UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
Campus de São José do Rio Preto

Ricardo Trivizan Fares

Simulação de Sistemas Exascale usando Time Warp

São José do Rio Preto

2023

Ricardo Trivizan Fares

Simulação de Sistemas Exascale usando Time Warp

Monografia apresentada como parte dos requisitos para obtenção do título de Bacharel em Ciência da Computação, junto ao Departamento de Ciências de Computação e Estatística do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista "Júlio de Mesquita Filho", Câmpus de São José do Rio Preto.

Financiadora: FAPESP 2022/15807-6

Universidade Estadual Paulista "Júlio de Mesquita Filho" - (UNESP)

Instituto de Biociências, Letras e Ciências Exatas - (IBILCE)

Departamento de Ciências de Computação e Estatística

Bacharelado em Ciência da Computação

Orientador: Prof. Dr. Aleardo Manacero Júnior

São José do Rio Preto

2023

F222s

Fares, Ricardo Trivizan
Simulação de sistemas exascale usando time warp / Ricardo Trivizan Fares. -- São José do Rio Preto, 2023
71 p.

Trabalho de conclusão de curso (Bacharelado - Ciência da Computação) - Universidade Estadual Paulista (Unesp), Instituto de Biociências Letras e Ciências Exatas, São José do Rio Preto
Orientador: Aleardo Manacero Júnior

1. Ciência da Computação. 2. Sistemas de Computação. 3. Sistemas Distribuídos. 4. Protocolos de Sincronização. I. Título.

Sistema de geração automática de fichas catalográficas da Unesp. Biblioteca do Instituto de Biociências Letras e Ciências Exatas, São José do Rio Preto. Dados fornecidos pelo autor(a).

Essa ficha não pode ser modificada.

Ricardo Trivizan Fares

Simulação de Sistemas Exascale usando Time Warp

Monografia apresentada como parte dos requisitos para obtenção do título de Bacharel em Ciência da Computação, junto ao Departamento de Ciências de Computação e Estatística do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista "Júlio de Mesquita Filho", Câmpus de São José do Rio Preto.

Financiadora: FAPESP 2022/15807-6

Banca Examinadora:

Prof. Dr. Aleardo Manacero Júnior
UNESP - Câmpus de São José do Rio Preto
Orientador

Prof. Dra. Renata Spolon Lobato
UNESP - Câmpus de São José do Rio Preto

Prof. Dr. Lucas Correia Ribas
UNESP - Câmpus de São José do Rio Preto

São José do Rio Preto
16 de novembro de 2023

Agradecimentos

Agradeço aos meus pais, Ahmed Hussein Fares e Ana Maris Trivizan Fares, que estiveram ao meu lado fornecendo todo o suporte necessário durante esta jornada, e ao meu irmão Eduardo Trivizan Fares.

Aos amigos de graduação que estiveram comigo em diversos momentos. Em especial, aqueles que conviveram comigo em todo o momento durante a minha graduação: João Barroso e Matheus Augusto.

Aos professores Prof. Dr. Aleardo Manacero Júnior, Profa. Dra. Renata Spolon Lobato e Prof. Dr. Lucas Correia Ribas, por todas as contribuições a mim e a este trabalho.

À Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP) pelo fomento do projeto através da bolsa de iniciação científica, sob o processo Nº 2022/15807-6.

Resumo

A análise de sistemas computacionais *exascale*, quando feita por sua simulação exige enorme custo computacional e de tempo que inviabiliza sua análise em simuladores sequenciais dependendo do nível de detalhamento da simulação. À vista disso, a simulação distribuída torna propício a análise de tais sistemas complexos. Entretanto, a partilha da simulação em um sistema distribuído origina o problema da sincronização. Assim, este trabalho propõe a extensão do simulador *iSPD* (*iconic Simulator of Parallel and Distributed Systems*) usando o protocolo de sincronização otimista *Time Warp* viabilizando a simulação de sistemas *exascale* em tempo hábil.

Palavras-chave: Avaliação, Medição e Predição de Desempenho, Modelagem e Simulação de Arquiteturas, Sistemas Distribuídos, Protocolos de Sincronização Otimistas.

Abstract

The analysis of exascale computing systems, when performed through simulation, requires large computational effort and time making its analysis unfeasible in sequential simulators depending on the level of detail. Using distributed simulation makes possible the analysis of such complex systems. Nonetheless, performing the simulation in a distributed system gives rise to synchronization problems. Thus, in this work we proporse an extension of the simulator iSPD (iconic Simulator of Parallel and Distributed Systems) using the Time Warp optimistic synchronization protocol enabling the simulation of exascale systems in a timely manner.

Keywords: *Performance Evaluation, Measurement and Prediction, Modeling and Simulation of Architectures, Distributed Systems, Optimistic Synchronization Protocols.*

Sumário

Lista de ilustrações

Lista de tabelas

Lista de algoritmos

Lista de abreviaturas e siglas

CMB	*Chandy-Misra-Bryant*
CODES	*CO-Design of multilayer Exascale Storage*
CSP	*Concurrent Sequential Process*
CSS	*Copy State Saving*
DAG	*Direct Acyclic Graph*
FIFO	*First In First Out*
GSPD	Grupo de Sistemas Paralelos e Distribuídos
GVT	*Global Virtual Time*
iSPD	*iconic Simulator of Parallel and Distributed Systems*
ISS	*Incremental State Saving*
JVM	*Java Virtual Machine*
LAN	*Local Area Network*
LLNL	*Lawrence Livermore National Laboratory*
LP	*Logical Process*
LVT	*Local Virtual Time*
MPI	*Message Passing Interface*
PDES	*Parallel Discrete Event Simulation*
PSS	*Periodic State Saving*
RC	*Reverse Computation*
ROSS	*Rensselaer's Optimistic Simulation System*
SSS	*Sparse State Saving*

1 Introdução

Os sistemas *exascale*, sistemas capazes de executar 10^{18} operações de ponto flutuante por segundo, tornou-se uma realidade recente com o advento do *Hewlett Packard Enterprise Frontier*. Assim, tal quebra de barreira computacional permitirá que pesquisadores conduzam seus estudos a novas etapas, antes inviáveis no universo *petascale* (CHANG et al., 2023).

Estes sistemas possuem custos de utilização e manutenção altíssimos, de modo que, a identificação dos gargalos de execução de aplicações torna-se uma tarefa importante. O estudo destes gargalos pode ser realizado por meio de simulações, que fornecem custo e precisão satisfatória, em comparação com *benchmarking* que necessita da utilização do sistema real, e modelos analíticos que em diversos casos necessita de múltiplas hipóteses sobre o modelo sendo solucionado (JAIN, 1991).

