabilidade dinâmica, distribuição eficiente de tarefas, adaptação a falhas e eventos imprevistos e melhoria na segurança do sistema. Ao adotar uma abordagem elástica, é possível otimizar o desempenho, melhorar a confiabilidade e até mesmo reduzir os impactos de possíveis gargalos inerentes ao padrão mestre/trabalhador, considerando as melhorias que podem ser trazidas para os dois papéis desse padrão (DUBREUIL; GAGNÉ; PARIZEAU, 2006).

Dentre as melhorias está a gestão dinâmica de recursos, onde o mestre pode adicionar ou remover trabalhadores conforme necessário para atender às demandas variáveis do sistema, maximizando a utilização de recursos, garantindo que o sistema possa lidar com qualquer volume de carga e proporcionando escalabilidade dinâmica. Tal processo é exemplificado na Figura 15, onde tem-se a T1 como mestre, T2, T3 e T4 como trabalhadores, quando há a necessidade de adicionar recursos, o nó mestre aloca nova tarefa T6, alocando nova unidade de processamento. É possível também alocar uma nova tarefa mestre, na figura chamada de T5, alocando nova unidade de processamento (RAJAN et al., 2011).

Considerando o papel do mestre, podemos ter dois tipos de elasticidade aplicada, quais sejam, elasticidade horizontal e elasticidade vertical. O mestre que implementa elasticidade horizontal será definido neste trabalho como padrão ElasticHor, já o mestre que implementa a elasticidade vertical será definido com o padrão ElasticVer. Para aplicações que implementem a elasticidade horizontal e vertical combinadas, tem-se o mestre ElasticVH. Para aplicação da elasticidade no papel dos trabalhadores, defini-se os trabalhadores ElasticVert, que implementaria a elasticidade vertical. Trabalhadores ElasticHor, que implementariam a elasticidade horizontal e o trabalhadores ElasticVH que implementaria a elasticidade vertical e horizontal combinada.

Figura 15 – Elasticidade horizontal Mestre/trabalhador.




Ao definir a arquitetura onde a aplicação será executada, pode-se definir o tipo de elasticidade que será implementada, podendo desta forma escolher o padrão que melhor se aplica.

Se a aplicação for executada em um sistema com memória compartilhada, o tipo de elasticidade indicada seria a elasticidade vertical e, em tal contexto, teríamos os padrões elásticos mestre ElasticVert e trabalhador ElasticVert, já que em sistemas de memória distribuída várias CPU's utilizam um mesmo endereçamento de memória física. Os nós mestre ElasticVert e trabalhadores ElasticVert podem ser escalados com recursos para melhorar o desempenho da aplicação. Também é possível implementar a elasticidade horizontal, utilizando os padrões mestre ElasticHor e trabalhador ElasticHor, mas existe a limitação do número de CPU's disponíveis. Nesse caso, é necessário avaliar se o número de tarefas trabalhadoras gerada no início da execução é menor do que o número de CPU's disponíveis no sistema. Ainda assim, durante a execução, caso a elasticidade horizontal seja implementada sozinha, pode haver um gargalo quando finalizar o número de CPU's disponíveis e o sistema necessite escalar novamente.

No nó mestre ElasticVert, o aumento da capacidade de processamento ao adicionar mais recursos, como CPU e RAM, melhora sua capacidade de processar e gerenciar tarefas de maneira eficiente. Isso pode ser implementado criando um monitoramento dos recursos do nó mestre ElastiVert, e quando esses recursos ultrapassarem um limite pré definido, a ação de elasticidade é acionada. Por exemplo, pode-se definir que, caso a utilização de memória atinja 80% do valor total do sistema, a ação de elasticidade seja acionada para que seja alocada mais memória para aquele nó e, caso a utilização caia para 50%, aciona-se novamente o mecanismo de elasticidade que libera a memória.

A elasticidade vertical permite dimensionar verticalmente os trabalhadores para processar mais dados simultaneamente, acelerando o processamento geral. Imagine um sistema de processamento de dados em lote em que um mestre coordena vários trabalhadores para executar cálculos complexos. Caso haja sobrecarga de utilização de recursos nesses nós trabalhadores, o mecanismo de elasticidade pode ser acionado para alocar mais recursos para os nós sobrecarregados. O limite de utilização pode ser definido no início da computação, pode ser implementado de forma proativa ou mesmo de forma mista (ROSSI, 2020).

Em cenários onde os trabalhadores executam tarefas demoradas como renderização de vídeo ou simulações científicas, a adição de recursos por meio da elasticidade vertical acelera o tempo de conclusão dessas tarefas, já que ao aumentar recursos de CPU e memória esses nós podem terminar a computação de forma mais rápida. Por exemplo, um estúdio de animação pode verticalizar seus trabalhadores de renderização para reduzir o tempo necessário para gerar uma cena finalizada. Neste mesmo contexto, a aplicação de elasticidade vertical do mestre permite lidar com a crescente complexidade das tarefas. Isso é particularmente valioso quando essas tarefas requerem grande poder de processamento e memória para execução (CABALLER et al., 2012).

Em sistemas que enfrentam variações na carga de trabalho como, por exemplo, sites de comércio eletrônico durante feriados ou eventos especiais, a elasticidade vertical pode ser aplicada aos trabalhadores para aumentar a capacidade de resposta. Quando o tráfego aumenta rapidamente, a infraestrutura pode ser expandida verticalmente para acomodar o aumento de demanda, garantindo que os clientes não enfrentem lentidão ou erros devido à sobrecarga. Isso pode ser feito adicionando recursos computacionais a estes nós. Em cenários onde a segurança e a conformidade são fundamentais, o mestre pode ser responsável por funções críticas como autenticação e autorização. Ao dimensionar verticalmente o mestre, aumentando recursos como CPU ou memória, pode-se aprimorar sua capacidade de lidar com cargas de trabalho intensivas de segurança como verificações de integridade ou análises de vulnerabilidades. Isso garante que o sistema esteja em conformidade com regulamentações rigorosas. Em sistemas que processam grandes volumes de dados como bancos de dados mestre/trabalhador, a elasticidade vertical do mestre, alocando mais memória ou CPU, pode melhorar a eficiência na indexação, consulta e replicação de dados. Se a aplicação for executada em um sistema com memória distribuída, o tipo de elasticidade indicada seria a elasticidade horizontal. Em tal contexto teria-se os padrões elásticos mestre ElasticHor e trabalhador ElasticHor, já que em sistemas de memória compartilhada é possível alocar novos nós de processamento para escalar o ambiente . Os nós mestre ElasticHor e trabalhador ElasticHor podem ser escalados com recursos para melhorar o desempenho da aplicação. Também é possível implementar a elasticidade vertical nesse tipo de sistema utilizando os padrões mestre ElasticVert e

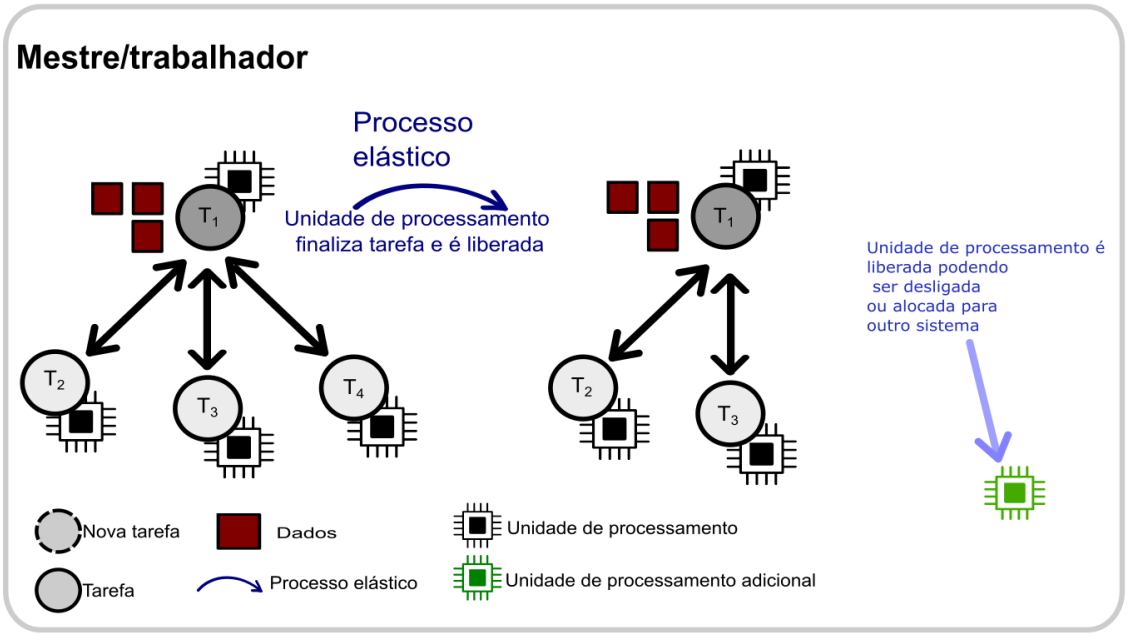
trabalhador ElasticVert. A elasticidade combinada também pode ser implementada neste tipo de sistema utilizando os padrões mestre ElasticVH e trabalhador ElasticVH. O mestre ElasticHor implementaria a elasticidade horizontal no papel do mestre.

Como o mestre tem o papel de dividir e distribuir as tarefas entre os nós trabalhadores, isso pode gerar para o nó mestre uma carga excessiva de computação caso haja muitas tarefas para distribuir e um grande número de trabalhadores para gerenciar. Nesse cenário, o mestre ElasticHor, implementando elasticidade horizontal, pode trazer inúmeros benefícios. Nesse caso, o mestre ElasticHor pode alocar um novo nó mestre que possa dividir e distribuir as tarefas, além de gerenciar os nós trabalhadores em relação à carga e tempo de execução, verificando periodicamente o tempo de execução e o que falta ser computado para, caso necessário, disparar ações de elasticidade a fim de alocar novos nós trabalhadores. Nesse caso, se faz necessário que o nó mestre divida e distribua tarefas aos novos nós alocados. Caso os nós trabalhadores tenham terminado a execução das suas tarefas, o nó mestre libera os nós de computação alocados para esses trabalhadores. A utilização do mestre ElasicHor proporciona ao padrão confiabilidade por tornar possível a tolerância a falhas pois, no caso de haver falha de um dos nós mestre, um novo nó é alocado, e o novo nó mestre consegue redistribuir as tarefas para novos nós trabalhadores. Uma das formas de se implementar a tolerância a falhas neste padrão é adicionar na execução do nó mestre uma verificação da saúde dos nós trabalhadores que estão em execução. Tal verificação pode ser executada dentro de um intervalo de tempo pré definido, de acordo com a tolerância de tempo que a aplicação pode ficar indisponível no caso de falha. Dos *frameworks* apresentados no 3.2 que permitem a implementação deste padrão em memória distribuída tem-se (RIGHI et al., 2018), (RODRIGUES, 2016) e (RAJAN et al., 2013).

Outro ponto de melhoria que a utilização do mestre ElasticHor pode proporcionar é a adaptação à flutuações na demanda, adicionando trabalhadores temporários para lidar com picos de carga e removê-los quando a demanda diminui, sendo que esse aspecto pode ser implementado definindo um limite de solicitações avaliado pelo mestre. Para fins de exemplificação, em um website que recebe um pico de acessos de usuários, o nó mestre pode manter um limite de acessos e definir que, a partir daquele limite, o número de nós trabalhadores será escalado, assim como serão liberados quando o pico de utilização diminuir. Esse limite pode ser reativo, definido antes do início da execução, mas pode ser também implementado de forma proativa, onde um sistema externo, um *middleware*, por exemplo, fica responsável por avaliar a execução da aplicação para prever picos de utilização, implementando um controle de elasticidade proativo. Essa melhoria traz economia, uma vez que diminui a ociosidade dos nós. A capacidade de reduzir o número de trabalhadores durante períodos de baixa demanda de tarefas sendo executadas permite economizar custos operacionais para aplicações hospedadas em nuvem. Um exemplo de

como a redução pode ser aplicada nos recursos está na Figura 16, onde tem-se a T1 como tarefa mestre e as tarefas T2, T3 e T4 como tarefas trabalhadoras. Quando a tarefa T4 encerra seu processamento, a unidade de processamento pode ser liberada.

Figura 16 – Elasticidade horizontal Mestre/trabalhador.




A elasticidade horizontal, utilizando trabalhador ElasticHor, proporciona também um balanceamento de carga eficiente ao distribuir de forma uniforme as tarefas entre esses trabalhadores para evitar sobrecarga em qualquer nó específico, pois permite a alocação de novos nós de processamento quando há a necessidade (RAJAN et al., 2011). Com o fim de implementar esta melhoria da elasticidade horizontal, pode ser criada uma rotina no nó mestre que avalie, de tempos em tempos, a carga de recursos utilizada em cada nó trabalhador e, a partir de uma métrica específica, adicionar novos nós trabalhadores ou até mesmo liberá-los. Novamente aqui, tal métrica pode ser definida no início da computação ou implementada de maneira proativa.

A adição ou remoção de trabalhadores para atualizações ou manutenções sem interromper o serviço é outra melhoria proporcionada pela elasticidade horizontal. Neste caso, no momento que houver atualização ou manutenção nos nós de um sistema mestre/-trabalhador, outros nós podem ser alocados temporariamente para realizar a computação, sendo liberados assim que a manutenção finalizar e os nós anteriores estiverem disponíveis.

Essas melhorias proporcionadas pela elasticidade horizontal no padrão mestre/trabalhador são essenciais para sistemas modernos que precisam se adaptar a cargas variáveis, garantir alta disponibilidade e serem eficientes em termos de custos operacionais.

Em sistemas que enfrentam variações na carga de trabalho, como sites de comércio eletrônico durante feriados ou eventos especiais, a elasticidade vertical pode ser aplicada

aos trabalhadores para aumentar a capacidade de resposta. Quando o tráfego aumenta rapidamente, a infraestrutura pode ser expandida verticalmente para acomodar o aumento de demanda, garantindo que os clientes não enfrentem lentidão ou erros devido à sobrecarga. Isso pode ser feito adicionando recursos computacionais a estes nós.

As vezes, a limitação de desempenho está em um único trabalhador responsável por uma parte crítica do processamento. A elasticidade vertical permite otimizar esse trabalhador, garantindo que ele tenha recursos suficientes para realizar sua tarefa de maneira eficiente. Para isso, o nó mestre pode ter uma rotina que monitore os nós trabalhadores responsáveis pelos processamentos críticos, acionando os mecanismos de elasticidade sempre que necessário, seja para alocar ou liberar recursos. A elasticidade vertical dos nós trabalhadores também pode diminuir a latência na comunicação com o mestre, pois recursos de rede podem ser adicionados para atingir esse objetivo (NEERAJ, 2015).

Quando os trabalhadores fazem solicitações frequentes ao mestre, uma instância verticalizada pode responder mais rapidamente, reduzindo a latência nas comunicações. Para atingir isso, o nó mestre pode ter seus recursos aumentados. Esse processo é vital em sistemas em tempo real, como jogos online, onde o mestre precisa coordenar ações instantaneamente. Ao verticalizar o mestre, a latência é minimizada, proporcionando uma experiência mais fluída aos usuários.

A utilização da elasticidade combinada seria possível em sistemas como nuvens computacionais, onde os recursos computacionais podem ser utilizados e requisitados do provedor conforme a demanda, e o pagamento é feito conforme a utilização. Nesse cenário, é possível implementar os padrões mestre ElasticVH, que implementaria a avaliação para alocar recursos verticalmente de acordo com a necessidade da aplicação e a especificação do desenvolvedor, além de alocar recursos horizontalmente quando necessário. O mesmo se aplicaria para a utilização do trabalhador ElasticVH, assim como é possível definir para que só um dos papéis escale verticalmente utilizando um dos padrões de mestre ou trabalhador. Para implementação da elasticidade combinada, temos os *Frameworks* apresentados por (RODRIGUES et al., 2017), (RODRIGUES et al., 2018) e (MOREIRA, 2018).

Quando pensamos no tipo de política para implementação da elasticidade, temos as políticas manuais e automáticas. Para o padrão mestre/trabalhador, a política manual pode trazer algumas melhorias como, por exemplo, o controle preciso que permite aos administradores humanos ajustar o número de trabalhadores de forma específica e detalhada, respondendo a padrões de uso e demandas específicas (KEHRER; BLOCHINGER, 2019b). O administrador aumenta manualmente o número de trabalhadores durante um evento de venda online, por exemplo, prevendo um aumento no tráfego do site. Outra melhoria

seria a adaptação a cenários complexos, onde os padrões de uso são difíceis de prever e a intervenção humana é necessária para tomar decisões contextuais. Um exemplo disso seria um sistema de análise financeira, onde um analista humano ajusta manualmente a capacidade de processamento com base em eventos econômicos imprevistos.