Entretanto, os modelos de sistemas *exascale* apresentam centenas de milhares até dezenas de milhões de componentes simuláveis, de forma que os atuais simuladores sequenciais, dependendo do nível de detalhamento da simulação, tornam inviável o estudo destes modelos em tempo hábil. Isto posto, este trabalho propõe a extensão do simulador iSPD (*iconic Simulator of Parallel and Distributed Systems*) (MANACERO et al., 2012) para torná-lo apto de realizar simulações paralelas e distribuídas.

Contudo, a partilha da simulação em um sistema distribuído origina o problema da sincronização, em que os eventos sendo processados em diferentes núcleos físicos podem quebrar a *restrição de causalidade*. Logo, para que tal restrição seja mantida durante a simulação, utilizaremos o protocolo de sincronização otimista *Time Warp*.

1.1 Motivação

Atualmente, as simulações de larga escala presentes na literatura como (MUBARAK et al., 2012), (WOLFE et al., 2016), (WOLFE et al., 2018) e (MUBARAK et al., 2014) focam na simulação e análise da topologia de rede de interconexão dos nós de processamento. Contudo, nossa intenção é possibilitar a simulação de sistemas de larga escala com o intuito de analisar o comportamento dos nós de processamento com relação, principalmente, aos algoritmos de escalonamento de tarefas.

Portanto, como simuladores de larga escala com este intuito estão presentes infimamente na literatura, esta ausência torna-se a motivação de prover um simulador que seja capaz de fornecer resultados precisos do comportamento de tais algoritmos em uma simulação contendo centenas de milhares até dezenas de milhões de objetos simuláveis.

1.2 Objetivos

O processo de simulação de sistemas *exascale* pode tornar o tempo de execução inviável, dependendo do nível de detalhamento da simulação. Por essa razão, o objetivo deste trabalho é paralelizar o motor de simulação do simulador iSPD para que torne-o capaz de realizar simulações com imensa quantidade de objetos sendo simulados em tempo hábil.

1.3 Justificativa

Como anteriormente especificado, o uso de simulações permite que a análise de sistemas seja feita com custo e precisão satisfatória. Embora, diversos simuladores já foram propostos e validados no intuito de simular sistemas de larga escala, estes tem a finalidade da análise da topologia de interconexão, de forma que, nenhum trata em qualquer aspecto o escalonamento de tarefas das aplicações.

Desta forma, com este trabalho torna-se interessante os resultados que podem ser obtidos uma vez que simularemos outros aspectos do modelo, como o escalonamento citado anteriormente.

1.4 Metodologia

Neste trabalho, o processo de paralelização do motor de simulação foi dividido nas seguintes quatro etapas.

1. Revisão bibliográfica sobre simuladores de eventos discretos paralelo;
2. Adequação do modelo de simulação antes sequencial para um modelo paralelo;
3. Otimização do modelo de simulação paralelo;
4. Execução de testes de desempenho e comparação dos diferentes modos de execução.

Na primeira etapa dedica-se ao entendimento do funcionamentos dos simuladores de eventos discretos paralelos, principalmente, naqueles que adotam o protocolo de sincronização otimista *Time Warp*. Em foco, busca-se entender a implementação, às vantagens e quais são as limitações.

Na segunda etapa, uma vez que temos o entendimento do funcionamento de simuladores de eventos discretos paralelos, realizaremos a transposição da implementação de um motor sequencial para um motor paralelo em que, principalmente, os desafios serão o gerenciamento das informações que agora estão dispersos em uma memória distribuída.

Na terceira etapa, otimizações no modelo serão realizadas no intuito de *aumento de desempenho* e *redução do uso de memória*, sendo esses alcançados pela redução da quantidade de eventos processados e diminuição dos estados dos processos lógicos.

Na quarta etapa, testes de desempenho serão realizados entre os diferentes modos de execução: *sequencial*, *paralelo* e *distribuído*, de tal forma que, podemos entender qual modo é mais vantajoso em cada caso e quais são os parâmetros principais que afetam o desempenho do simulador.

1.5 Exequibilidade

Neste trabalho, conhecimentos de *programação orientada a objetos*, *estruturas de dados*, *projeto e análise de algoritmos*, *arquiteturas de computadores*, *sistemas operacionais*, *compiladores* e *sistemas distribuídos* são fundamentais para a execução deste projeto, com todos esses temas abordados durante a graduação.

O desenvolvimento do projeto ocorreu dentro do Grupo de Sistemas Paralelos e Distribuídos (GSPD) com as instalações necessárias para o projeto que está sendo desenvolvido como, por exemplo, o *cluster* GSPD com mais de 100 núcleos físicos.

Portanto, o projeto possui toda a estrutura necessária para o seu desenvolvimento.

1.6 Organização do Texto

Este trabalho está dividido em cinco capítulos. Neste Capítulo 1 uma introdução ao projeto foi dada contendo a motivação, objetivos e sua justificativa. No Capítulo 2, uma revisão bibliográfica sobre simuladores de eventos discretos paralelos será abordada.

No Capítulo 3, será tratado a metodologia abordada e as etapas do desenvolvimento do projeto como, por exemplo, a arquitetura e a implementação do simulador.

No Capítulo 4, os testes e resultados de simulação e de desempenho do simulador serão abordados a partir da utilização de diferentes modos de execução do simulador, principalmente, o modo de execução otimista. Por fim, no Capítulo 5 serão apresentadas as conclusões e trabalhos futuros.

2 Revisão Bibliográfica

Neste capítulo, serão apresentados os conceitos fundamentais sobre simulação distribuída, o qual inclui a introdução ao problema da sincronização, protocolos de sincronização e simuladores de eventos discretos paralelos presentes na literatura.

Por fim, apresentaremos os principais trabalhos do estado da arte com relação à simulação de sistemas de larga escala utilizando simuladores de eventos discretos paralelos, principalmente, com aqueles que utilizam o protocolo otimista *Time Warp*.

2.1 Simulação de Eventos Discretos

A simulação consiste na imitação de um processo do mundo real ou imaginário sobre o tempo, compreendendo a geração de um histórico artificial do sistema e a execução de inferências sobre as características do sistema real (BANKS, 1999). Ainda, a simulação é uma técnica para obtenções de resultados sem a necessidade de intervenção do sistema real e, com isso, aplicando a análise estatística nos resultados obtém-se contribuições nas tomadas de decisão.