A política manual também traz uma garantia de segurança ao oferecer controle total sobre as operações, garantindo que mudanças no número de trabalhadores não comprometam a segurança ou a integridade dos dados. Em um banco de dados que lida com transações financeiras críticas, esta política permite que os administradores controlem manualmente o número de servidores em operação.

Já a política de elasticidade automática pode proporcionar escalabilidade dinâmica ao permitir a adição ou remoção automatizada de trabalhadores com base em métricas de desempenho como carga da CPU ou tráfego de rede. Um exemplo seria o serviço de hospedagem de sites na nuvem que escala automaticamente o número de servidores com base na quantidade de tráfego recebido, garantindo uma experiência de usuário estável.

A política automática gera uma resposta imediata às flutuações na demanda, garantindo que o sistema possa lidar com picos de tráfego sem atrasos e também com uma economia de custos, permitindo a redução automática de trabalhadores durante períodos de baixa demanda, economizando recursos e custos operacionais (RODRIGUES, 2016).

A elasticidade manual oferece controle detalhado e é valiosa para situações onde intervenção humana é necessária. Ao mesmo tempo, a elasticidade automática é crucial para cenários dinâmicos, onde respostas rápidas e eficiência são fundamentais. A escolha entre elas geralmente depende dos requisitos específicos do sistema e dos objetivos de desempenho e economia.

Quando se considera o controle da elasticidade para o padrão mestre/trabalhador tem-se algumas melhorias como, por exemplo, resposta imediata às mudanças de carga quando o controle implementado é o reativo. Nesses casos, o sistema pode se adaptar instantaneamente a picos inesperados ou quedas na demanda, escalando ou reduzindo trabalhadores conforme a necessidade em tempo real.

Já o controle proativo traz melhorias na antecipação de picos de demanda como, por exemplo, um sistema de reserva de passagens aéreas que antecipa um aumento na demanda durante as férias de verão e proativamente adiciona trabalhadores para lidar com a demanda esperada. Além disso, o controle proativo auxilia com a otimização baseada em análise preditiva para identificar padrões e ajustar a capacidade com antecedência, melhorando a eficiência operacional (LIMA, 2019). E tudo isso gera uma economia de custos antecipada evitando gastos excessivos escalando trabalhadores com antecedência durante períodos de alta demanda e economizando custos operacionais a longo prazo. A tolerância a falhas, aprimorada por antecipação, também é uma melhoria proporcionada pelo controle proativo,

utilizando-se de análise preditiva para antecipar falhas em trabalhadores e proativamente substituindo-los para garantir a continuidade do serviço (DAROLT; SOUZA; KOSLOVSKI, 2016)

### 4.2.2 Padrão SPMD

A programação em SPMD pode ser complexa e exigir um conhecimento aprofundado da arquitetura do sistema e das características da aplicação. A divisão eficiente do trabalho, a distribuição dos dados e a coordenação das operações requerem uma cuidadosa consideração e podem exigir um esforço significativo de programação.

A elasticidade pode colaborar na melhoria do padrão SPMD, permitindo a escalabilidade, o balanceamento de carga, a eficiência na utilização de recursos e a adaptação a variações de carga. A capacidade de ajustar dinamicamente os recursos segundo as demandas da aplicação ajuda a otimizar o desempenho e a eficiência do sistema SPMD.

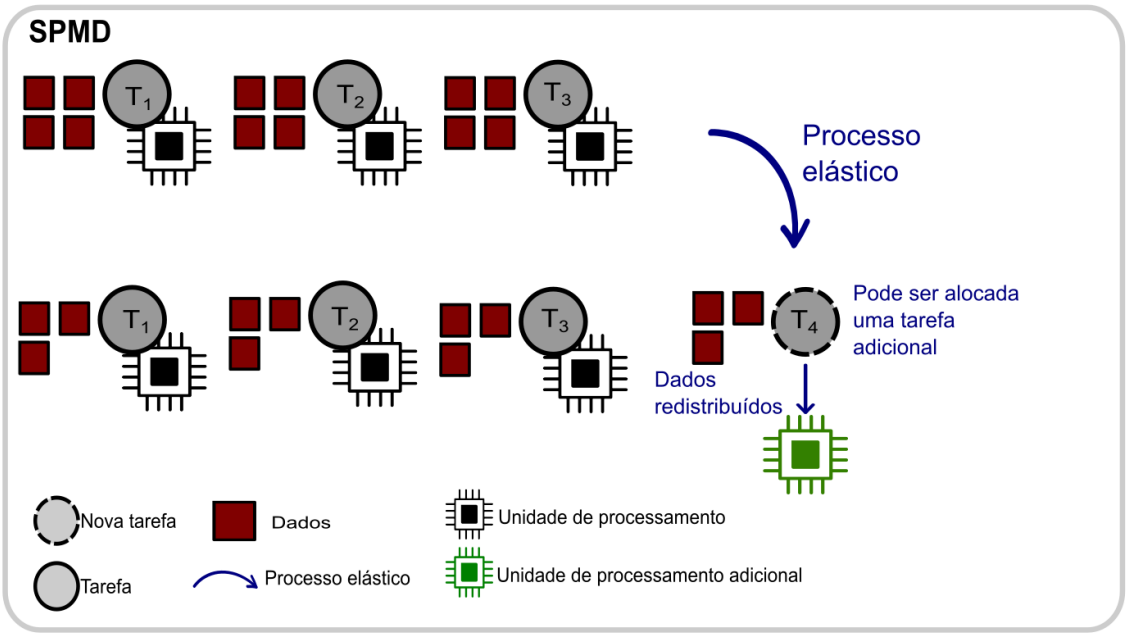
O padrão SPMD, que implementa elasticidade horizontal, será definido neste trabalho como ElasticHor SPMD, e o padrão que implementa a elasticidade vertical ficará definido como ElasticVert SPMD. Para definir o tipo de padrão que será aplicado para a utilização da elasticidade é necessário a definição do tipo de sistema onde a aplicação será implementada.

Se o sistema for de memória distribuída, é possível implementar a elasticidade horizontal, utilizando o padrão ElasticHor SPMD, mas existe a limitação quanto ao número de CPU's disponíveis. Nesse caso, é necessário avaliar se o número de tarefas trabalhadores gerada no início da execução é menor do que o número de CPU's disponíveis no sistema. Ainda assim, durante a execução, caso a elasticidade horizontal seja implementada sozinha, pode haver um gargalo quando finalizar o número de CPU's disponíveis e o sistema necessite escalar novamente.

Ao aplicar a elasticidade utilizando o padrão ElasticHor SPMD, é possível ajustar dinamicamente o número de nós de processamento ou a quantidade de recursos alocados a cada nó com base nas necessidades da aplicação. Isso ajuda a lidar com os gargalos e problemas do SPMD de várias maneiras (ERL; COPE; NASERPOUR, 2017). Um exemplo de como a elasticidade horizontal pode ser aplicada para alocação de mais recursos está na Figura 17. Têm-se ali as tarefas T1, T2 e T3 e, caso seja necessário mais recursos para finalizar o processamento, pode-se criar uma nova tarefa e disponibilizar uma nova unidade de processamento para essa tarefa criada, T4.

A elasticidade horizontal permite que um sistema SPMD seja escalado para lidar com grandes conjuntos de dados ou tarefas computacionais intensivas. A medida que a carga de trabalho aumenta, mais nós de processamento podem ser adicionados para distribuir a carga e melhorar o desempenho global. A Figura 18 mostra o início da

Figura 17 – Elasticidade horizontal SPMD.




computação com as tarefas T1, T2 e T3. Ao encerrar a computação de T3, o sistema libera a unidade de processamento que estava alocada para T3. A elasticidade horizontal permite adicionar mais nós de processamento para lidar com grandes volumes de dados e realizar operações distribuídas, como mapeamento e redução, de maneira mais eficiente (QIN; MA; NIU, 2019).

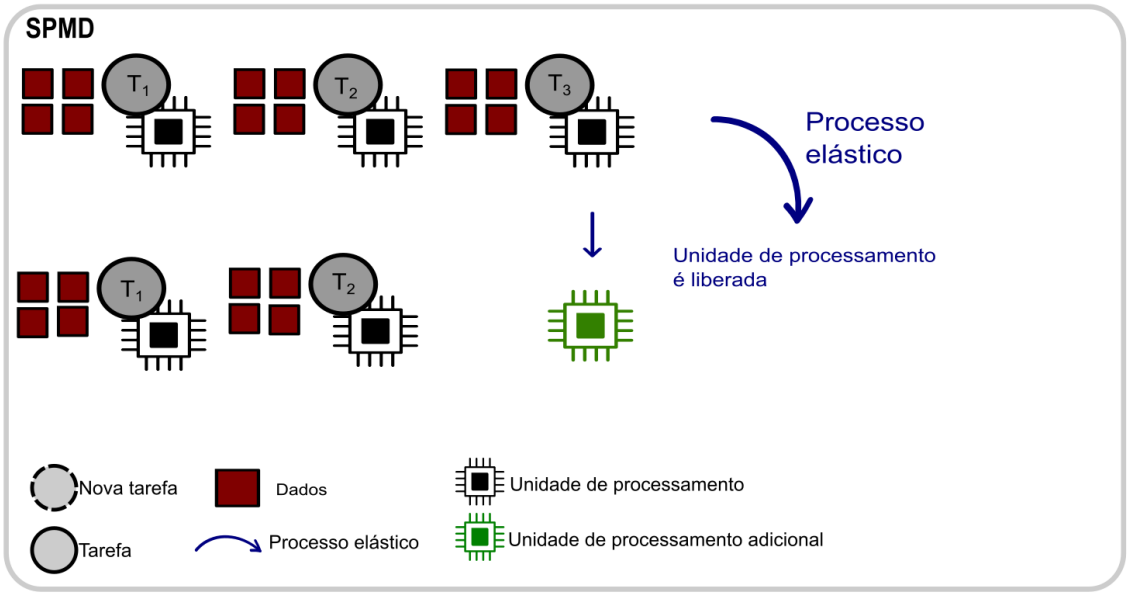
Para a implementação da elasticidade horizontal no padrão SPMD pode ser utilizado um monitoramento realizado por um *middleware*, ou uma rotina na aplicação que avalia a utilização de recursos em cada nó e, de acordo com limites pré definidos, aloca ou libera nós de computação. Esse monitoramento pode avaliar o tempo que cada nó leva para executar suas tarefas, permitindo que, caso alguma tarefa esteja excedendo o tempo de execução das demais, alguma ação de elasticidade seja executada. Para implementar essa modalidade de elasticidade, tem-se o trabalho apresentado por (FATÉMA et al., 2017).

Esses mesmos nós podem ser desalocados quando encerram o seu processamento, assim como mostra a Figura 18 com as tarefas T1, T2 e T3. Ao terminar o processamento de T3, pode ser liberado a unidade de processamento que estava alocada para esta tarefa.

Exemplos de implementação da elasticidade horizontal no padrão SPMD incluem (QIN; MA; NIU, 2019):

- Computação em Grade: Em projetos de computação em grade, onde várias instituições colaboram para resolver problemas computacionais intensivos, a elasticidade horizontal pode ser usada para adicionar mais recursos computacionais conforme a

Figura 18 – Elasticidade horizontal SPMD.




demanda cresce, permitindo a conclusão mais rápida dos cálculos.

- Aplicações Web Escaláveis: aplicações web que seguem o modelo SPMD podem se beneficiar da elasticidade horizontal para lidar com picos de tráfego. Ao adicionar mais instâncias de servidores web ou de aplicativos, a aplicação pode acomodar um grande número de usuários simultâneos de forma eficaz.

A adição de mais recursos de computação pode levar a tempos de resposta mais rápidos e a um maior *throughput*. Porém, em algumas situações, as tarefas precisam ser sincronizadas para garantir que compartilhem dados corretamente. No entanto, uma sincronização excessiva pode levar a atrasos, especialmente se as tarefas precisarem esperar umas pelas outras com frequência. Nesse caso, a elasticidade vertical pode ser útil ao aumentar os recursos de um nó especifico, permitindo que esse nó finalize a execução de sua tarefa mais rapidamente, evitando que outros nós fiquem tempo demais aguardando para sincronização (PERICÀS, 2018). Se as tarefas precisam trocar muitos dados entre si, pode ocorrer um gargalo de comunicação, especialmente se a largura de banda de comunicação entre as tarefas for limitada. Tal problema pode ser minimizado através da utilização da elasticidade, aumentando a largura de banda de comunicação entre as tarefas (PERICÀS, 2018).

Além disso, o padrão SPMD pode enfrentar dificuldades na escalabilidade. À medida que o número de nós de processamento aumenta, a complexidade da coordenação e a comunicação também aumenta. Isso pode resultar em limitações de desempenho e na necessidade de soluções adicionais, como otimização da rede ou estratégias de balanceamento de carga para manter a eficiência do sistema (FURQUIM, 2006).

Se a aplicação for executada em um sistema com memória compartilhada, o tipo de elasticidade ideal seria a elasticidade vertical, já que em sistemas de memória distribuída várias CPU's utilizam um mesmo endereçamento de memória física. Os nós podem ser escalados com recursos para melhorar o desempenho da aplicação.

A elasticidade vertical pode ser implementada utilizando o ElasticVert SPMD, o qual permite um desempenho melhorado do padrão SPMD ao adicionar mais recursos à máquina existente. Se um subconjunto de tarefas tiver mais trabalho do que outras, isso pode levar a um desbalanceamento de carga, onde algumas tarefas terminam seu trabalho muito antes das outras, resultando em ociosidade e subutilização de recursos. Além disso, elasticidade vertical pode proporcionar mais poder de processamento (CPU), memória e armazenamento, permitindo que as tarefas possam ser executadas mais rapidamente e com maior eficiência, especialmente quando as operações envolvem grandes conjuntos de dados. Isso pode ser implementado ao adicionar um monitoramento nos nós ou em um *middleware*. Quando a utilização dos recursos naqueles chegar a um determinado limite, o mecanismo de elasticidade pode ser acionado para alocar mais recursos, ocorrendo o mesmo para desalocar os recursos. No padrão SPMD, cada nó pode ter uma rotina que monitore a sua utilização de recursos para disparar mecanismos de elasticidade vertical. Um dos *frameworks* apresentado no Capítulo 3.2 que possibilita a implementação desse padrão é o trabalho apresentado por (MARTÍN et al., 2015) que aborda a implementação de elasticidade vertical alocando ou liberando novas CPU's para processos que requisitarem.

Em algumas situações, especialmente em tarefas que envolvem interações frequentes entre os dados, a elasticidade vertical pode melhorar a latência, pois as comunicações entre os processos são mais rápidas em uma única máquina do que através da rede entre várias máquinas. Como várias instâncias do programa estão operando simultaneamente em diferentes partes dos dados, é necessário sincronizar as operações para evitar conflitos e inconsistências. Essa sincronização pode levar a um alto *overhead* de comunicação e atrasos na execução, especialmente em sistemas com grande número de nós de processamento. Nesse caso, o aumento do processamento (CPU, memória, etc.) em um único nó, com a criação de novas tarefas e divisão de dados em um mesmo nó, melhora a latência de comunicação. Ela simplifica o gerenciamento em comparação com a adição de várias máquinas ao sistema, pois se lida com menos entidades. Esse processo pode facilitar o monitoramento, a manutenção e a administração do sistema, reduzindo a complexidade operacional.

Alocar mais recursos a uma única máquina pode levar a uma utilização mais eficiente dos recursos, especialmente em cargas de trabalho que não podem ser facilmente divididas entre várias máquinas. Isso evita a fragmentação de recursos que pode ocorrer quando várias máquinas são adicionadas.

Exemplos de aplicação da elasticidade vertical em sistemas SPMD incluem:

- Bancos de Dados Grandes e Complexos: bancos de dados que exigem operações complexas e consultas intensivas podem ser melhorados pela elasticidade vertical. Alocar mais memória e CPU a uma única instância de banco de dados pode melhorar significativamente o desempenho, especialmente para consultas que envolvem várias tabelas e grandes volumes de dados.
- Processamento de Imagens e Vídeos de Alta Resolução: aplicações que lidam com o processamento de imagens ou vídeos de alta resolução podem se beneficiar da elasticidade vertical. Ao adicionar mais recursos a uma única máquina, é possível realizar operações intensivas de processamento de imagem e vídeo de forma mais rápida e eficiente.
- Simulações Científicas Complexas: em simulações que exigem grande poder de processamento e grandes quantidades de memória, como simulações de dinâmica molecular em química computacional, adicionar mais CPUs e memória a uma única tarefa pode acelerar significativamente o tempo necessário para realizar cálculos complexos.

Já para a implementação da elasticidade combinada, um dos *Frameworks* que permitem essa implementação estão (MO-HELLENBRAND, 2019). O padrão definido neste trabalho para a implementação da elasticidade combinada é o padrão ElasticVH SPMD. Esse padrão pode ser utilizado para implementação em sistemas de nuvem computacional, mas também em sistemas de memória distribuída. Nesse último é necessário levar em consideração a quantidade de recurso que seria escalado horizontalmente

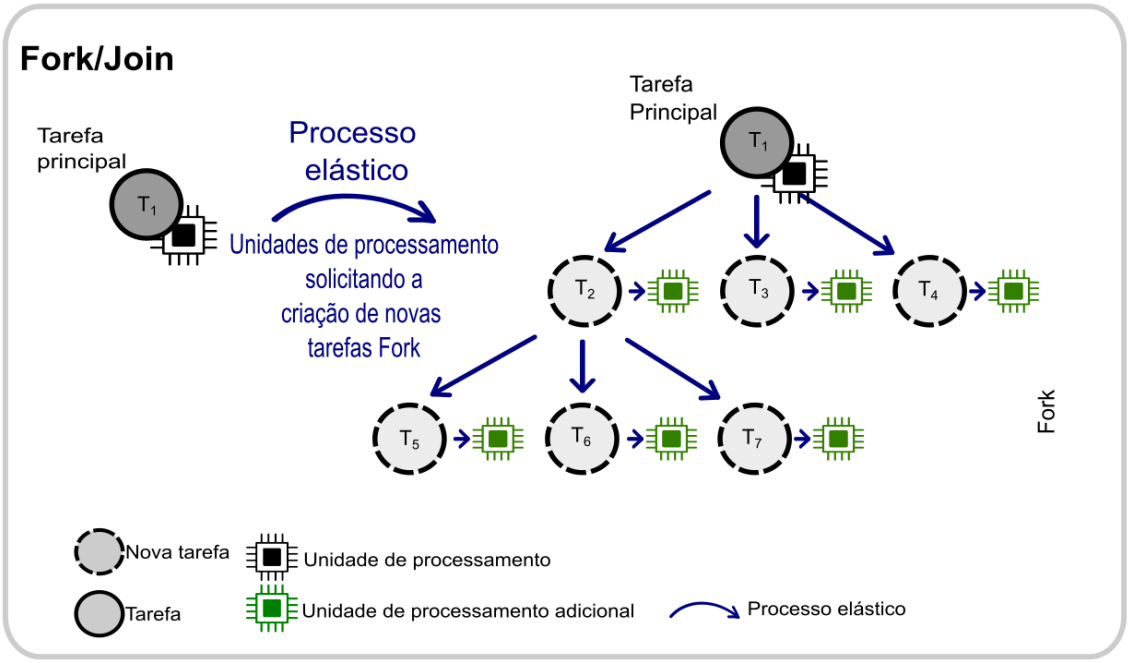
### 4.2.3 Padrão *Fork/Join*

A estrutura de um programa*fork/Join* possui um comportamento dinâmico natural, uma vez que as tarefas são bifurcadas e unidas em tempo de execução. Por ter uma estrutura de algoritmo recursiva, as tarefas são criadas em tempo de execução. Geralmente esse padrão é implementado utilizando *pools* de *threads*.

Um aplicativo *fork/join* consiste em uma tarefa de origem que se bifurca em muitas tarefas de ramificação que, por sua vez, se unem novamente em uma tarefa coletora. A tarefa de origem prepara e divide as tarefas de ramificação. Já a tarefa coletora reúne e combina resultados para as etapas subsequentes, conforme mostra a Figura 19. Ali, têm-se a tarefa de origem T1, a qual gera outras tarefas T2, T3, T4, T5, T6 e T7, sendo que cada uma dessas tarefas recebe unidades de processamento. Com este tipo de aplicação, inicia-se alocando recursos suficientes para a tarefa de origem e, ocorrendo a bifurcação,

novas unidades de processamento são alocadas para as tarefas recém-criadas. No final, quando as tarefas são combinadas, os recursos utilizados pelas tarefas intermediárias podem ser liberados.

Figura 19 – Elasticidade horizontal *fork/join*.



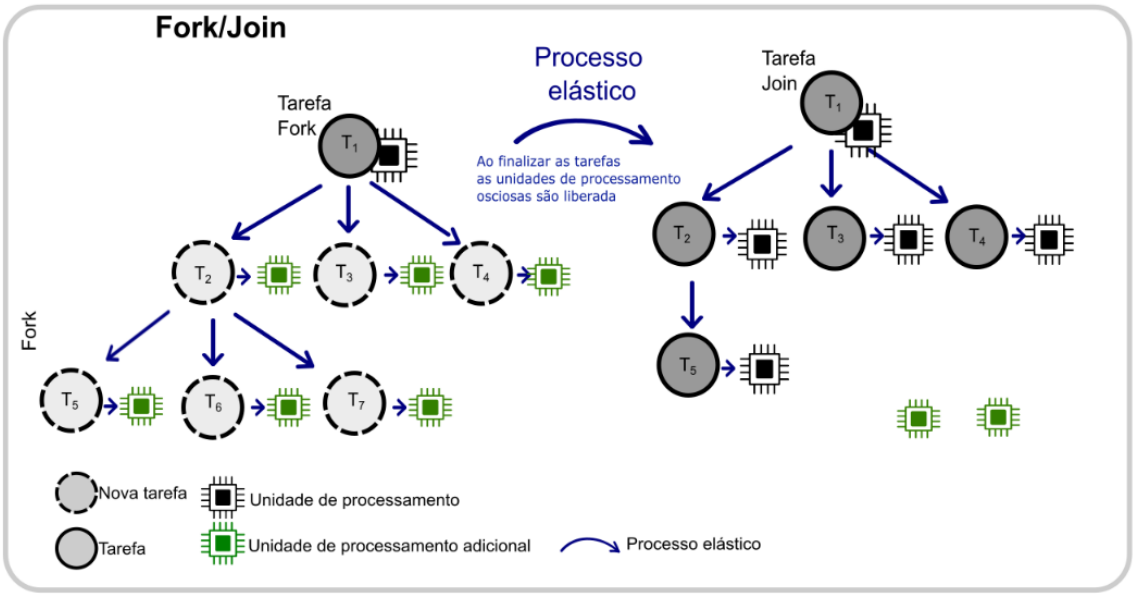

Idealmente, cada tarefa deveria ser atribuída a uma unidade de processamento diferente no contexto. Contudo, como o número de subproblemas pode crescer exponencialmente, isso nem sempre é possível, já que a maioria das plataformas computacionais possui um número fixo de processadores. Uma solução é parar a recursão quando o número de subproblemas ativos for igual ao número de processadores. No entanto, tal abordagem limita o grau de paralelismo. Uma alternativa é a alocação dinâmica de recursos, começando pelos recursos para a raiz da árvore e, a cada divisão da carga de trabalho, criar recursos adicionais para mapear as tarefas. À medida que os resultados se fundem, os recursos que estiverem ociosos podem ser desalocados e alocados para outras tarefas. Isso é feito utilizando a elasticidade horizontal, permitindo aumentar a capacidade de processamento e reduzindo o tempo necessário para concluir a tarefa principal. No contexto deste trabalho a denominação desse padrão será ElasticHor *fork/join*.

Se o sistema for de memória distribuída é possível implementar a elasticidade horizontal, utilizando o padrão ElasticHor fork/join. Porém, existe a limitação do número de CPU's disponíveis e, nesse caso, é necessário avaliar se o número de tarefas trabalhadores gerada no início da execução é menor do que o número de CPU's disponíveis no sistema. Ainda assim, durante a execução, caso a elasticidade horizontal seja implementada sozinha, pode haver um gargalo quando finalizar o número de CPU's disponíveis e sistema necessite escalar novamente. Para implementar essa modalidade de elasticidade tem-se o trabalho

apresentado por (FATÉMA et al., 2017).

Se uma máquina falha durante a execução de uma sub-tarefa, outras máquinas podem ser alocadas para continuar a processar as demais sub-tarefas, garantindo a conclusão bem-sucedida da tarefa principal, trazendo ao padrão tolerância a falhas e confiabilidade. Isso pode ser implementado ao criar um monitoramento para avaliar se os nós estão executando, sendo esse monitoramento executado de tempos em tempos, e esse limite de tempo sendo determinado pela resiliência da aplicação. Caso esse monitoramento constate que algum nó está inoperante, outro é alocado e recebe a tarefa para execução. Um exemplo disso pode ser observado em um sistema de análise de dados em tempo real onde, se uma máquina falhar ao processar uma parte dos dados, outras máquinas ainda podem processar o restante dos dados, assegurando que a análise seja concluída (ERL; COPE; NASERPOUR, 2017). Quando a execução finaliza, essas máquinas e/ou processadores podem ser desalocados liberando os recursos para que não haja subutilização, conforme mostrado na Figura 20, onde se tem as tarefas T1, T2, T3, T4, T5, T6 e T7. Quando as tarefas T6 e T7 terminam sua execução, as unidades de processamentos alocadas são liberadas.

Figura 20 – Elasticidade horizontal *fork/join*.




Se a aplicação for executada em um sistema com memória compartilhada, o tipo de elasticidade ideal seria a elasticidade vertical e, para isso, propõem-se o padrão ElasticVert fork/join. Para implementar esse padrão neste tipo de sistema os recursos podem ser escalados para melhorar o desempenho da aplicação.

O padrão ElasticVert *fork/join* aplica a elasticidade vertical ao modelo *fork/join*, adicionando recursos como CPU e memória a uma máquina existente para melhorar o desempenho geral do padrão. Com mais recursos, as sub-tarefas podem ser executadas mais

rapidamente evitando espera excessiva da tarefa join. Aumentar a capacidade de memória da máquina ou de um nó pode permitir que o sistema mantenha uma grande quantidade de dados na memória, reduzindo a necessidade de acessar dados no armazenamento, o que pode ser mais lento. Em um sistema de banco de dados que utiliza o padrão *fork/join* para consultas complexas, uma maior capacidade de memória pode permitir que mais dados sejam armazenados em cache, acelerando as consultas subsequentes (RIZK; POLOCZEK; CIUCU, 2015). Monitorar a memória dos nós em execução permite que ações de elasticidade possam ser acionadas. Esse monitoramento pode ser feito pela tarefa principal, ou até mesmo por um *middleware*.

No padrão ElasticVert *fork/join* é necessário se atentar à granularidade das tarefas. Se as tarefas forem muito pequenas, o custo de criar e gerenciar os nós ou as *threads* pode superar o benefício obtido pela paralelização. Por outro lado, se as tarefas forem muito grandes, pode haver um desequilíbrio de carga entre os nós ou as *threads*, onde algumas ficam ociosas enquanto outras estão sobrecarregadas. Encontrar o tamanho ideal das tarefas é um desafio e pode exigir ajustes e otimizações cuidadosas.

Para tarefas que são intensivas em utilização de CPU, a adição de mais núcleos de CPU ou o aumento da velocidade do clock pode ser benéfico. A elasticidade vertical no padrão *fork/join* também pode ser implementada configurando dinamicamente a pool de threads com base nos recursos disponíveis na máquina. Em Java, que oferece suporte ao padrão *fork/join* por meio do *framework ForkJoinPool*, a configuração do número de *threads* na pool pode ser ajustada dinamicamente com base na capacidade da máquina.

Outro cuidado que se deve ter utilizando *fork/join* está relacionado à sincronização e à comunicação entre as *threads*. À medida que as *threads* dividem e executam as tarefas, pode ser necessário sincronizar os resultados intermediários ou compartilhar recursos comuns. A sincronização excessiva pode levar a um alto custo de comunicação e a condições de corrida, onde as *threads* competem por recursos, resultando em degradação de desempenho. Além disso, a comunicação entre as *threads* também pode introduzir atrasos, especialmente se a troca de dados for intensiva.

A escalabilidade é outro aspecto desafiador do padrão *fork/join*. Embora o paralelismo possa melhorar o desempenho em sistemas com um número limitado de núcleos de processamento, em ambientes com muitos núcleos o padrão pode enfrentar limitações. A coordenação entre um número muito grande de nós ou de *threads* e o gerenciamento de recursos compartilhados pode se tornar ineficiente, levando a gargalos de desempenho. Além disso, problemas de escalabilidade podem surgir quando a carga de trabalho não é distribuída de maneira uniforme entre os nós ou as *threads*, resultando em subutilização de recursos.

Alguns exemplos de utilização do padrão *fork/join* são:

- Pensando em um sistema de análise de *big data* que lida com a mineração de dados em grande escala, pode-se dividir a tarefa em sub-tarefas menores usando o padrão *fork/join*. Ao escalar horizontalmente, essas sub-tarefas podem ser distribuídas entre diferentes máquinas, acelerando o processamento e melhorando a eficiência geral (RIZK; POLOCZEK; CIUCU, 2015).

- Sistemas que otimizam rotas, por exemplo, de estoques e distribuição em grandes cadeias de suprimentos podem se beneficiar da elasticidade horizontal para lidar com grandes volumes de dados em tempo real. O padrão *fork/join* pode ser usado para otimizar rotas de entrega com base nas condições do tráfego em tempo real. A elasticidade horizontal permite que o sistema processe dados de tráfego em tempo real de várias fontes, garantindo que as entregas sejam feitas eficientemente, alocando novos nós a medida que mais tarefas são criadas.

Para a implementação do padrão ElasticVert *fork/join* o trabalho apresentado por (GALANTE; BONA, 2014) permite a implementação de elasticidade vertical alocando ou liberando novas *threads* para tarefas que requisitarem.

Se o sistema for o disponibilizado por nuvem, a utilização da elasticidade combinada seria possível através do padrão proposto ElasticVH *fork/join* para implementação da elasticidade combinando elasticidade vertical e elasticidade horizontal. Isso é possível ao definir um número de tarefas trabalhadores próximo ao número total de CPU's e escalar horizontalmente enquanto houver CPU's disponíveis, e quando o sistema atingir o *threshold* máximo passar a utilizar a elasticidade vertical para escalar os recursos, para implementação da elasticidade combinada, um dos *Frameworks* que permitem essa implementação estão Wrzesinska et al. (2005) e (NGUYEN et al., 2020).

### 4.2.4 Padrão Paralelismo de laço

Uma implementação eficiente para o padrão de paralelismo de laço depende da independência entre as iterações do laço. Cada iteração deve ser capaz de ser executada de forma independente das outras. Essa característica é crucial para garantir que as diferentes *threads* ou processos possam executar paralelamente sem conflitos de dados.

Se as iterações possuem um custo de comunicação elevado, o ideal seria a implementação em sistemas de memória compartilhada. A elasticidade vertical seria a ideal para a implementação desta modalidade de elasticidade, uma vez que isso seria possível através do padrão Laço ElasticVert. Nesse padrão os recursos seriam escalados ou liberados de forma vertical sem que houvesse a necessidade de redividir as tarefas para alocar um novo nó.

A elasticidade da nuvem pode ser explorada neste cenário para executar um conjunto de iterações. Assim, com uma carga de trabalho dinâmica, a tarefa principal, responsável por coordenar as iterações, pode alocar ou desalocar recursos para executar a aplicação com um determinado número de tarefas que melhor atendam à demanda atual. Embora simples, tal estratégia só é fácil de implementar alterando o código da aplicação. Ou seja, um novo recurso só é útil se tivermos pelo menos uma tarefa em execução nele e a tarefa deve ser incorporada à aplicação. Uma solução aqui é explorar a estratégia em tempo de compilação para transformar uma aplicação não elástica em uma aplicação elástica (ERL; COPE; NASERPOUR, 2017). Sem alterar nenhuma linha de código da aplicação, é possível compilá-la com um *middleware* que insere linhas de código no processo de coordenação. Alguns controles de elasticidade poderiam ser inseridos para analisar o equilíbrio do sistema ao iniciar um novo laço. Se for detectada uma saturação do sistema (por exemplo, se a carga média de recursos da CPU for superior a 90%), um novo contador poderá ser inserido e uma nova tarefa poderá ser mapeada para ele utilizando o padrão Laço ElasticVert. Portanto, a cada laço, observar-se o comportamento do sistema, ligando ou desligando unidades de processamento para acomodar a execução adequada de um determinado laço. Nesse contexto, com a aplicação da elasticidade, é possível dimensionar a quantidade de recursos disponíveis de acordo com a carga de trabalho (FAKHFAKH; KACEM; KACEM, 2017).