Uma simulação de eventos discretos tem a finalidade de capturar o comportamento de um sistema discreto real ou imaginário durante um período de tempo. Contudo, diferentemente de outras simulações que evoluem continuamente no tempo, em uma simulação de eventos discretos as mudanças no sistema ocorrem em instantes distintos no tempo a partir da ocorrência de eventos (FUJIMOTO, 2016).

Além disso, modelos de simulação podem ser *determinísticos* ou *estocásticos*. Os modelos determinísticos não possuem quaisquer variáveis aleatórias e, portanto, em múltiplas execuções dos modelos obtém-se o mesmo resultado. Por outro lado, modelos estocásticos possuem variáveis aleatórias como, por exemplo, a distribuição do tempo entre chegada dos eventos.

Isto posto, em um programa de simulação de eventos discretos sequencial, em geral, incluem-se três conceitos fundamentais: as **variáveis de estado** que capturam o estado do sistema sendo modelado, uma **fila de eventos** contendo todos os eventos pendentes que serão processados e um **relógio global** que denota quão longe a simulação progrediu (FUJIMOTO, 1990).

Além disso, em uma simulação de eventos discretos, mudanças no estado da simulação ocorrem *somente* através do processamento de tais eventos. Os eventos contém uma *marca de tempo*, denotada $t(E)$ para um evento E, que representa o instante no tempo de simulação em que a transição de estado ocorre quando E é processado. Deste modo, em geral, os simuladores

de eventos discretos sequenciais são implementados a partir do seguinte algoritmo.

Algoritmo 1 Laço Principal de um Simulador de Eventos Discretos Sequencial

```
enquanto fila não está vazia faça
    Remova o primeiro evento da fila
    Atualize o relógio para o instante marcado no evento
    Processe o evento
fim enquanto
```

Desta forma, tendo em vista o paradigma de execução de um simulador sequencial apresentado, temos que o simulador repetidamente remove o evento com menor marca de tempo e processa-o. Neste paradigma de execução, é crucial que o simulador sempre selecione o evento com a menor marca de tempo E_{min} a partir da lista de eventos, como sendo o próximo evento a ser processado, pois se fosse escolhido outro evento contendo uma marca de tempo maior, digamos E_X, seria então possível que E_X modificasse as variáveis de estados usados por E_{min}, de modo que, isto indicaria um sistema em que o futuro pudesse alterar o passado.

Desse modo, esta ordem de processamento é claramente inaceitável, de tal forma, que erros desta natureza são chamados de *erros de causalidade* e, portanto, temos uma definição precisa para tal (FUJIMOTO, 1990).

Definição 2.1.1 (Restrição de Ordem)**:** *Sejam E_1 e E_2 dois eventos quaisquer, e $\rightarrow$ a relação* **acontece antes** *com a mesma semântica dada em (LAMPORT, 1978). Se $E_1 \rightarrow E_2$, então E_1 deve ser processado antes de E_2.*

Se a restrição de ordem for mantida durante a simulação, então não ocorrerá *erros de causalidade*. Desta forma, como em simulações de eventos discretos cada evento E tem uma marca de tempo $t(E)$, então a fila de eventos pode ser implementada por uma fila ordenada com o evento com menor marca de tempo E_{min} sendo o primeiro da fila garantindo, portanto, a restrição de ordem.

2.2 Simulação Distribuída

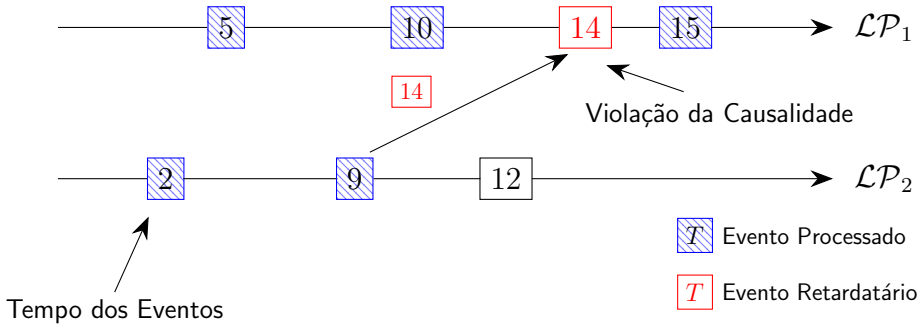
À medida que cresce a necessidade de simular sistemas cada vez maiores e mais complexos, uma abordagem sequencial pode tornar inviável a simulação. Em vista disso, a simulação de eventos discretos paralela (PDES, do inglês *Parallel Discrete Event Simulation*), algumas vezes chamadas de simulação distribuída, refere-se a execução de uma única simulação de eventos discretos em um computador paralelo, com a finalidade de acelerar a execução da simulação (FUJIMOTO, 1990) e (FUJIMOTO, 2016).

Uma simulação de eventos discretos paralela pode ser vista como uma coleção de simulações de eventos discretos sequenciais distribuídos por diferentes núcleos físicos que interagem a partir do envio e recebimento de mensagens temporais, de tal forma, que cada mensagem está representando um evento que é enviado de um simulador a outro. Neste caso, cada simulador é referido como **processo lógico** ou $\mathcal{LP}$ (do inglês, *Logical Process*) (JAFER; LIU; WAINER, 2013). Contudo, a distribuição dos processos lógicos ocasiona o problema da sincronização, definida a seguir.

Definição 2.2.1 (Problema da Sincronização)**:** *A execução concorrente dos eventos pelos processos lógicos em uma simulação distribuída despreza a ordem de causalidade entre os eventos, de tal forma que, resultados inesperados podem ocorrer.*

A seguir, na Figura (2.2.1) um exemplo deste problema é representado.

Figura 2.2.1 – Exemplo de Violação de Causalidade.



Fonte: Elaborado pelo autor.

Neste caso, temos que o processo lógico $\mathcal{LP}_2$ envia uma mensagem com marca de tempo 14 ao processo lógico $\mathcal{LP}_1$, contudo, tal processo lógico está a frente deste tempo, pois já processou o evento com marca de tempo 15. Desta forma, é necessário um procedimento para resolver esta situação, desde que o evento que quebrou a restrição de causalidade, denominado **evento retardatário**, precisa ser processado em $\mathcal{LP}_1$ antes do evento com marca de tempo 15.