Um exemplo de como esse tipo de elasticidade pode ser aplicado na distribuição recursos está na Figura 21. Ali tem-se a tarefa T1 e as iterações de laço. Quando há o processo elástico essas iterações são distribuídas entre as tarefas criadas, quais sejam, T2, T3 e T4 e essas tarefas recebem unidades de processamento solicitadas através de processo elástico.

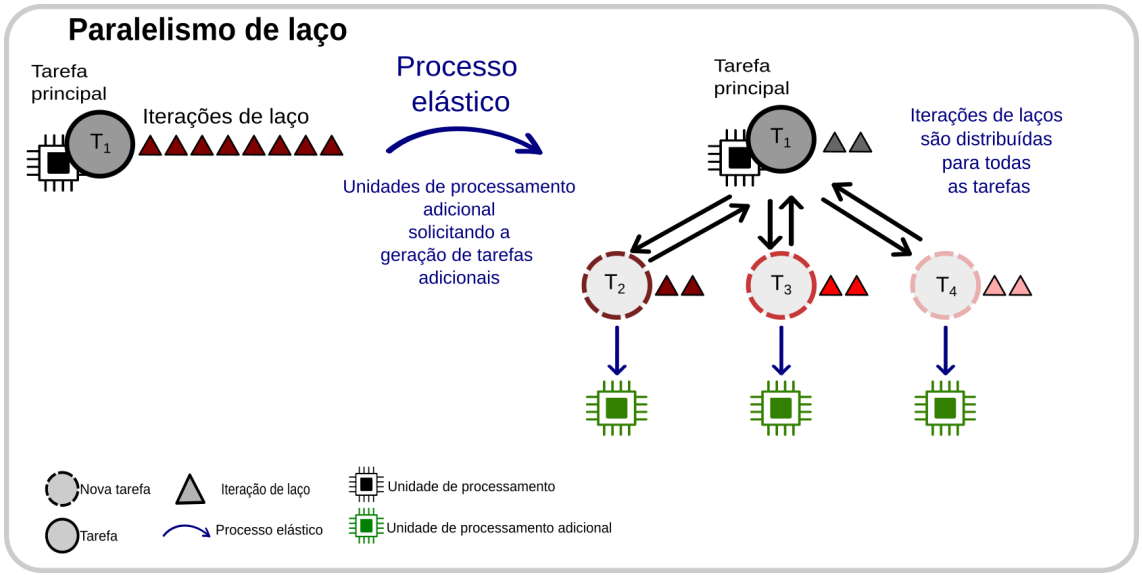
A elasticidade permite que recursos ociosos sejam aproveitados de forma eficiente. Quando algumas *threads* ou processadores terminam suas tarefas antes das outras, esses recursos podem ser reatribuídos para ajudar a acelerar a conclusão do laço. Isso ajuda a reduzir a subutilização de recursos e a melhorar o desempenho geral.

A elasticidade também pode ajudar a lidar com falhas em recursos. Se uma *thread* ou processador falhar durante a execução do laço, a elasticidade pode substituí-lo por um novo recurso disponível. Dessa forma, a execução do laço pode continuar sem interrupções, garantindo a conclusão da computação.

Ela também ajuda no gerenciamento eficiente da memória compartilhada. Com a distribuição dinâmica de iterações entre *threads* ou processadores, é possível reduzir os conflitos de acesso simultâneo à memória compartilhada, minimizando assim os problemas de consistência e as condições de corrida. Se o sistema utilizado for de memória distribuída, seria possível a utilização da elasticidade horizontal que, no contexto deste trabalho, será

definido como Laço ElasticHor.

Figura 21 – Elasticidade horizontal paralelismo de laço.



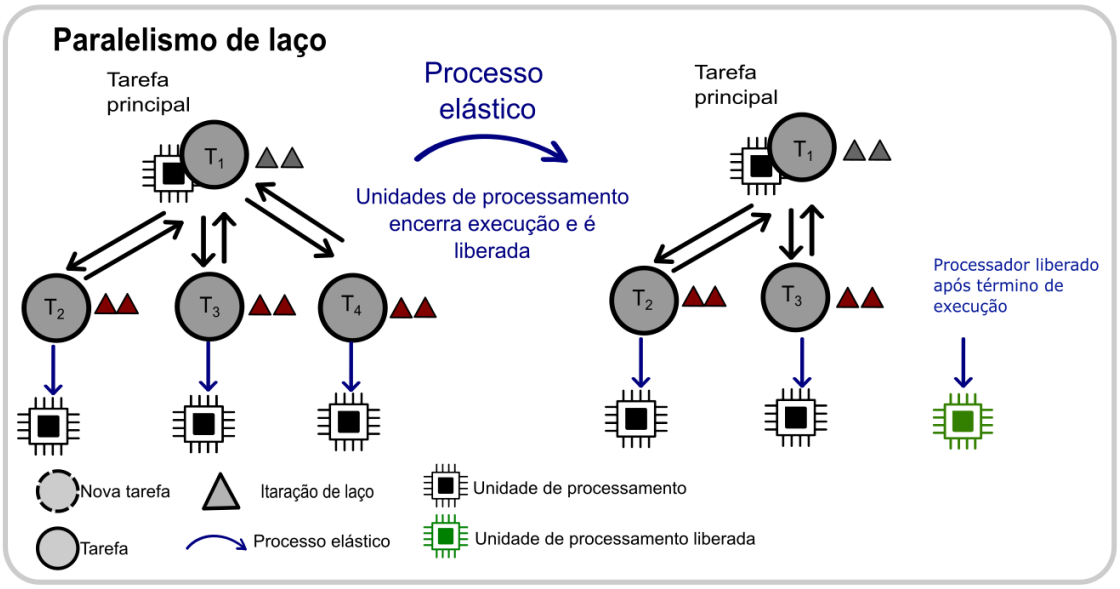

Se o sistema utilizado for de memória distribuída, seria possível a utilização da elasticidade horizontal que no contexto deste trabalho será definido como Laço ElasticHor. Para a implementação há que se levar em consideração como a comunicação será feita entre os nós de computação, e considerar se existe a possibilidade de redividir as iterações entre esses nós. Se o sistema utilizado for nuvem computacional, na maioria dos casos a utilização da elasticidade vertical será o ideal. Alguns trabalhos apresentados na seção 3.2 permitem que aplicações baseadas em paralelismo de laço sejam reescritas para utilizar elasticidade, dentre eles está (DANELUTTO; TORQUATI, 2014) e (BU et al., 2012).

Como exemplo de implementação da elasticidade horizontal utilizando o padrão laço ElasticHor, pode-se dividir os laços em pequenos pedaços de trabalho e distribuí-los em diferentes nós do sistema conforme a demanda. Cada nó pode então processar seu próprio pedaço de trabalho em paralelo com outros nós. Se a carga de trabalho aumentar, podem ser adicionados mais nós ao sistema de forma dinâmica, aproveitando a elasticidade horizontal para lidar com um maior número de iterações do laço (FAKHFAKH; KACEM; KACEM, 2017).

A elasticidade permite também a redução de recursos quando as iterações vão sendo finalizadas e não há mais para serem distribuídas, assim evitando a subutilização de recursos, conforme mostra a Figura 22. Na referida figura pode-se ver as T1, T2, T3 e T4. Elas possuem iterações que estão sendo executadas por unidades de processamento e, ao terminar a execução das suas iterações, a tarefa T4 libera a unidade de processamento que estava utilizando.

Outra questão para ser avaliada é o balanceamento de carga. Nem todas as iterações de um laço têm a mesma carga computacional, e a distribuição desigual de trabalho entre as

Figura 22 – Elasticidade horizontal Paralelismo de laço.




*threads* ou processadores pode resultar em baixa utilização de recursos. Isso ocorre quando algumas*threads* ou processadores terminam suas tarefas rapidamente, enquanto outras continuam ocupadas com iterações mais pesadas. Como resultado, o desempenho geral pode ser prejudicado. Nesse caso a elasticidade vertical pode trazer melhorias, aumentando os recursos de nós com uma carga computacional maior, ou que estão demorando mais tempo para executar as iterações dos laços (BU et al., 2012).

Além disso, a paralelização de laços requer uma granularidade adequada. Se o laço for muito pequeno, os custos de sincronização e comunicação entre as *threads*, processadores ou nós de processamento podem superar os benefícios obtidos. A elasticidade vertical pode resolver essa questão aumentando a capacidade dos nós de processamento, a fim de aumentar a granularidade do laço, diminuindo a latência relacionada com a sincronização e comunicação. Por outro lado, se o laço for muito grande, pode haver dificuldades em particionar o trabalho eficientemente e garantir um bom equilíbrio de carga. Nesses casos, a utilização da elasticidade horizontal pode ser útil, criando novos nós para a execução de laços menores (BU et al., 2012).

Outro desafio é a gestão eficiente da memória. Quando várias *threads* ou processadores estão envolvidos, pode ocorrer acesso simultâneo a regiões de memória compartilhada, resultando em conflitos e condições de corrida. É necessário garantir a correta sincronização e coordenação do acesso à memória para evitar problemas de consistência e erros inesperados (ERL; COPE; NASERPOUR, 2017).

Embora o paralelismo de laço possa trazer ganhos de desempenho em sistemas com um número moderado de *threads* ou processadores, pode não escalar bem para sistemas com um grande número de recursos. A sobrecarga adicional de coordenação e sincronização

entre um número crescente de *threads*ou processadores pode limitar os benefícios obtidos.

Sistemas modernos muitas vezes usam serviços de orquestração como Kubernetes para gerenciar a escalabilidade horizontal. Ao utilizar esses serviços pode-se configurar políticas para adicionar ou remover nós automaticamente com base na carga de trabalho, garantindo que os laços sejam paralelizados eficientemente, independentemente das flutuações na demanda (AL-DHURAIBI et al., 2017).

Para a implementação da elasticidade no padrão de paralelismo de laço é importante considerar a comunicação respondendo algumas perguntas como:

- Qual o custo de comunicação entre as iterações?
- As iterações estão divididas de forma que o custo computacional desta divisão não seja maior do que o beneficio trazido pela execução em paralelo?

### 4.2.5 Políticas e mecanismos de elasticidade aplicada aos padrões estrutura de apoio

Em relação à política de elasticidade, a manual é melhor aplicada quando requer um maior controle sobre os recursos para os administradores da infraestrutura, mas serve para casos bem específicos onde tal controle é necessário. Decisões contextuais complexas, em que a decisão de escalar ou reduzir recursos é altamente dependente do contexto do negócio, como decisões estratégicas baseadas em análises de mercado ou eventos específicos.

Já a política automática é benéfica quando eficiência operacional e uma resposta rápida do sistema se fazem necessárias. O sistema pode monitorar automaticamente a utilização dos recursos. Se essa taxa ultrapassar um limite predefinido, a elasticidade automática pode adicionar mais instâncias de servidor para lidar com o aumento da demanda. Quando a demanda diminui, as instâncias extras podem ser automaticamente desativadas, economizando recursos. Quanto aos mecanismos de elasticidade, os proativos utilizam algoritmos de previsão para antecipar mudanças na carga de trabalho. Eles proporcionam uma eficiência na utilização de recursos ao antecipar variações na demanda. Assim, a elasticidade proativa ajuda a evitar a subutilização ou superutilização de recursos. Enquanto isso, os mecanismos reativos respondem às mudanças na carga de trabalho em tempo real, com base em sinais imediatos, como aumento repentino na demanda ou queda abrupta.

A Tabela 2 apresenta os padrões elásticos propostos.

| **Padrão Paralelo** | **Padrão Elástico** |
|---|---|
| Mestre/Trabalhador | Mestre: Mestre ElasticVert, Mestre ElasticHor, Mestre ElasticVH<br>Trabalhador: Trabalhador ElasticVert, Trabalhador ElasticHor, Trabalhador ElasticVH |
| SPMD | SPMD ElasticVert, SPMD ElasticHor e SPMD ElasticVH |
| Paralelismo de laço | Laço ElasticVert, Laço ElasticHor e Laço ElasticVH |
| Fork/join | ElasticVert *fork/join*, ElasticHor *fork/join* e ElasticVH *fork/join* |

Tabela 2 – Padrões propostos

# 5 Aplicação do modelo proposto

Este capítulo visa exemplificar a aplicação do modelo proposto, e também a implementação da elasticidade para o modelo utilizando os *Frameworks* apresentados. Para isso serão utilizadas aplicações apresentadas por Foster (1995).

## 5.1 Método das diferenças finitas

O método das diferenças finitas é um método de resolução de equações diferenciais que se baseia na aproximação de derivadas por diferenças finitas. A fórmula de aproximação e obtida através da série de Taylor da função derivada. Hoje, os MDFs são a abordagem dominante das soluções numéricas de equações diferenciais parciais ao considerar um problema unidimensional para exemplificar a aplicação da análise desta fase. Para a implementação desse método, tem-se um vetor com $N$ elementos, e precisamos aplicar a Equação 5.1 em todos os elementos deste vetor.

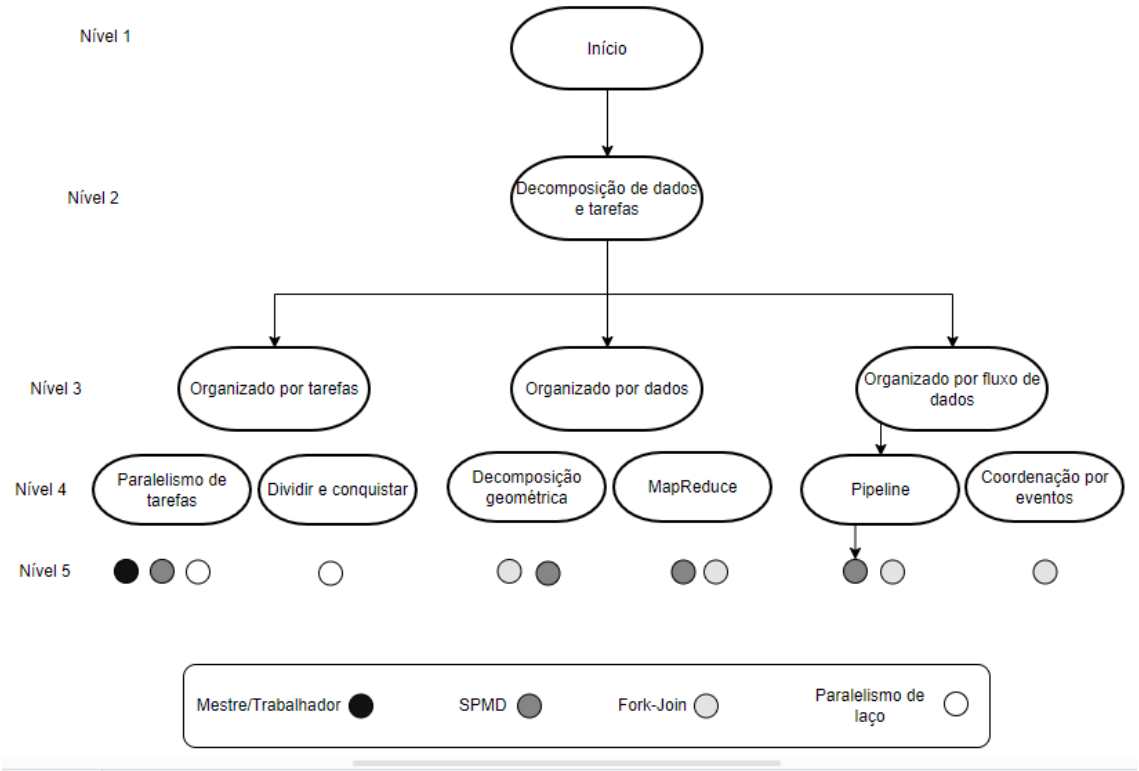
$$0 < i < N-1, 0 \leq t < T : X_i^{(t+1)} = \frac{X_{i-1}^{(t)} + 2X_i^{(t)} + X_{i+1}^{(t)}}{4} \tag{5.1}$$

Para a implementação dessa aplicação pode-se pensar na criação de um vetor que armazena os valores de $N$ nas posições i que vão de 0 até $N$ , onde cada número armazenado na posição i do vetor executará a Equação 4.1 e retornará o valor em um novo vetor. Importante observar que as $N$ tarefas podem ser executadas de forma independente, mas há restrição na ordem de execução, pois requer a sincronização imposta pelas operações de recebimento. Essa sincronização garante que nenhum valor de dados seja atualizado na etapa $t + 1$ até que os valores dos dados nas tarefas vizinhas tenham sido atualizados na etapa $t$.Para a implementação dessa aplicação pode-se pensar na criação de um vetor que armazena os valores de N nas posições $i$ que vão de 0 até $N$ , onde cada número armazenado na posição $i$ do vetor executará a Equação 4.1 e retornará o valor em um novo vetor. Importante observar que as $N$ tarefas podem ser executadas de forma independente, mas há restrição na ordem de execução, pois requer a sincronização imposta pelas operações de recebimento. Essa sincronização garante que nenhum valor de dados seja atualizado na etapa $t + 1$ até que os valores dos dados nas tarefas vizinhas tenham sido atualizados na etapa $t$.