Desta forma, muito dos trabalhos em simulações de eventos discretos paralela têm focado na sincronização da simulação para resolver problemas de sincronização, como mostrado na Figura (2.2.1). Assim, demonstrou-se que para garantir a correta sincronização é necessário e suficiente que o sistema de simulação paralela obedeça a seguinte condição (FUJIMOTO, 1990) e (JAFER; LIU; WAINER, 2013).

Definição 2.2.2 (Restrição de Causalidade Local)**:** *Uma simulação de eventos discretos, consistindo de múltiplos processos lógicos $\mathcal{LP}_1, \mathcal{LP}_2, \ldots, \mathcal{LP}_n$, que interagem exclusivamente por troca de mensagens temporais obedece a restrição de causalidade local se, e somente se, cada processo lógico $\mathcal{LP}_i$ processar os eventos em ordem não decrescente.*

Por fim, para satisfazer a restrição de causalidade local, inúmeras técnicas de sincronização foram propostas as quais caem em dois grandes grupos, denominados: *conservativos*, os quais evitam *estritamente* qualquer violação da restrição de causalidade e, os *otimistas* os quais permitem a quebra da causalidade e recuperam-se a partir disto. Além disso, também existem os mecanismos híbridos que misturam ambos os protocolos.

2.2.1 Protocolos de Sincronização Conservativos

Os primeiros protocolos de sincronização foram conservativos, nestes protocolos evita-se *estritamente* qualquer quebra de restrição de causalidade dos eventos. Na técnica conservativa isto é obtido garantindo que um processo lógico $\mathcal{LP}_i$ com relógio virtual local LVT_i (LVT, do inglês *Local Virtual Time*) não receberá qualquer evento E com $t(E) < LVT_i$ (VEE; HSU, 2007).

Definição 2.2.3 (Relógio Virtual Local (LVT))**:** *O relógio virtual local LVT_i de um processo lógico $\mathcal{LP}_i$ corresponde à marca de tempo do último evento processado por este processo lógico.*

Inúmeros protocolos conservativos foram propostos, contudo, em especial, o precursor dos protocolos conservativos foi o protocolo *Chandy-Misra-Bryant* (CMB) (CHANDY; MISRA, 1979) e (BRYANT, 1977). Este protocolo usa um mecanismo que bloqueia a execução de um processo lógico até garantir que nenhum evento com marca de tempo menor será recebido posteriormente. Isto requere a especificação de um valor L denominado, *lookahead* que é um valor dependente do modelo sendo simulado.

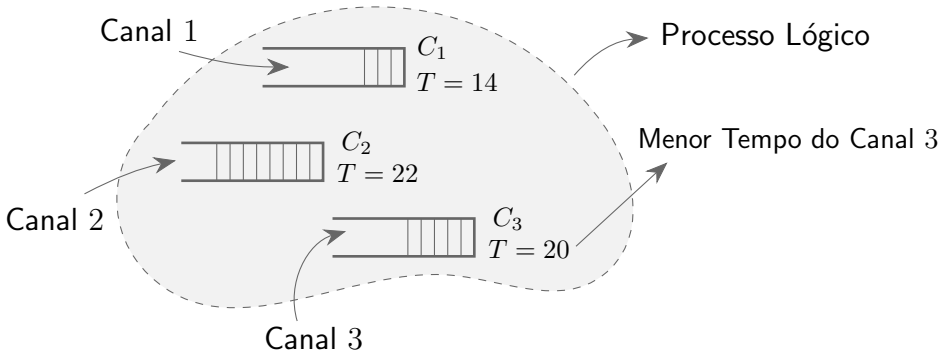
A partir disto, se um processo lógico $\mathcal{LP}_i$ está no instante T no tempo de simulação e, o *lookahead* é L, então qualquer mensagem recebida posteriormente por este processo lógico deve ter marca de tempo de pelo menos $T + L$, isto é, o evento E a ser recebido, se houver, deve obedecer à restrição $t(E) \geq T + L$.

Ainda, neste protocolo necessita-se que os canais de comunicação entre processos lógicos comunicantes sejam definidos antecipadamente com cada canal funcionando no modo *FIFO* (do inglês, *First In First Out*). Além disso, para determinar quando um evento pode ser processado de maneira segura, é necessário que a sequência de marcas de tempo $t(E_1), t(E_2), \ldots, t(E_n)$ dos eventos enviados a um canal seja não decrescente, isto é, $t(E_1) \leq t(E_2) \leq \cdots \leq t(E_n)$.

Esta última condição garante que a marca de tempo da última mensagem recebida $t(E_1)$ funcione como um limitante inferior para a marca de tempo de qualquer mensagem posteriormente recebida. Desta forma, cada processo lógico $\mathcal{LP}_i$ seleciona como próximo evento a ser processado aquele com a menor marca de tempo entre todas as filas de entrada Contudo, se as filas estiverem vazias, então o processo é bloqueado e aguarda a chegada de mensagens. Assim, garante-se que não haverá quebras de restrição de causalidade.

A seguir, na Figura (2.2.2) é representado um processo lógico neste protocolo com três canais de comunicação.

Figura 2.2.2 – Representação de um Processo Lógico no Protocolo CMB.



Fonte: Elaborado pelo autor.

Embora, o protocolo CMB resolva o problema da sincronização, este sofre de algumas limitações, a primeira delas é que o fato de os processos lógicos bloquearem à espera de mensagens pode causar situações de *deadlocks*. Os *deadlocks* ocorrem quando existe um ciclo de processos lógicos bloqueados à espera de mensagens e cada um está bloqueado devido a outro processo no ciclo (FUJIMOTO, 1990) e (MISRA, 1986). Neste caso, mecanismos de envio de mensagens nulas são utilizados para prevenir tal situação (JAFER; LIU; WAINER, 2013).

Além disso, uma segunda limitação é a necessidade de um bom *lookahead*. O *lookahead* é a quantidade de tempo que um processo lógico pode "investigar"o futuro, se este *lookahead* for baixo, então os processos lógicos bloquearão frequentemente para garantir que as mensagens podem ser processadas seguramente. Ainda, demonstrou-se que valores maiores de *lookahead* aumenta o desempenho da simulação (PERUMALLA, 2006) e (NKETSA; KHALIFA, 2001).

Por fim, embora o protocolo CMB seja o precursor, novos protocolos conservativos mais flexíveis e com menor *overhead* de sincronização foram propostos na literatura como, por exemplo, o protocolo *Bounded Lag* que tem o intuito de incorporar otimismo em um mecanismo totalmente conservativo (DAS, 1996), e o protocolo YAWNS, que funciona utilizando um mecanismo de processamento em janelas (DICKENS et al., 1996).

2.2.2 Protocolos de Sincronização Otimistas

Os protocolos otimistas detectam e recuperam-se dos erros de causalidade, não sendo necessário evitar *estritamente* eles. Desta forma, diferentemente dos protocolos conservativos, a estratégia otimista não é garantir quando é seguro processar um evento, em vez disso eles

determinam quando um erro de causalidade ocorreu e, invoca um procedimento para recuperar o estado consistente (FUJIMOTO, 1990).

O termo otimista refere-se ao fato de que os processos lógicos executam os eventos supondo que quebras na restrição da causalidade não irão ocorrer. Ainda, da mesma forma que nos protocolos conservativos haverá um grupo de processos lógicos comunicantes, porém diferentemente desse, os processos lógicos não precisam estabelecer canais de comunicação antecipadamente e não é necessário que a comunicação seja confiável (QUAGLIA; CORTELLESSA; CICIANI, 1999) e (BAUER et al., 2005).

Isto posto, uma das principais vantagens dos protocolos otimistas é que este permite a simulação explorar o máximo de paralelismo possível em situações que *talvez* ocorreria quebra de causalidade, mas que não ocorrem de fato. Assim, em geral, os protocolos otimistas possuem maior desempenho que os protocolos conservativos, pois nestes casos enquanto os protocolos otimistas avançam na simulação, os protocolos conservativos bloqueariam seus processos lógicos até garantir se o evento é seguro de ser processado.

O precursor dos protocolos otimistas é o protocolo *Time Warp* e, embora haja variações deste, o *Time Warp* é o protocolo otimista mais usado atualmente (JEFFERSON, 1985), (FUJIMOTO, 1990) e (SPOLON, 2001). Neste trabalho, o motor de simulação proposto usa o protocolo otimista *Time Warp* e, portanto, abordaremos este protocolo.

2.2.2.1 *Time Warp*

O mecanismo do protocolo *Time Warp* é baseado no paradigma do *Virtual Time*, em que este é um sistema de coordenadas temporal unidimensional cujos instantes de tempo são representados por números reais, incluindo o $+\infty$ e, ordenados totalmente pela relação $<$ [1]. Este sistema de coordenadas é utilizado para medir o progresso da simulação e para atividades de sincronização (JEFFERSON, 1985).

O *Virtual Time* é essencial, pois impõe regras fundamentais sobre as marcas de tempo das mensagens trocadas entre os processos lógicos, de modo que, caso estas regras não sejam obedecidas não é possível garantir sincronismo. A seguir, as regras são apresentadas.

1. O *tempo virtual de envio* de cada mensagem deve ser menor que o seu *tempo virtual de recebimento*.
2. O *tempo virtual de recebimento* de cada mensagem em um mesmo processo deve ser menor que o *tempo virtual de recebimento* do próximo evento neste mesmo processo.

Ambas as regras são exatamente aquelas definidas para os relógios de *Lamport* em (LAMPORT, 1978). Ainda, segundo (JEFFERSON, 1985) as regras impõe que a relação de

[1] Geralmente, os simuladores otimistas implementam o *Virtual Time* com uma relação de ordem parcial e, utilizam um mecanismo de *tie-breaking* (desempate) para se aproximar de uma ordem total.

causalidade entre dois eventos, se houver, sempre apontará em direção ao crescimento do tempo virtual.

Isto posto, o *Time Warp* segue tais regras e, para isso, possui dois mecanismos fundamentais.

- **Mecanismo de Controle Local**: O objetivo é garantir que os eventos sejam executados na ordem correta;
- **Mecanismo de Controle Global**: É responsável por realizar gerenciamento de memória, funções de E/S, tratamento de erros e detecção do fim da simulação.

Mecanismo de Controle Local

Primeiramente, para entendermos o funcionamento do controle local, devemos compreender quais são as estruturas de dados fundamentais em cada processo lógico no *Time Warp* (JEFFERSON, 1985) e (SPOLON, 2001).

- **Identificador do Processo Lógico**[2]: Identifica o processo lógico, unicamente, no sistema distribuído;
- **Relógio Virtual Local (LVT)**[3]: Equivalente à marca de tempo do último evento processado por esse processo lógico;
- **Estado**: Armazena o estado atual deste processo lógico;
- **Fila de Estados**[4]: Armazena cópia dos estados recentes deste processo lógico. Desta forma, é possível realizar o mecanismo de *rollback* salvando os estados periodicamente;
- **Fila de Entrada de Mensagens**: Armazena as mensagens recebidas por este processo lógico em ordem não decrescente pelo seu *tempo virtual de recebimento*;
- **Fila de Saída de Mensagens**: Armazena as cópias das mensagens enviadas por este processo lógico em ordem não decrescente pelo seu *tempo virtual de envio*. Nesta fila todas as mensagens possuem marca de tempo menor ou igual ao LVT do processo lógico e, são denominadas *antimensagens* e, é necessária para o mecanismo de *rollback*.