Ao aplicar o modelo proposto no Capítulo 4, pode-se utilizar os passos mostrados na Figura 23, iniciando no passo identificado como 1, que tem como entrada essa aplicação para ser desenvolvida em qualquer linguagem de programação. No passo identificado pelo nível 2 na árvore, realiza-se a decomposição inicial do algoritmo, identificando se ele possui

uma estrutura majoritariamente definida por decomposição de tarefas, decomposição de dados ou de fluxo de dados. Como a execução da função MDF é realizada em cada elemento de forma independente, pode-se aplicar aqui a organização por tarefas. No passo identificado pelo número 3, quando aplica-se a decomposição de tarefas, verifica-se o tipo de sistema onde esta aplicação será executada, se será em uma arquitetura de memória compartilhada ou distribuída. É necessário avaliar de forma aproximada a quantidade de processamento que esse sistema possuirá, se as tarefas geradas são suficientes, além do custo de comunicação do sistema. Esta fase é a de avaliação de dependências e agrupamento de tarefas. Neste caso específico, há dependência entre as tarefas relacionadas à restrições de ordem de execução, pois é necessário garantir que o valor de dados não seja atualizado na etapa $t + 1$ até que os valores dos dados nas tarefas vizinhas tenham sido atualizados na etapa $t$.

Figura 23 – Árvore de decisão do modelo proposto para aplicação MDF.




Ao analisar os dados e suas dependências, tem-se que os dados executados pelas tarefas são de leitura e escrita, ou seja, é preciso implementar uma proteção em tais dados. Por exemplo, cada tarefa pode manter um *buffer* com o dado recebido pela tarefa calculado na etapa t e, com o seu dado local, onde será aplicada a Equação **??**, será enviado para a etapa $t + 1$. Ou seja, considerando uma tarefa na posição $i$, tem-se que essa tarefa tem uma entrada de dados, a qual é resultado de saída de dados da tarefa $i - 1$, e também terá uma saída de dados que será a entrada da tarefa $i + 1$. Todas essas dependências devem ser tratadas pelo desenvolvedor.

Em relação à decomposição por fluxo de dados, esse algoritmo possui uma dependência organizada por fluxo de dados, pois o cálculo realizado para cada elemento $i$ depende do valor de entrada calculado para o elemento $i - 1$. Nesse caso, é necessário avaliar questões de comunicação entre as tarefas $i$. Embora esse algoritmo possa ser inicialmente organizado por tarefas, percebe-se uma dependência do fluxo destes dados, o que faz com que a implementação dele seja baseada na organização pelo fluxo de dados, conforme mostra a Figura 23.

Em relação à ordem de execução de tarefas, ou seja, o escalonamento, como tem-se um número de elementos $N$ definido no início da computação, gerando $N$ tarefas, o escalonamento nesse caso é estático.

Para a definição do padrão de estrutura de apoio, presente no nível 5 da árvore, pode-se avaliar as características das tarefas. Nesse caso, é possível definir a quantidade de tarefas no início da computação, o que permite realizar o mapeamento das mesmas para UE's para o caso da implementação utilizar *threads* ou PE's, caso a escolha seja de implementar utilizando processos. A partir dessas características tem-se o padrão que melhor se aplica, o padrão SPMD, conforme mostra o último nível da árvore de decisão.

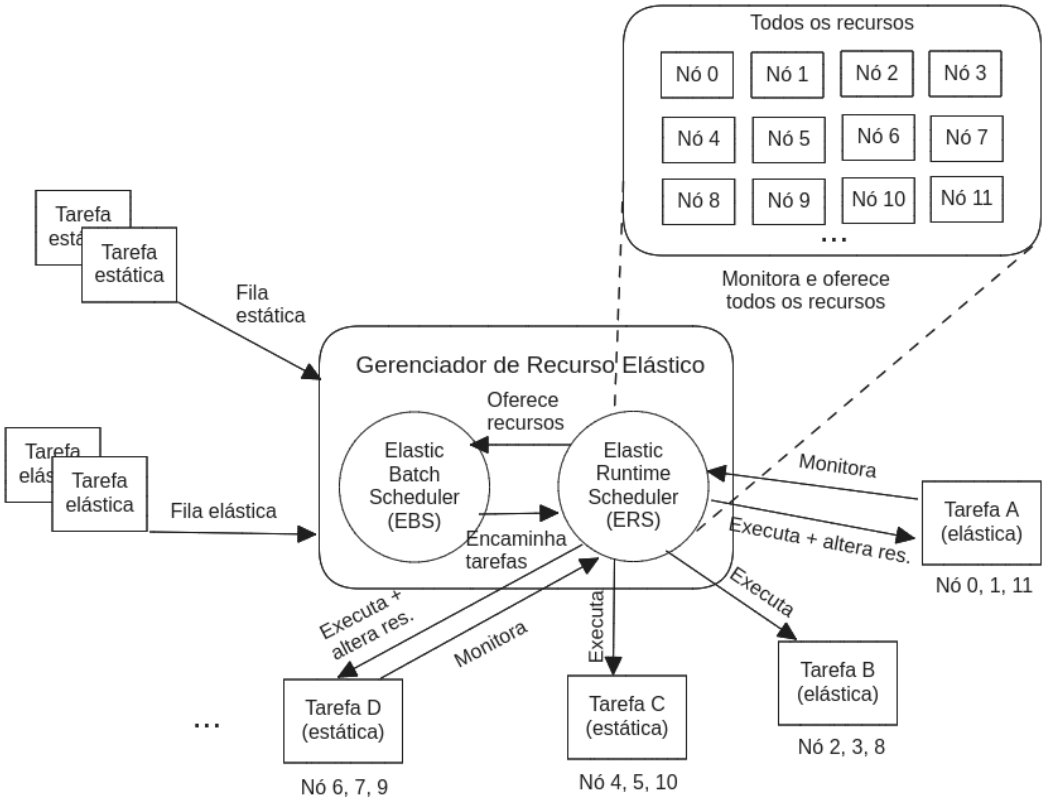
Como na execução desta aplicação há a dependência na ordem da execução das tarefas, e há a necessidade da comunicação entre as tarefas, é necessário que o desenvolvedor se atente ao custo da comunicação entre os nós, já que um dos gargalos do padrão SPMD é a comunicação entre os nós de execução.

Aplicando a elasticidade no padrão de estrutura de apoio SPMD, utilizamos o padrão ElasticVH SPMD, pois é possível usar a elasticidade horizontal e vertical de forma combinada para melhorar o desempenho do algoritmo, no caso de considerarmos a implementação dessa aplicação em um sistema de memória distribuída, conforme tratado no Capítulo 4.2.2. A elasticidade horizontal pode ser utilizada para a criação de novos nós ou também para a desalocação de nós ociosos, permitindo assim a escalabilidade e confiabilidade da aplicação. Nesse cenário, caso um nó pare de funcionar, será rapidamente substituído por outro, assim como se um nó se tornar ocioso, será desalocado. A elasticidade horizontal pode ser aplicada executando uma função que monitore a execução dos N nós alocados no início da aplicação. Caso algum nó $i$ pare de funcionar, pode ser alocado um novo nó que receberá os dados armazenados no nó $i$ e continuará a execução. Já a elasticidade vertical pode ser utilizada para aumentar os recursos dos nós. Caso necessário, a mesma função de monitoramento pode manter uma variável de tempo de execução, por exemplo, que avalie se há nós que estejam demorando mais para executar o cálculo, podendo assim acionar o mecanismo de elasticidade e alocar mais recursos para esse nó.

A implementação deste padrão pode ser feita utilizando o *framework* Elastic MPI apresentado por Mo-Hellenbrand (2019), pois este *framework* implementa a elasticidade

vertical e horizontal. O gerenciador do Elastic MPI é apresentado na Figura 24. O gerenciador consiste em dois agendadores, o *Elastic Batch Scheduler* (EBS) e o *Elastic Runtime Schedule* (ERS).

Figura 24 – Elastic MPI (MO-HELLENBRAND, 2019).




O módulo EBS gerencia duas filas de tarefas. Uma das filas com tarefas estáticas e a outra com tarefas elásticas. Ele decide quais tarefas serão executadas e as encaminha para o ERS, que aloca os recursos para essas novas tarefas. Quando há recursos disponíveis para a execução de novas tarefas, o EBS seleciona tarefas de ambas as filas com base em políticas definidas pelo administrador do sistema, como seus tamanhos (recursos solicitados), sua prioridade, imparcialidade (a quanto tempo a tarefa está esperando, por exemplo), etc. No caso da aplicação MDF, a restrição que deve ser definida para a execução das tarefas é para que a tarefa $t + 1$ só pode ser executada ao término da tarefa $t$.

O ERS supervisiona e gerencia todos os recursos, desde a inicialização, execução e finalização das tarefas. Quando os recursos ficam disponíveis, o agendador ERS toma decisões sobre alocá-los para novas tarefas ou atribuí-los a outras tarefas que já estejam em execução. O ERS também toma decisões em tempo de execução sobre a realocação de recursos de acordo com a necessidade das tarefas em execução. Essas decisões do ERS são tomadas a partir da análise de ações de sondagem pelas tarefas utilizando uma nova operação implementada pelo *Framework*. Espera-se que as tarefas elásticas se comuniquem

(sondagem) com o gerenciador de recursos de maneira frequente e tomem ações imediatas de acordo com a necessidade apresentada. Essas ações de comunicação e adaptação de recursos são facilitadas pela biblioteca Elastic MPI. Nessa operação, demonstrada na Figura Figura 25, o parâmetro de saída *dapt_flag* informa ao aplicativo se ele deve ou não se adaptar. O valor *MPI_ADAPT_FALSE* é fornecido quando não há nenhuma ação de elasticidade, ou *MPI_ADAPT_TRUE* quando há uma ação de elasticidade.

Figura 25 – Exemplo código 1

```
1 int MPI_Probe_adapt(
2 int *adapt\_flag, int *proc_status, MPI_Info *info);
```

Figura 26 – Exemplo código 2

```
1 int MPI_Init_adapt(
2 int *argc, char **argv, int *proc_status);
```

O ERS comunica suas decisões ao EBS e à todas as tarefas em execução. Para trabalhos estáticos, o ERS apenas os inicia e permite que sejam executados até a conclusão sem interação. Para que seja possível diferenciar tarefas elásticas de tarefas estáticas, o*Framework*implementa uma nova operação mostrada na Figura 26. Essa operação permite que o EBS saiba diferenciar as tarefas.

No caso da aplicação MTD, o código inicia com esse método indicando que as tarefas serão elásticas. O Elastic MPI permite que a aplicação MDF implementada possa ser confiável, pois caso um nó que está executando uma tarefa deixe de funcionar, quando o agendador ERS realizar a sondagem das tarefas em execução, irá identificar que o nó não está funcionando e poderá alocar outro recurso livre para a execução daquela tarefa. Assim como, caso um nós termine a execução da sua tarefa, o agendador ERS será notificado, podendo liberar os recursos alocados para a execução dessa tarefa, proporcionando economia nos custos e podendo aumentar a velocidade de execução da aplicação, já que o recurso não estará ocioso, podendo ser alocado a uma tarefa que está presente na fila gerenciada pelo EBS aguardando execução. Além disso, o *Framework* permite que o programador especifique no início da execução da aplicação a restrição no fluxo de execução das tarefas, o qual será controlado pelo agendador EBS, garantindo que a tarefa $t + 1$ não seja executada antes da tarefa t ter finalizado sua execução.

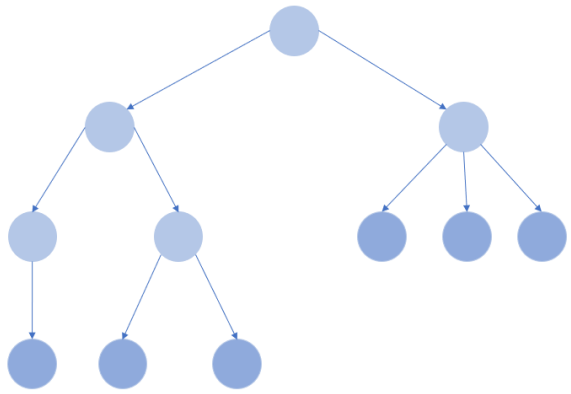
Através da utilização do *Framework Elastic MPI* é possível tornar o padrão SPMD elástico para a aplicação MDF, o que o torna confiável, resiliente, econômico e escalável

para diferentes tamanhos de N .

## 5.2 Árvore de Busca Simples

O algoritmo de busca simples consiste em uma chamada de função que gera uma árvore de busca como mostra a Figura 27. No caso dessa aplicação, uma função de busca é aplicada recursivamente em um conjunto de dados, formando assim uma árvore de execução. Essa função de busca é executada em um conjunto de dados que deve ser definido e separado no início da execução do algoritmo.

Figura 27 – Árvore de busca.

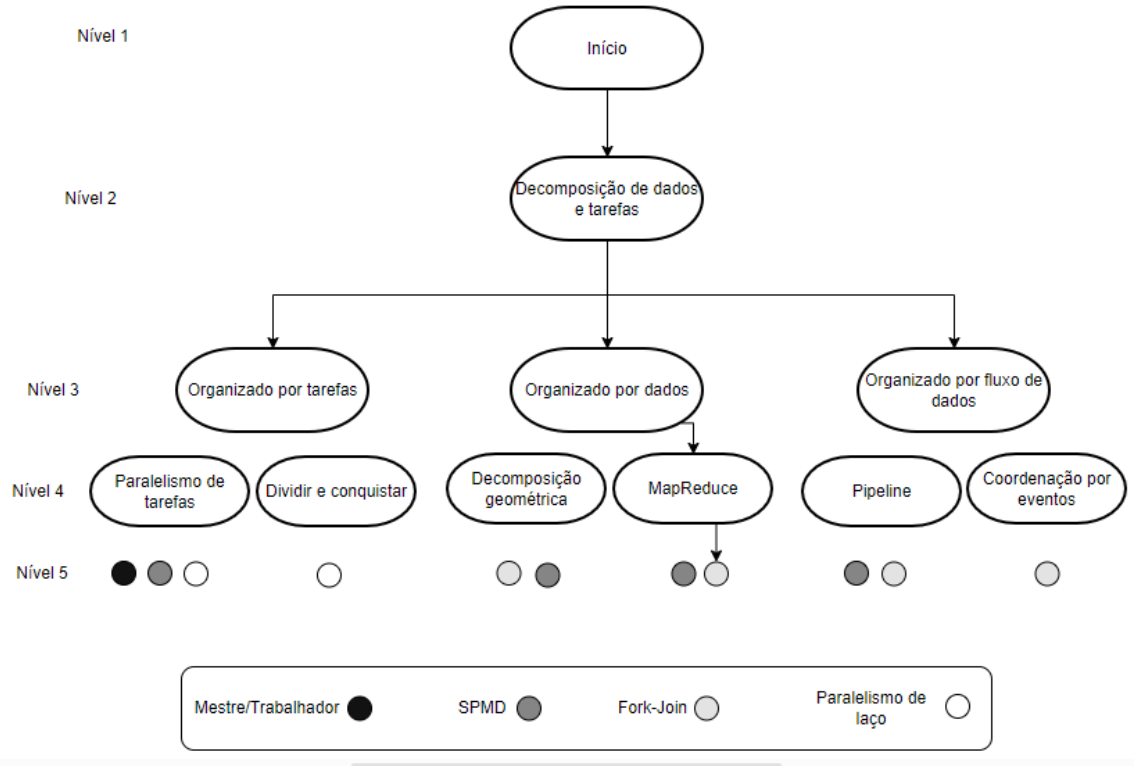


Pode-se acompanhar a aplicação do modelo proposto na Figura 28, iniciando no passo 1, tendo como entrada essa aplicação para ser desenvolvida em qualquer linguagem de programação. No passo identificado pelo nível 2 da árvore, realiza-se análise de concorrência, onde é feita a decomposição inicial do algoritmo. Pode-se identificar que esse algoritmo possui uma estrutura majoritariamente organizada por decomposição de dados.