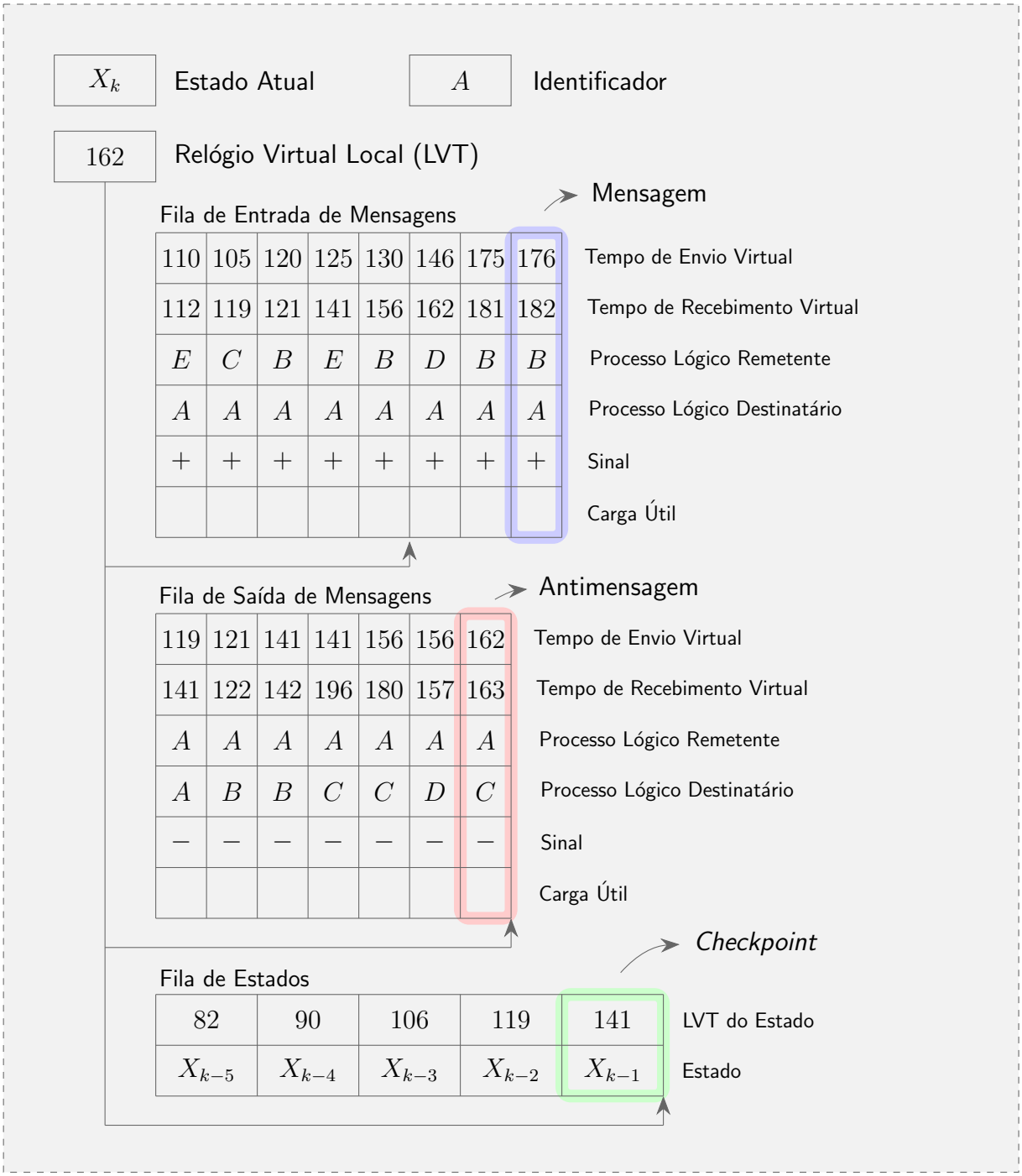
Adicionalmente, devemos compreender o corpo das mensagens que são trocadas pelos processos lógicos. Nesse sentido, apresenta-se a Definição (2.2.4) conforme descrita por (JEFFERSON; SOWIZRAL, 1982).

[2] Na literatura, também é referido como *coordenada espacial virtual*.
[3] Na literatura, também é referido como *coordenada temporal virtual*
[4] Existem implementações do *Time Warp* que não possuem esta estrutura de dado como parte do processo lógico e, para tornar o *rollback* viável é utilizado um mecanismo denominado *reverse computation*.

Definição 2.2.4 (Corpo da Mensagem)**:** *O corpo de toda mensagem é composto pelo: identificador do remetente, identificador do destinatário, tempo virtual de envio, tempo virtual de recebimento, sinal e carga útil.*

A seguir, temos a representação de um processo lógico no protocolo *Time Warp*.

Figura 2.2.3 – Representação de um Processo Lógico no Protocolo *Time Warp*.



Fonte: (JEFFERSON, 1985) [Adaptado].

Isto posto, cada processo lógico, repetidamente, executa o evento E_{min} com a menor

marca de tempo na fila de entrada de mensagens sem qualquer garantia de restrição de causalidade. Esta execução prossegue enquanto não houver um evento E cuja marca de tempo é menor que a do último evento processado, sendo tal situação mostrada na Figura (2.2.1).

Desse modo, quando um evento E com marca de tempo menor que o último evento processado chega ao processo lógico, é necessário que um mecanismo de *rollback* seja acionado para que este processo lógico desfaça todos os efeitos colaterais causados pela execução prematura dos eventos (JEFFERSON; SOWIZRAL, 1982; SPOLON, 2001).

O mecanismo de *rollback* está dividido em três etapas, sendo elas: Restauração, Cancelamento e *Coast Forwarding*. Na primeira etapa, realiza-se a restauração do estado do processo lógico, isto é, restaura-se o conteúdos das variáveis, incluindo-se as sementes dos possíveis geradores de números aleatórios utilizados. Ainda, tais valores restaurados são para aqueles que foram salvos em um instante *estritamente* anterior à marca de tempo do evento retardatário (JEFFERSON; SOWIZRAL, 1982).

Isto posto, é por este motivo que é necessário o salvamento dos estados dos processos lógicos e, assim, foram propostas diferentes técnicas de salvamento de estados que diferenciam-se, principalmente, no custo de tempo para o salvamento e no consumo de memória durante a simulação (RONNGREN et al., 1996; CAROTHERS; BAUER; PEARCE, 2000). A seguir, cita-se algumas técnicas.

- **Copy State Saving (CSS)**: O estado do processo lógico é salvo toda vez que um evento é processado. Nesta técnica o custo de salvamento e de memória são extremos.
- **Sparse State Saving (SSS/PSS)**: O estado do processo lógico é salvo periodicamente, em intervalos fixos durante toda simulação, ou intervalos dinâmicos que se adaptam ao comportamento da simulação. Nesta técnica, o custo de salvamento e de memória são amenizados, porém ainda há a necessidade do salvamento completo do estado.
- **Incremental State Saving (ISS)**: Nesta técnica, somente as variáveis de estado modificadas são salvas. Nesta técnica o custo de salvamento e de memória são melhores que os últimos citados.
- **Reverse Computation (RC)**: Nesta técnica, não há qualquer salvamento de estado, de tal forma que, o estado é obtido a partir do *processamento reverso* dos eventos processados prematuramente. Nesta técnica há um custo computacional maior, porém não há custo de memória.

Na segunda etapa, o mecanismo de *rollback* verifica se o processo lógico $\mathcal{LP}_i$ enviou alguma mensagem para outro processo lógico $\mathcal{LP}_j$ durante o processamento de um evento prematuro. Neste caso, é necessário o envio de antimensagens indicando que as mensagens

enviadas anteriormente não devem ser processadas (FUJIMOTO, 1990). Contudo, existem três casos a serem tratados pelo processo lógico receptor da antimensagem.

1. **A mensagem não foi processada**: Neste caso, quando o processo lógico $\mathcal{LP}_j$ receber a antimensagem, basta que a mensagem seja removida da fila de entrada de mensagens.
2. **A mensagem foi processada**: Neste caso, quando o processo lógico $\mathcal{LP}_j$ receber a antimensagem, este acionará seu mecanismo de *rollback* para desfazer os efeitos colaterais obtidas pela execução prematura do evento correspondente à antimensagem. Neste caso, denominados este tipo de *rollback* de **secundário**.
3. **A antimensagem chegou primeiro que a mensagem correspondente**: Neste caso, o processo lógico $\mathcal{LP}_j$ simplesmente armazenará na fila de entrada de mensagens e, esperará a mensagem correspondente chegar, ao chegar ambas mensagens serão aniquiladas da fila de entrada.

Existem diferentes técnicas para aplicar a etapa de cancelamento, as quais diferenciam-se pelo desempenho. Neste caso, citamos as seguintes técnicas (RAJAN; RADHAKRISHNAN; WILSEY, 1999; CHETLUR; WILSEY, 2001).

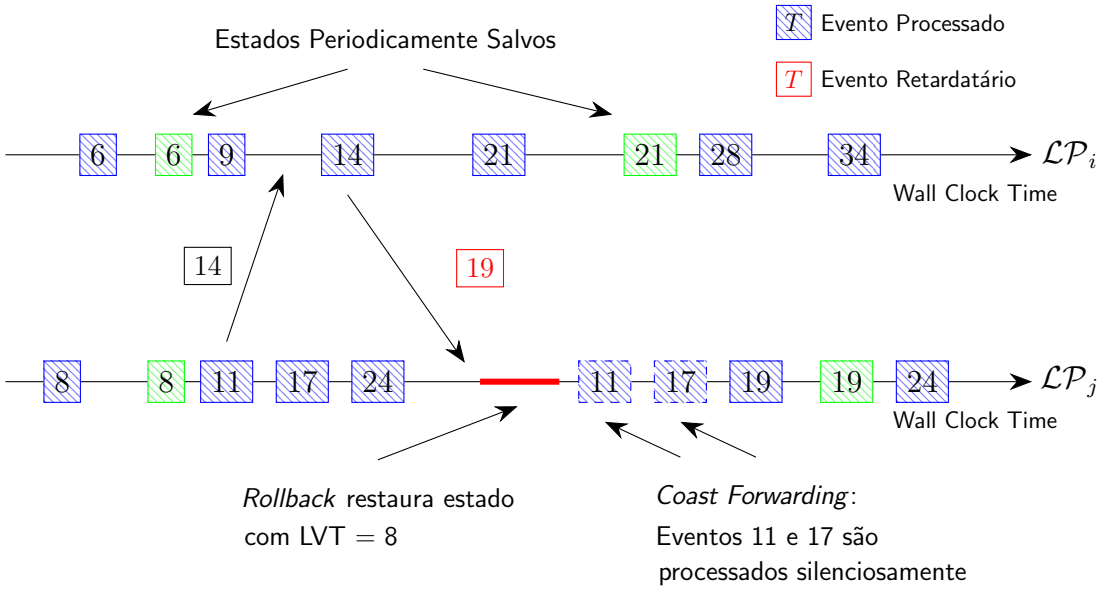
- **Cancelamento Agressivo**: Nesta técnica, quando um processo lógico efetuar um *rollback* para o instante de tempo T, as antimensagens são enviadas **imediatamente**, para qualquer mensagem anteriormente enviada com uma marca de tempo maior do que T. Por padrão, o *Time Warp* utiliza esta técnica.
- **Cancelamento Preguiçoso**: Nesta técnica, o envio de antimensagens é postergado até que o processamento dos eventos posterior ao evento retardatário demonstre (por comparação) que as mensagens foram de fato enviadas incorretamente.

Na terceira etapa, temos que por causa que algumas técnicas de salvamento de estados nem sempre salvam o estado exatamente anterior ao instante de tempo do evento retardatário, é necessário que um mecanismo de processamento, denominado *Coast Forwarding*, processe os eventos até a marca de tempo anterior ao evento retardatário para corrigir esta diferença do estado salvo com relação ao estado necessário para prosseguir a execução.

Contudo, deve-se notar que nesta etapa não há envio de mensagens pelo processo lógico realizando o *rollback*, pois tais eventos sendo processados não foram processados de maneira prematura, porém eles são processados somente para a obtenção do estado necessário (JEFFERSON; SOWIZRAL, 1982). Esta etapa, na literatura também é denominada de **processamento silencioso**.

A seguir, na Figura (2.2.4) um exemplo do mecanismo de *rollback* é representado.

Figura 2.2.4 – Exemplo do Mecanismo de *Rollback* com *Sparse State Saving*.



Fonte: (PRINCIPE et al., 2017) [Adaptado].

Neste caso, temos que o período de salvamento de estados é 3 eventos. Ainda, note que o processo lógico $\mathcal{LP}_j$ recebe um evento retardatário com marca de tempo 19 no meio de seu período de *checkpointing*. Portanto, é necessário que este retorne para o último estado salvo, que possui $LVT_j = 8$ e, assim, é necessário que os eventos com marca de tempo 11 e 17 sejam processados novamente, porém vemos que nesta fase, denominada *Coast Fowarding*, não foi enviado o evento 14 como anteriormente, pois nesta fase nenhuma mensagem é enviada.

Logo, após o processamento dos eventos 11 e 17, o estado anterior ao evento retardatário é obtida e, portanto, podemos prosseguir o processamento sem a quebra de restrição de causalidade.

Mecanismo de Controle Global

O mecanismo de controle local não é capaz de resolver questões relacionadas ao gerenciamento global da simulação como, por exemplo, como é assegurado a progressão global da simulação e manipulação de operações de E/S. Portanto, é necessário que um mecanismo de controle global fique encarregado destas questões (JEFFERSON, 1985).

O principal conceito do mecanismo de controle global é o tempo virtual global (GVT, do inglês *Global Virtual Time*). Esta valor é uma propriedade global da simulação que tem como principal fator indicar sua progressão e, de tal forma, que a podemos definir a seguir.