Na fase de decomposição de dados e de análise de dependência verifica-se que os dados são somente de leitura. Não há modificação, e é possível manter cópias locais dos dados no caso de sistemas distribuídos. Vale neste caso uma avaliação do custo para a separação inicial de tais dados. Verifica-se também que esses dados serão decompostos entre tarefas que fazem, por exemplo, a comparação desse conjunto de dados com uma *string*. Depois de aplicada a decomposição de dados, verifica-se o tipo de sistema onde o tipo de sistema onde a aplicação será executada, se será de memória compartilhada ou distribuída. É importante avaliar, mesmo que de forma aproximada, a quantidade de processamento que esse sistema possuirá, qual o custo de comunicação do sistema e se as tarefas geradas são suficientes.

A próxima fase, presente no nível 3 da árvore, é a de compor os dados que serão distribuídos para cada tarefa. Nessa fase, é possível perceber que existe uma dependência entre as tarefas, já que cada tarefa pode ser entendida como a folha de uma árvore, como

Figura 28 – Árvore de decisão do modelo proposto para aplicação de busca simples.




mostra a Figura 27. Cada tarefa pai só pode ser encerrada ao término da execução das tarefas filhos. Nesse caso, a dependência está na ordem da execução.

Em relação à ordem da execução das tarefas na aplicação de busca simples, o escalonamento é dinâmico, pois não é possível definir o número de tarefas no início da computação. Isso pode requerer algum tipo de tratamento para balanceamento de carga. Após a avaliação realizada nos níveis 2 e 3 da árvore, é possível identificar que o padrão de estrutura de algoritmo que pode ser aplicado é o mapreduce, presente no nível 4 da árvore de decisão. Esse seria o melhor padrão, pois as tarefas são geradas de forma dinâmica e o algoritmo de busca segue criando tarefas (nós) enquanto a solução não é encontrada.

No nível 5 da árvore tem-se o padrão de estrutura de apoio mais indicado que, segundo o modelo proposto, é o *fork/join*. Esse padrão é o ideal pelo fato das tarefas serem geradas recursivamente, estarem conectadas de forma irregular e também pela carga de trabalho em cada PE ou UE variar de maneira imprevisível, pois novas tarefas (nós) podem ser geradas a qualquer momento.

Para a implementação da elasticidade, pode-se considerar a implementação da aplicação em um ambiente em nuvem, que permite que a elasticidade vertical e horizontal possam ser implementadas de forma combinada, conforme apresentado no Capítulo 4.2.3, para isso o padrão utilizado será o ElasticVH *fork/join*.

A aplicação do padrão *fork/join* pode ser implementado da seguinte forma: a tarefa

inicial contém toda a massa de dados, os quais são divididos e distribuídos para os nós fork. Conforme a necessidade, a elasticidade horizontal pode ser utilizada quando, ao gerar uma nova divisão de dados originando mais tarefas fork, o sistema aloque mais nós de computação para realizar a busca naquela massa de dados. Ao final da busca naquela massa de dados, o resultado é reportado ao nó que deu origem a tarefa finalizada, liberando assim o recurso computacional utilizado por aquele nó em execução. A elasticidade vertical também pode ser utilizada aqui, permitindo aumentar a capacidade de computação de um nó em específico quando o mesmo está levando muito tempo para realizar as buscas na massa de dados que compõem a sua tarefa.

Dos *frameworks* apresentados, o que mais se adequa para implementar o algoritmo de busca simples, é o apresentado por Nguyen et al. (2020). Segundo os autores, esse *framework* possibilita a implementação do padrão *fork-join* para aplicações intensivas em dados, se caracterizando como um modelo que implementa controle de elasticidade proativo. O termo Black-box, utilizado pelos autores, indica que o modelo é projetado para ser aplicável a uma variedade de aplicações de processamento de dados, independentemente dos detalhes internos de implementação. É ideal para aplicações onde não se conhece inicialmente o número de tarefas a serem executadas e nem a quantidade de dados a serem processados por cada uma das tarefas criadas. Isso se torna ideal na implementação do algoritmo de busca simples, conforme se avaliou anteriormente, já que não se sabe inicialmente a quantidade de dados de entrada e nem o número de tarefas que serão executadas. O modelo Black-box se baseia na análise de características observáveis das tarefas, como tamanho dos dados, complexidade computacional e padrões de acesso, para fazer previsões de latência, permitindo um balanceamento de carga em tempo de execução.

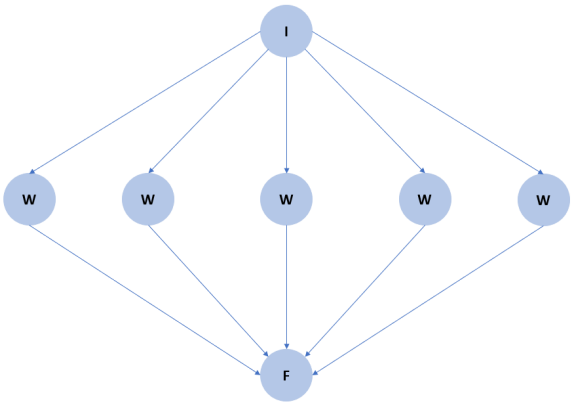
A implementação inicia com uma tarefa que contem toda a massa de dados inicial, a qual será dividida por um número de nós calculado pelo modelo. Esses nós terão os recursos redimensionados, tanto de CPU's como de memória, caso haja necessidade de acordo com os cálculos realizados dentro de cada Black Box. Esse processo pode ser feito recursivamente pelo *Framework*. Assim, para uma implementação resolvendo um problema de árvore de busca, todos os dados da árvore são carregados para a tarefa inicial, e o modelo fica responsável pelo cálculo para divisão dessa tarefa em outras e, também, pelo redimensionamento dos recursos conforme a necessidade para a busca em cada um dos nós. Se ao término do processamento dos dados em um nó, o dado buscado não for encontrado, o nó encerra a execução liberando os recursos utilizados. Nesse ponto, o *Framework* avalia a necessidade de redimensionar a carga ou recursos entre os nós que ainda estão sendo executados. Isso traz mais rapidez para a execução de uma busca em uma grande quantidade de dados. Isso é demonstrado pelos autores em um dos estudos de casos realizados para buscas de palavras-chave de um pool de 50.000 palavras, demonstrando que o modelo de predição proposto pode ser 2 vezes melhor do que um modelo de controle

de elasticidade reativo.

## 5.3 Estudo de parâmetros

Em alguns programas chamados embaraçosamente paralelos, as tarefas podem ser executadas de forma praticamente independente sem qualquer comunicação entre si, realizando essa comunicação apenas no final, conforme mostra a Figura 29. Os valores dos parâmetros são lidos por uma tarefa a partir de um arquivo de entrada, e os resultados dos diferentes cálculos são gravados em um arquivo de saída por outra tarefa.

Figura 29 – Estudo de parâmetros.



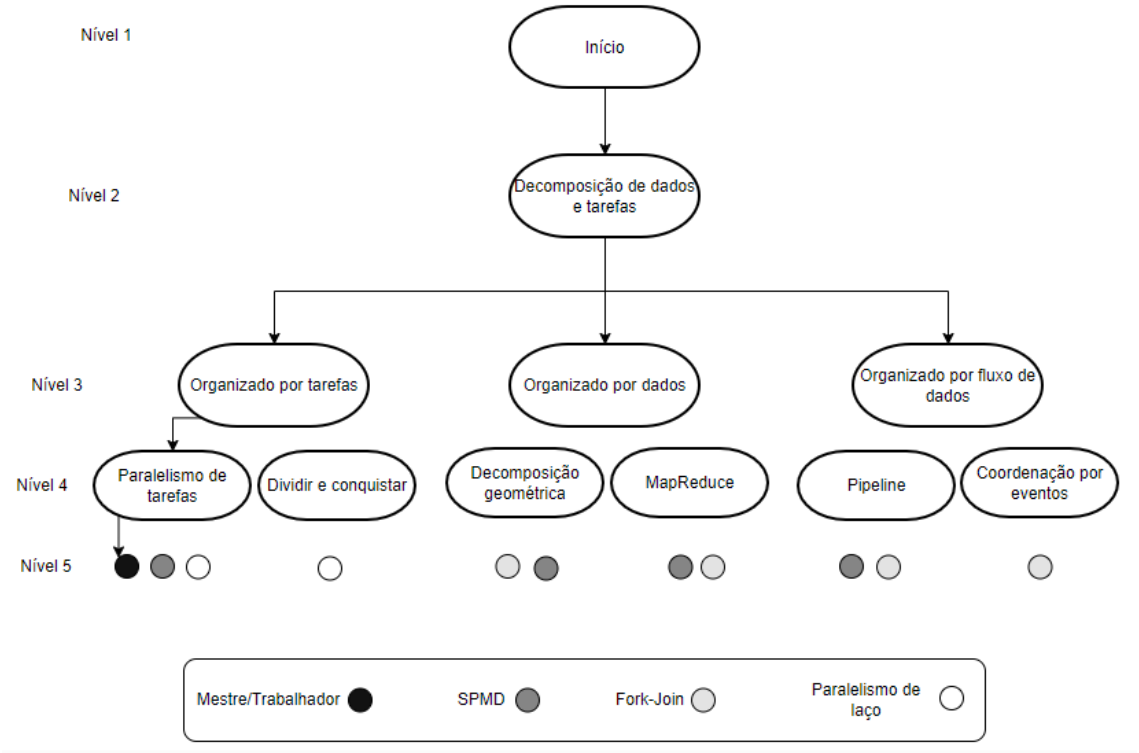

Para a implementação desta aplicação podemos pensar na criação de tarefas independentes que podem se chamar *"task"*. Aplicando o modelo proposto na Figura 30, inicia-se no passo identificado como 1, tendo como entrada essa aplicação para ser desenvolvida em qualquer linguagem de programação. Para tal algoritmo, no nível 2 na árvore, analisa-se se a organização da aplicação é majoritariamente disposta por tarefas, dados ou fluxo de dados. No caso da aplicação de estudo de parâmetros, identifica-se que o algoritmo possui uma estrutura majoritariamente definida por decomposição de tarefas.

Para a aplicação que é organizada por tarefas, a fase presente no nível 3 consiste em avaliar as dependências e o agrupamento das tarefas. Nesse caso especifico, não há dependências entre as tarefas, nem de tempo, nem de restrições de ordem. Ou seja, as tarefas são independentes e a execução de uma não interfere na outra.

Aplicando a decomposição das tarefas, tenta-se avaliar de forma aproximada a quantidade de processamento que o sistema possuirá, se o algoritmo gera tarefas suficientes e qual o custo de comunicação do sistema. Verifica-se o tipo de sistema onde a aplicação será executada, se será de memória compartilhada ou distribuída.

Em relação a ordem de execução de tarefas, ou seja, o escalonamento, como tem-se um número de tarefas que é definido no início da computação, o escalonamento será

Figura 30 – Árvore de decisão do modelo proposto para aplicação de estudo de parâmetros.




estático. Analisando os dados e suas dependências, tem-se que os dados utilizados pelas tarefas podem ser locais, ou seja, não é preciso que uma proteção seja implementada neles. Cada tarefa pode copiar e alterar localmente o seu dado em caso do sistema ser distribuído, diminuindo assim o custo com comunicação.

O padrão de estrutura de algoritmo que pode ser aplicado no nível 4, após as avaliações realizadas no nível anterior, é o paralelismo de tarefas. Pode-se pensar na estrutura do algoritmo criando um **pool** de tarefas. Cada *"task"* executaria o que fosse necessário ao receber o dados do estado "I", retornando o resultado da sua computação no estado "F".

Levando em consideração o nível 5 da árvore, o padrão de estrutura de apoio indicado seria o mestre/trabalhador, pois nesse caso apresentado, embora seja um algoritmo embaraçosamente paralelo, existe uma tarefa mestre que lê os arquivos de entrada e distribui para que outras tarefas os executem. O padrão mestre/trabalhador também é indicado, já que as tarefas são conhecidas no início da computação. Devido a isso, é possível realizar o mapeamento dessas tarefas para as UE's no caso da implementação ser realizada por *threads*, ou para as PE's no caso da implementação ser realizada por processos.

Ao considerar a implementação desta aplicação em nuvem computacional, conforme apresentado na seção 4.2.1, é possível utilizar a elasticidade horizontal e vertical de forma combinada para melhorar o desempenho do algoritmo. Para isso, utiliza-se o padrão

mestre ElasticVH e trabalhador ElasticVH. A elasticidade horizontal seria utilizada para a criação de novos nós trabalhadores, aumentando a escalabilidade e confiabilidade pois, no caso de falha de um nó trabalhador, outro pode ser alocado. O mesmo acontece caso o nó mestre falhe, pois ele pode ser substituído por outro ou até mesmo ser colocado mais de um nó mestre gerenciando a execução quando o número de nós trabalhadores se torna grande demais para apenas um nó mestre gerenciar. Já a elasticidade vertical é utilizada para aumentar os recursos do nó tanto mestre quanto trabalhadores, caso haja gargalo no processamento das tarefas. É possível aumentar os recursos do nó mestre verticalmente caso o mesmo esteja enfrentando um gargalo na execução. O mesmo pode ser feito com os nós trabalhadores.

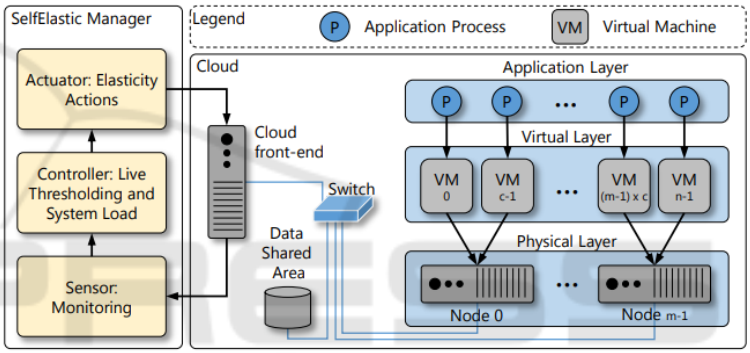
Para a implementação do padrão mestre/trabalhador, podemos utilizar o *Framework* SelfElastic, proposto por Rodrigues et al. (2018). A Figura 31 apresenta o *Framework* proposto que possui um gerenciador com três módulos, que implementam as politicas de elasticidade reativa e proativa.

Para a implementação do algoritmo de estudos de parâmetros, seriam instanciadas as MV's no início da computação pelo SelfElastic, cada MV's executaria uma *"task"*. O *Framework* então começaria a coleta de dados de execução. O módulo sensor monitora a carga de CPU de cada MV que está executando uma tarefa *"task"* de forma periódica, passando dados ao controlador para que seja possível a definição das métricas de elasticidades, já que ao invés de utilizar limites estáticos para executar ações de elasticidade, o SelfElastic propõe a adaptação dinâmica dos limites inferior e superior, os quais são iniciados com os valores 0 e 100, respectivamente. Esses valores são modificados ao executar as funções de adaptação: limiares() e redefinir limiares(). O primeiro é calculado a cada observação de monitoramento, enquanto o segundo é chamado apenas quando ocorre uma ação de elasticidade.

Como na definição dos autores, o *Framework* não precisa de ação do programador para utilização e o programador só precisaria executar o código da aplicação no SelfElastic. O monitoramento realizado das *"task"* permitiria a alocação de outra MV´s como mais recursos para a execução de uma *"task"*, ou a divisão de tarefas para aplicação de elasticidade horizontal na *"task"* com utilização de CPU acima do limiar definido pelo *Framework*.

Ao final da computação teríamos um arquivo de saída com os cálculos finais das *"tasks"*.

Figura 31 – SelfElastic (RODRIGUES et al., 2018).




## 5.4 Interação de pares

Esta aplicação possui restrições de comunicação semelhantes a aplicação apresentada na Seção 5.1, mas requer um algoritmo de comunicação mais complexo. Muitos problemas requerem o cálculo de todas as interações $N(N-1)$ entre pares $I(X_i, X_j), i \neq j$ entre $N$ dados $X_0, ..., X_N - 1$.

Um algoritmo paralelo simples para o problema geral de interações entre pares pode criar N tarefas. A tarefa i recebe o dado $X_i$ e é responsável por calcular as interações $I(X_i, X_j)|i \neq j$. Pode-se observar que, como cada tarefa precisa de um dado de todas as outras tarefas, serão necessários $N(N-1)$ canais para realizar as comunicações necessárias.

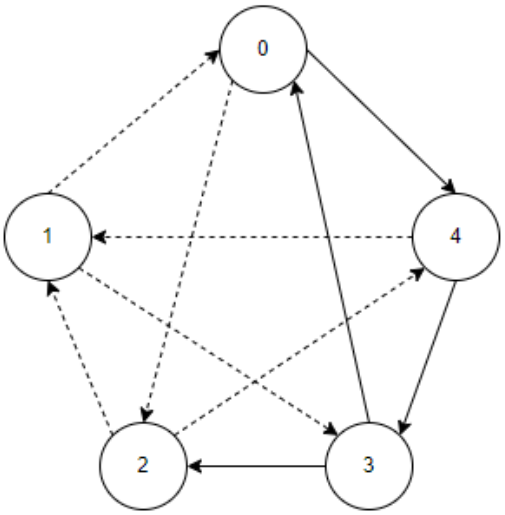
Porém, é possível uma estrutura mais econômica que utilize apenas $N$ canais. Esses canais são usados para conectar as $N$ tarefas em um anel unidirecional. Portanto, cada tarefa possui uma entrada e uma saída. Pode-se pensar em uma implementação em que cada tarefa inicializa primeiro um *buffer* (com o valor de seu dado local) e um acumulador que manterá o resultado do cálculo, assim a tarefa $X_i$ envia o dado calculado armazenado em seu *buffer* para o elemento $X_i + 1$ e recebe em seu *buffer* o valor do dado do elemento $X_i - 1$ e atualiza o seu acumulador local.

Porém, há como construir uma aplicação com interações simétricas, sendo assim possível reduzir pela metade o número de interações computadas e o número de comunicações, refinando a estrutura de comunicação. São criados N canais de comunicação adicionais, ligando cada tarefa ao deslocamento da tarefa $N/2$ ao redor do anel, como mostra a Figura 32.

Cada vez que uma interação $I(X_i, X_j)$ é calculada entre um dado local $X_i$ e um dado de entrada $X_j$, esse valor é acumulado não apenas no acumulador do dado $X_i$, mas também em outro acumulador que circula com $X_j$. Após as $[N/2]$ etapas, os acumuladores associados aos valores circulados retornam à sua tarefa inicial utilizando os novos canais, combinados com os acumuladores locais. Consequentemente, cada interação simétrica é calculada apenas uma vez: no nó $I(X_i, X_j)$ que contém ou no nó que mantém $X_i$ ou como

$I(X_i, X_j)$ no que contém $X_j$.

Figura 32 – Interação de pares.



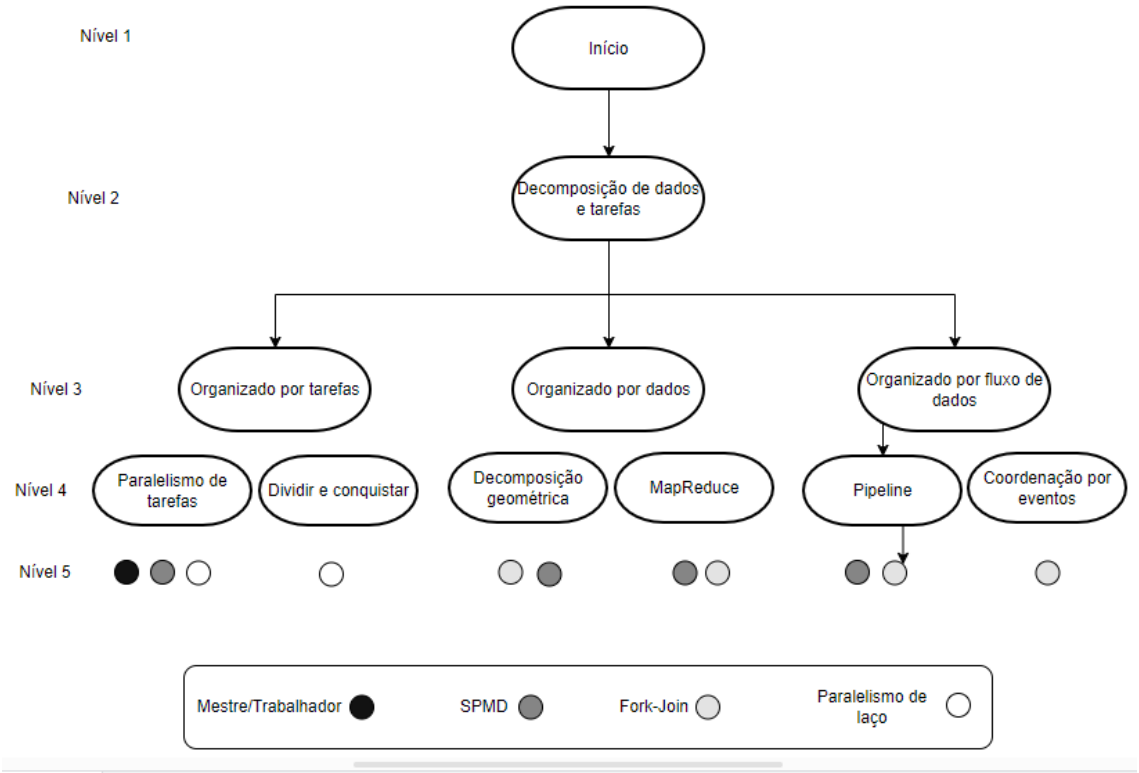

Aplicando modelo proposto no Capítulo 4, mostrado na Figura 33, para a implementação dessa aplicação inicia-se a avaliação pela análise de concorrência, nesta primeira fase mostrada no nível 2 da árvore. Aplicando a decomposição das tarefas que serão executadas, verifica-se o tipo de sistema onde a aplicação será executada, se será de memória compartilhada ou distribuída. Avalia-se de forma aproximada a quantidade de processamento que esse sistema possuirá, o custo de comunicação do sistema e se as tarefas geradas são suficientes para manter as UE's ou PE's ocupadas.

Em relação à ordem de execução de tarefas, ou seja, o escalonamento, como tem-se um número de elementos $N$ definido no início da computação, gerando $N$ tarefas, o escalonamento neste caso é estático. Ao analisar os dados e suas dependências, tem-se que os dados executados pelas tarefas são de leitura e escrita, ou seja, é preciso implementar uma proteção nesses dados. Como cada tarefa mantém um *buffer*, tem-se dados locais, e esse dado é lido e alterado pela tarefa, além da tarefa alterar o segundo *buffer* com o valor que foi calculado.

Em relação à decomposição por fluxo de dados, esse algoritmo possui uma dependência organizada por fluxo de dados, pois o cálculo realizado tem depência dos demais cálculos realizados pelas outras tarefas. Nesse caso, é necessário avaliar questões de comunicação entre os elementos. Embora esse algoritmo possa ser inicialmente organizado por tarefas, percebe-se uma dependência do fluxo dos dados, o que faz com que a implementação dele seja baseada na organização pelo fluxo de dados, conforme mostra a Figura 33.

O padrão de estrutura de algoritmo que pode ser utilizado na aplicação 4 é o pipeline, uma vez que há uma dependência estática no fluxo de execução e já que na

Figura 33 – Árvore de decisão do modelo proposto para aplicação interação de pares.




execução desse algoritmo a direção do fluxo de dados é conhecida. Já o padrão de estrutura de apoio, ou seja, o padrão que implementará para o *hardware*, é o *fork-join*, pois apesar de o algoritmo ter um número de tarefas definido no início da execução, existe uma dependência na ordem da execução dessas tarefas.

Conforme apresentado na seção 4.2.3, faz-se necessário definir o tipo de sistema que a aplicação será implementada para a definição da elasticidade a ser aplicada. Nesse caso, pode-se definir como exemplo que a implementação será realizada em um sistema de memória compartilhada e a melhor modalidade de elasticidade a ser aplicada seria a elasticidade vertical através do padrão ElasticVert *fork/join*.

Os *frameworks* apresentados na 3.2 que permitem a implementação do padrão *fork-join*, está o apresentado por Galante e Bona (2014). Os autores propõem um *Framework* que utiliza o OpenMP para implementar o padrão *fork-join*, e torna a elasticidade vertical aplicável a este padrão através da inserção do controle de elasticidade nas diretivas OpenMP, para possibilitar o ajuste automático do número de processadores virtuais (VCPUs), de acordo com a quantidade de *threads* em execução.

O *Framework* também inclui um conjunto de rotinas na biblioteca de nível de usuário, o que permite ao usuário configurar a elasticidade fornecida pelas diretivas. Usando esses recursos, é possível, por exemplo, expandir o conjunto de VCPUs para lidar com um número maior de *threads*, ou liberar VCPUs quando trechos seriais de código já

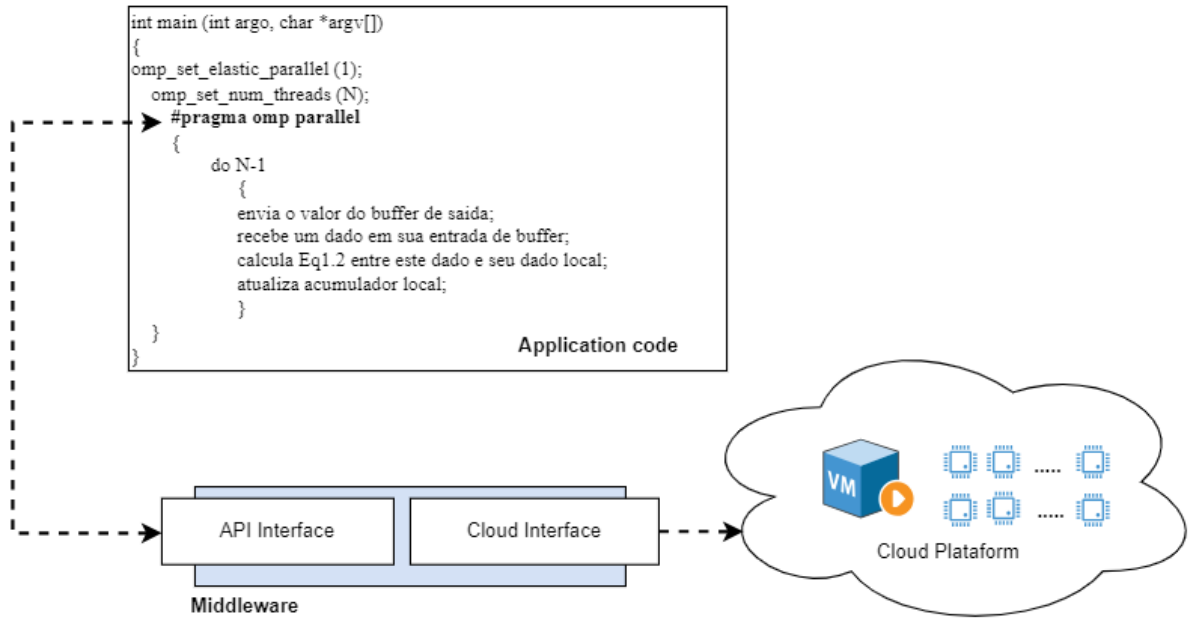
finalizaram a execução. Ou seja, ao pensar na implementação do algoritmo de interação de pares, o início da execução tem $N$ *threads* alocadas com $N$ VCPUs disponíveis e, a medida em que as *threads* finalizam a execução do seu cálculo, liberam as VCPUs que estavam utilizando.

Já a elasticidade vertical é proporcionada pelo *Framework* ao adicionar suporte para alocação dinâmica de memória à biblioteca OpenMP. Para habilitar os recursos de elasticidade do OpenMP, os autores propõem uma arquitetura composta por uma API OpenMP e um *middleware*. A elasticidade horizontal seria aplicada para esse algoritmo caso houvesse a necessidade de aumentar os recursos de memória para finalizar a execução do cálculo. Quando há essa necessidade, a API interage com o *middleware*, o qual acessa a infraestrutura da nuvem e realiza as alocações e desalocações de recursos.

Um exemplo de código para esta aplicação utilizando a diretiva apresentada no *Framework* é mostrado na figura Figura 34, onde $N$ é passado como parâmetro de números de *threads*. Nesse caso, é necessário a alteração do código pelo programador.

A Figura 34 apresenta como o código é implementado utilizando o *Framework*, a diretiva *#pragma omp parallel* inicia a execução paralela se comunicando com o *Middleware* através de um API, e esta API aciona uma interface que se comunica com a nuvem, alocando as MV's e CPU's passadas como parâmetro para a diretiva *omp_set_num_threads(N)*, que no caso desta aplicação é conhecido no início da computação $N$, o *Framework* gerencia a necessidade de alocar mais *threads*/memória ou liberar esses recursos.

Figura 34 – Adaptado de Galante e Bona (2014).



# 6 Conclusão

À medida que adentramos o cenário contemporâneo das aplicações de Computação de Alto Desempenho (HPC), torna-se evidente que estamos testemunhando uma era de transformação e crescimento. A demanda por processamento de dados cada vez mais complexos impulsiona a necessidade de soluções HPC mais avançadas e eficientes. No entanto, esse panorama traz grandes desafios, particularmente no desenvolvimento de aplicações HPC paralelas.

No centro desses desafios encontra-se a necessidade de maximizar a eficiência dos recursos computacionais, explorando o potencial pleno de arquiteturas paralelas e distribuídas. A natureza heterogênea dos sistemas modernos implica que os desenvolvedores enfrentam a complexidade de otimizar códigos para uma grande diversidade de arquiteturas, desde GPUs especializadas até *clusters* de computadores distribuídos. Essa diversidade exige uma abordagem cuidadosa e adaptativa no desenvolvimento de algoritmos e estruturas de dados de forma a garantir o desempenho ideal em cada contexto.

Além disso, a escalabilidade das aplicações HPC tornou-se um grande desafio. O design de algoritmos que podem tirar proveito de sistemas distribuídos e paralelos, utilizando elasticidade sem perder eficiência, é um desafio constante que exige inovação contínua. A complexidade inerente às aplicações HPC paralelas é agravada pela necessidade de lidar com a crescente heterogeneidade dos ambientes computacionais. O desenvolvimento de ferramentas e métodos que simplifiquem a programação para arquiteturas diversas, sem comprometer o desempenho, é uma prioridade para os pesquisadores e desenvolvedores nesse campo.

Até onde se sabe, este trabalho propôs a primeira contribuição de um modelo unificado de padrões de programação paralela elásticos, projetado para guiar os programadores desde as fases iniciais do desenvolvimento até a otimização final, oferecendo uma estrutura abrangente para enfrentar os desafios intrínsecos à computação paralela. O modelo propõe questionamentos e direcionamentos que permitem definir como a aplicação está organizada, seja por tarefas, dados ou fluxo de dados, e a partir daí proporciona um direcionamento para a escolha de padrões que definirão a estrutura do algoritmo, ou seja, o padrão que define a forma como essa aplicação está organizada majoritariamente. Além disso, orienta em como escolher os padrões de estrutura de apoio, sendo esses os padrões que efetivamente realizarão a comunicação da aplicação com o hardware. É nesses quatro principais padrões - mestre/trabalhador, fork-join, SPDM e paralelismo de laço - onde adicionou-se a elasticidade propondo os padrões elásticos mestre ElasticVert, mestre ElasticHor, mestre ElasticVH, trabalhador ElasticVert, trabalhador ElasticHor,

trabalhador ElasticVH, *fork/join* propoem-se: *fork/join* ElasticVert, *fork/join* ElasticHor, *fork/join* ElasticVH, SPMD ElasticVert, SPMD ElasticHor e SPMD ElasticVH. Outrosim, traz considerações sobre alguns pontos negativos que cada um desses padrões pode enfrentar e de que forma a elasticidade pode mitigar tais pontos negativos. Este trabalho também apresenta uma revisão do estado da arte sobre como a elasticidade vem sendo implementada para cada um desses padrões, trazendo também como utilizar o modelo proposto e os *frameworks* em aplicações amplamente utilizadas em HPC, utilizados para cálculo de dinâmica molecular, algoritmo de busca, etc, mostrando uma infinidade de possibilidades de melhorias que a elasticidade traz.

Ao incorporar a elasticidade como um princípio central, esse modelo oferece uma abordagem flexível e adaptável para o design de aplicações paralelas, permitindo a exploração eficaz dos recursos computacionais disponíveis. A elasticidade adaptada ao modelo proporciona uma resposta dinâmica às variações na carga de trabalho, otimizando a utilização dos recursos e minimizando gargalos de desempenho. Isso não apenas acelera a execução das aplicações, mas também promove uma utilização mais eficiente dos recursos computacionais, resultando em uma abordagem mais sustentável e econômica. Ao adotar esse modelo de padrões elásticos, as equipes de desenvolvimento podem se concentrar mais na inovação e no aprimoramento das funcionalidades da aplicação, em vez de lidar constantemente com questões de escalabilidade e gerenciamento de recursos.

Além disso, a orientação desde o início do projeto é um dos pontos-chave desse modelo, capacitando os desenvolvedores a considerar a paralelização como parte integrante do processo de projeto. Essa abordagem não apenas simplifica a implementação de soluções paralelas, mas também reduz a necessidade de retrabalho durante as fases posteriores do desenvolvimento, economizando tempo e recursos. Padrões de programação paralela elásticos oferecem um direcionamento para o desenvolvimento de aplicações HPC. O modelo apresentado traz uma mudança de paradigma na abordagem do desenvolvimento paralelo.

Por fim, o direcionamento futuro visando a melhoria do modelo seria a definição de um questionário para o programador preencher e conseguir definir a forma como sua aplicação está organizada, permitindo assim uma escolha mais assertiva dos padrões a serem empregados. Além disso, também focar na implementação de aplicações para testar a eficácia do modelo e para avaliar as melhorias trazidas pela implementação utilizando padrões elásticos em comparação com os padrões tradicionais. Um exemplo disso seria a implementação das aplicações apresentadas no capítulo 5.

# Referências

AL-DHURAIBI, Y. et al. Autonomic vertical elasticity of docker containers with elasticdocker. In: IEEE. *2017 IEEE 10th international conference on cloud computing (CLOUD)*. [S.l.], 2017. p. 472–479. Citado na página 73.

ALVENTOSA, V. G.; MARTÍNEZ, G. M.; QUILIS, J. Ruper-lb: Load balancing embarrasingly parallel applications in unpredictable cloud environments. *arXiv preprint arXiv:2005.06361*, 2020. Citado na página 22.

AMDAHL, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In: *Proceedings of the April 18-20, 1967, spring joint computer conference*. [S.l.: s.n.], 1967. p. 483–485. Citado na página 31.

BONETTA, D. *The parallel event loop model and runtime: a parallel programming model and runtime system for safe event-based parallel programming*. Tese (Doutorado) — Università della Svizzera italiana, 2014. Citado na página 27.

BRAWER, S. *Introduction to parallel programming*. [S.l.]: Academic Press, 2014. Citado na página 15.

BU, Y. et al. The haloop approach to large-scale iterative data analysis. *The VLDB Journal*, Springer, v. 21, p. 169–190, 2012. Citado 3 vezes nas páginas 46, 71 e 72.

CABALLER, M. et al. Towards sla-driven management of cloud infrastructures to elastically execute scientific applications. In: *6th Iberian Grid Infrastructure Conference (IberGrid)*. [S.l.: s.n.], 2012. p. 207–218. Citado na página 56.

COUTINHO, E. F. et al. Elasticity in cloud computing: a survey. *annals of telecommunications-annales des télécommunications*, Springer, v. 70, p. 289–309, 2015. Citado na página 41.

CZAPPA, F. et al. Design-time performance modeling of compositional parallel programs. *Parallel Computing*, Elsevier, v. 108, p. 102839, 2021. Citado na página 16.

CZARNUL, P. *Parallel programming for modern high performance computing systems*. [S.l.]: CRC Press, 2018. Citado 4 vezes nas páginas 24, 26, 28 e 30.

DANELUTTO, M. et al. Algorithmic skeletons and parallel design patterns in mainstream parallel programming. *International Journal of Parallel Programming*, Springer, v. 49, p. 177–198, 2021. Citado na página 16.

DANELUTTO, M.; TORQUATI, M. Loop parallelism: a new skeleton perspective on data parallel patterns. In: IEEE. *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. [S.l.], 2014. p. 52–59. Citado 3 vezes nas páginas 46, 48 e 71.

DAROLT, D. D.; SOUZA, F. R. de; KOSLOVSKI, G. P. Explorando a elasticidade de nuvens iaas para reconfigurar dinamicamente aplicações n-camadas. *Revista Brasileira de Computação Aplicada*, v. 8, n. 2, p. 2–15, 2016. Citado 2 vezes nas páginas 43 e 61.

DOOLEY, J. F.; DOOLEY, J. F. Parallel design patterns. *Software Development, Design and Coding: With Patterns, Debugging, Unit Testing, and Refactoring*, Springer, p. 191–209, 2017. Citado na página 16.

DUBREUIL, M.; GAGNÉ, C.; PARIZEAU, M. Analysis of a master-slave architecture for distributed evolutionary computations. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, IEEE, v. 36, n. 1, p. 229–235, 2006. Citado na página 54.

ERL, T.; COPE, R.; NASERPOUR, A. *Cloud Computing Design Patterns (paperback)*. [S.l.]: Prentice Hall Press, 2017. Citado 4 vezes nas páginas 61, 67, 70 e 72.

ESTELLÉS, J. I. A. et al. A survey on malleability solutions for high-performance distributed computing. MDPI, 2022. Citado na página 52.

FAKHFAKH, F.; KACEM, H. H.; KACEM, A. H. Dealing with structural changes on provisioning resources for deadline-constrained workflow. *The Journal of Supercomputing*, Springer, v. 73, p. 2896–2918, 2017. Citado 2 vezes nas páginas 70 e 71.

FATÉMA, Z. B. et al. Toward a new massively distributed virtual machine based cloud micro-services team model for hpc: Spmd applications. *International Journal of Advanced Computer Science and Applications*, Science and Information (SAI) Organization Limited, v. 8, n. 8, 2017. Citado 4 vezes nas páginas 46, 48, 62 e 67.

FOSTER, I. *Designing and building parallel programs: concepts and tools for parallel software engineering*. [S.l.]: Addison-Wesley Longman Publishing Co., Inc., 1995. Citado 2 vezes nas páginas 15 e 75.

FURQUIM, G. A. *Balanceamento de cargas de aplicações SPMD em sistemas computacionais distribuídos*. Tese (Doutorado) — Universidade de São Paulo, 2006. Citado 2 vezes nas páginas 28 e 63.

GALANTE, G.; BONA, L. C. Supporting elasticity in openmp applications. In: IEEE. *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. [S.l.], 2014. p. 188–195. Citado 6 vezes nas páginas 9, 47, 48, 69, 88 e 89.

GALANTE, G.; BONA, L. C. E. de. A survey on cloud computing elasticity. In: IEEE. *2012 IEEE fifth international conference on utility and cloud computing*. [S.l.], 2012. p. 263–270. Citado na página 41.

GRIEBLER, D.; FERNANDES, L. Padrões e frameworks de programação paralela em arquiteturas multi-core. *11a ERAD-Escola Regional de Alto Desempenho*, 2011. Citado na página 27.

HERBST, N. R.; KOUNEV, S.; REUSSNER, R. H. Elasticity in cloud computing: What it is, and what it is not. In: *ICAC*. [S.l.: s.n.], 2013. v. 13, n. 2013, p. 23–27. Citado 2 vezes nas páginas 41 e 42.

HOUZEAUX, G. et al. Dynamic resource allocation for efficient parallel cfd simulations. *Computers & Fluids*, Elsevier, v. 245, p. 105577, 2022. Citado na página 48.

HUANG, K.-C.; WANG, F.-J. Design patterns for parallel computations of master-slave model. In: IEEE. *Proceedings of ICICS, 1997 International Conference on Information, Communications and Signal Processing. Theme: Trends in Information Systems Engineering and Wireless Multimedia Communications (Cat.* [S.l.], 1997. v. 3, p. 1508–1512. Citado na página 29.

INTEL. *Get Started with the Intel® oneAPI DPC++/C++ Compiler*. 2022. Consultado na INTERNET: 29 de set. de 2022. "https://www.intel.com/content/www/us/en/develop/documentation/get-started-with-dpcpp-compiler/top.html". Citado na página 14.

KAMIL, A. A. *Single program, multiple data programming for hierarchical computations*. [S.l.]: University of California, Berkeley, 2012. Citado na página 28.

KEHRER, S.; BLOCHINGER, W. Elastic parallel systems for high performance cloud computing: state-of-the-art and future directions. *Parallel Processing Letters*, World Scientific, v. 29, n. 02, p. 1950006, 2019. Citado na página 41.

KEHRER, S.; BLOCHINGER, W. A survey on cloud migration strategies for high performance computing. 2019. Citado na página 59.

KEHRER, S.; BLOCHINGER, W. Taskwork: a cloud-aware runtime system for elastic task-parallel hpc applications. 2019. Citado na página 43.

KEUTZER, K.; MATTSON, T. Our pattern language (opl): A design pattern language for engineering (parallel) software. In: CITESEER. *ParaPLoP Workshop on Parallel Programming Patterns*. [S.l.], 2009. v. 14, p. 10–1145. Citado na página 16.

KIESSLING, A. An introduction to parallel programming with openmp. In: *The University of Edinburgh, A Pedagogical Seminar (accessed 24 September 2020), URL: https://www. roe. ac. uk/ifa/postgrad/pedagogy/2009_kiessling. pdf*. [S.l.: s.n.], 2009. Citado na página 15.

LIMA, A. P. M. d. *Uma abordagem para o controle de elasticidade dinâmico voltada a sistemas cloud-network definidos por slices*. Dissertação (Mestrado) — Brasil, 2019. Citado na página 43.

LUZ, C. S. F. d. *Metodologia e ferramentas para paralelização de laços perfeitamente aninhados com processamento heterogêneo*. Tese (Doutorado) — Universidade de São Paulo, 2018. Citado na página 31.

MARTÍN-ÁLVAREZ, I. et al. Dynamic spawning of mpi processes applied to malleability. *The International Journal of High Performance Computing Applications*, SAGE Publications Sage UK: London, England, p. 10943420231176527, 2023. Citado na página 15.

MARTÍN, G. et al. Enhancing the performance of malleable mpi applications by using performance-aware dynamic reconfiguration. *Parallel Computing*, Elsevier, v. 46, p. 60–77, 2015. Citado 3 vezes nas páginas 47, 48 e 64.

MATTSON, T. G.; SANDERS, B.; MASSINGILL, B. *Patterns for parallel programming*. [S.l.]: Pearson Education, 2004. Citado 14 vezes nas páginas 16, 18, 19, 23, 24, 25, 26, 28, 30, 31, 32, 33, 40 e 49.

MCCOOL, M.; REINDERS, J.; ROBISON, A. *Structured parallel programming: patterns for efficient computation*. [S.l.]: Elsevier, 2012. Citado 11 vezes nas páginas 9, 15, 16, 18, 29, 34, 35, 36, 38, 40 e 49.

MCCOOL, M. D. Structured parallel programming with deterministic patterns. *Proc. HotPar*, 2010. Citado 3 vezes nas páginas 16, 26 e 36.

MELL, P.; GRANCE, T. et al. The nist definition of cloud computing. Computer Security Division, Information Technology Laboratory, National . . . , 2011. Citado na página 42.

MO-HELLENBRAND, A. *Resource-Aware and Elastic Parallel Software Development for Distributed-Memory HPC Systems*. Tese (Doutorado) — Technische Universität München, 2019. Citado 6 vezes nas páginas 9, 47, 48, 65, 77 e 78.

MOR, S. D. K. Escalonamento on-line eficiente de programas fork-join recursivos do tipo divisão e conquista em mpi. 2010. Citado na página 32.

MOREIRA, G. A. S. Vertelastic: um módulo de decisão para explorando elasticidade vertical no autoelastic. Universidade do Vale do Rio dos Sinos, 2018. Citado 4 vezes nas páginas 42, 45, 48 e 59.

NEERAJ, N. *Mastering Apache Cassandra*. [S.l.]: Packt Publishing Ltd, 2015. Citado na página 59.

NGUYEN, M. et al. A black-box fork-join latency prediction model for data-intensive applications. *IEEE Transactions on Parallel and Distributed Systems*, IEEE, v. 31, n. 9, p. 1983–2000, 2020. Citado 4 vezes nas páginas 47, 48, 69 e 82.

PACHECO, P. *An introduction to parallel programming*. [S.l.]: Elsevier, 2011. Citado na página 14.

PERICÀS, M. Elastic places: An adaptive resource manager for scalable and portable performance. *ACM Transactions on Architecture and Code Optimization (TACO)*, ACM New York, NY, USA, v. 15, n. 2, p. 1–26, 2018. Citado na página 63.

PROGRAMMIERUNG, P. *Identification of Suitable Parallelization Patterns for Sequential Programs*. Tese (Doutorado) — Iowa State University, 2021. Citado 2 vezes nas páginas 14 e 15.

QIN, J.; MA, L.; NIU, J. Thbase: A coprocessor-based scheme for big trajectory data management. *Future Internet*, MDPI, v. 11, n. 1, p. 10, 2019. Citado na página 62.

RAJAN, D. et al. Converting a high performance application to an elastic cloud application. In: IEEE. *2011 IEEE Third International Conference on Cloud Computing Technology and Science*. [S.l.], 2011. p. 383–390. Citado 2 vezes nas páginas 54 e 58.

RAJAN, D. et al. Case studies in designing elastic applications. In: IEEE. *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*. [S.l.], 2013. p. 466–473. Citado 3 vezes nas páginas 45, 48 e 57.

RIGHI, R. da R. Elasticidade em cloud computing: conceito, estado da arte e novos desafios. *Revista Brasileira de Computação Aplicada*, v. 5, n. 2, p. 2–17, 2013. Citado 2 vezes nas páginas 41 e 42.

RIGHI, R. da R. et al. A lightweight plug-and-play elasticity service for self-organizing resource provisioning on parallel applications. *Future Generation Computer Systems*, Elsevier, v. 78, p. 176–190, 2018. Citado 3 vezes nas páginas 44, 48 e 57.

RIZK, A.; POLOCZEK, F.; CIUCU, F. Computable bounds in fork-join queueing systems. In: *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. [S.l.: s.n.], 2015. p. 335–346. Citado 2 vezes nas páginas 68 e 69.

RODRIGUES, V. F. Autoelastic: explorando a elasticidade de recursos de computação em nuvem para a execução de aplicações de alto desempenho iterativa. Universidade do Vale do Rio dos Sinos, 2016. Citado 5 vezes nas páginas 42, 45, 48, 57 e 60.

RODRIGUES, V. F. et al. Towards combining reactive and proactive cloud elasticity on running hpc applications. In: *IoTBDS*. [S.l.: s.n.], 2018. p. 261–268. Citado 7 vezes nas páginas 9, 43, 44, 48, 59, 85 e 86.

RODRIGUES, V. F. et al. Towards enabling live thresholding as utility to manage elastic master-slave applications in the cloud. *Journal of Grid Computing*, Springer, v. 15, p. 535–556, 2017. Citado 3 vezes nas páginas 44, 48 e 59.

ROSSAINZ-LÓPEZ, M. et al. Implementation of the pipeline parallel programming technique as an hlpc: Usage, usefulness and performance. *Annals of Multicore and GPU Programming*, v. 4, n. 1, p. 9–22, 2017. Citado na página 26.

ROSSI, F. Auto-scaling policies to adapt the application deployment in kubernetes. In: *ZEUS*. [S.l.: s.n.], 2020. p. 30–38. Citado na página 56.

SAMADI, M. et al. Paraprox: Pattern-based approximation for data parallel applications. In: *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*. [S.l.: s.n.], 2014. p. 35–50. Citado na página 37.

SCHMALFUSS, M. F. *Uma interface para escalonamento de algoritmos iterativos em C++*. Dissertação (Mestrado) — Universidade Federal de Pelotas, 2020. Citado na página 37.

SENA, A. C. et al. Autonomic malleability in iterative mpi applications. In: IEEE. *2013 25th International Symposium on Computer Architecture and High Performance Computing*. [S.l.], 2013. p. 192–199. Citado na página 15.

SILVA, L. M.; BUYYA, R. Parallel programming models and paradigms. In: *High Performance Cluster Computing Programming and Applications*. [S.l.]: Prentice Hall PTR, 1999. p. 4–27. Citado 4 vezes nas páginas 24, 26, 28 e 29.

SUTTER, H. et al. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's journal*, v. 30, n. 3, p. 202–210, 2005. Citado na página 14.

VERGARA, G. F. Arquitetura de um controlador de elasticidade para nuvens federadas. 2017. Citado na página 43.

VOSS, M.; ASENJO, R.; REINDERS, J. *Pro TBB: C++ parallel programming with threading building blocks*. [S.l.]: Springer, 2019. Citado na página 27.

WHITE, T. *Hadoop: The definitive guide*. [S.l.]: "O'Reilly Media, Inc.", 2012. Citado na página 39.

WILKINSON, P. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers, 2/E*. [S.l.]: Pearson Education India, 2006. Citado 2 vezes nas páginas 26 e 32.

WRZESINSKA, G. et al. Fault-tolerance, malleability and migration for divide-and-conquer applications on the grid. In: IEEE. *19th IEEE International Parallel and Distributed Processing Symposium*. [S.l.], 2005. p. 10–pp. Citado 2 vezes nas páginas 48 e 69.

ZHAO, J.; GAO, X.; LI, Y. Research on elastic extension of multi type resources for openmp program. In: IEEE. *2022 IEEE 24th Int Conf on High Performance Computing & Communications; 8th Int Conf on Data Science & Systems; 20th Int Conf on Smart City; 8th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys)*. [S.l.], 2022. p. 971–978. Citado na página 48.