Definição 2.2.5 (Tempo Virtual Global (GVT))**:** *O tempo virtual global (GVT) em um instante de tempo real r é o mínimo entre todos os tempos virtuais locais no instante de tempo real r e, de todos os tempos virtuais de envio das mensagens enviadas, porém ainda não processadas no instante de tempo real r.*

Desta forma, durante a execução do protocolo otimista *Time Warp*, este deve calcular o GVT periodicamente. A frequência com que este valor é calculado é um *trade-off* entre tempo de resposta mais rápido e consumo de memória total da simulação (JEFFERSON, 1985).

A seguir, é citado algumas das principais funções do mecanismo de controle global.

- **Gerenciamento de Memória**: A memória utilizada por, estados antigos na lista de estados, antimensagens armazenadas e mensagens processadas que possuem marca de tempo anterior ao GVT podem ser descartadas, pois é impossível ocorrer *rollback* para um tempo anterior ao GVT. Portanto, toda esta memória utilizada pode ser recuperada e, denominamos tal etapa de *fossil collection*.
- **Detecção do Término da Simulação**: Quando um processo lógico não tem mais mensagens para serem processadas seu relógio virtual local é definido como $+\infty$. Logo, quando o GVT for definido como $+\infty$, todos os relógios virtuais locais devem ser $+\infty$ e nenhuma mensagem deve estar em trânsito e, portanto, o término da simulação é atingido.

Além disso, outras funções do mecanismo de controle global como, por exemplo, *Gerenciamento de Erros*, *Operações de E/S* e *Recuperação de Falhas* podem ser encontradas em (JEFFERSON, 1985).

2.3 Simuladores de Sistemas Distribuídos

Nesta seção apresenta-se um levantamento dos principais simuladores de sistemas distribuídos encontrados na literatura, realizando uma análise comparativa de suas funcionalidades, características e limitações. Além disso, também é analisado se os simuladores abordados usam algum tipo de mecanismo de simulação paralela ou distribuída.

No entanto, deve-se notar que existe uma quantidade relativamente aceitável de simuladores de sistemas distribuídos, porém na literatura existem poucos simuladores de sistemas de computação de alta performance *exascale* de propósito geral, de tal maneira que, em sua grande maioria tais simuladores limitam-se apenas a simular a topologia de rede.

A seguir, é descrito algumas ferramentas de simulação de sistemas distribuídos.

2.3.1 SimGrid

Este simulador teve início em 1998, por Henri Casanova, na Universidade da Califórnia em *San Diego* e, sua motivação surge da necessidade de usar métodos mais práticos e menos

custosos como, por exemplo, a simulação em vez da utilização de sistemas reais para o estudo de algoritmos de escalonamento em plataformas heterogêneas (CASANOVA et al., 2014).

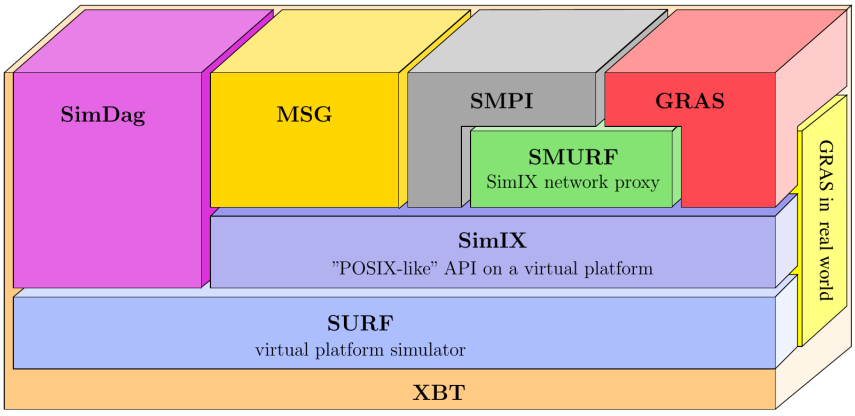
Atualmente, tal ferramenta provê ferramentas para pesquisas na área de computação paralela, sistemas distribuídos em larga escala, sistemas de alto desempenho, sistemas *peer-to-peer* e computação em nuvem.

Além disso, o SimGrid está na versão v3.34[5], sendo seu motor de simulação implementado utilizando simulação de eventos discretos e solucionadores lineares (por exemplo, *Linear Max-Min Solver*). Ainda, este é construído em C/C++ e está disponível para as plataformas Windows, Linux e macOS (CASANOVA; LEGRAND; QUINSON, 2008).

Este simulador é dividido em diversos componentes fornecendo, portanto, quatro interfaces ao usuário. A seguir, é listado tais interfaces (CASANOVA; LEGRAND; QUINSON, 2008).

- **SimDag**: Este componente foi projetado para a análise de heurísticas de escalonamento para aplicações por meio de modelos de grafos direcionados acíclicos (DAG, do inglês *Direct Acyclic Graph*).
- **MSG**: Este componente foi projetado para o estudo de processos concorrentes (CSP, do inglês *Concurrent Sequential Process*).
- **GRAS**: Este componente foi projetado para facilitar o desenvolvimento de aplicações, permitindo que um único código possa ser executado tanto em ambiente de simulação quanto em um ambiente real.
- **SMPI**: Este componente foi projetado para permitir a simulação direta de aplicações MPI.

Figura 2.3.1 – Visão geral dos componentes do SimGrid



Fonte: (CASANOVA; LEGRAND; QUINSON, 2008)

[5] Esta versão é segundo o seu último *release*. Fonte: https://github.com/simgrid /simgrid/tags.

2.3.1.1 Limitações

Embora na literatura considera-se o SimGrid um simulador paralelo, o mesmo afirma que seu motor de simulação é totalmente sequencial, sendo apenas o código de usuário, possivelmente, paralelizável. Logo, pode-se afirmar que o SimGrid não utiliza qualquer protocolo de sincronização para simulação distribuída uma vez que seu motor é sequencial.

Desta forma, é de notável importância que isso afetaria sua escalabilidade, de tal forma que, eles afirmam que seu modelo de simulação suporta no máximo aproximadamente 10.000 nós, segundo (CASANOVA; LEGRAND; QUINSON, 2008). Isto posto, este simulador possui um desafio para a simulação de sistemas de alta performance *exascale* em tempo hábil.

Por fim, é importante notar que houve uma tentativa de paralelização utilizando um protocolo de sincronização, obtendo-se um desempenho muito inferior ao esperado. Desse modo, os próprios autores citaram que em sua opinião, o fracasso relativo da simulação distribuída neste contexto vem do fato de que a simulação de sistemas distribuídos é diferente das simulações classicamente paralelizadas na literatura de PDES (QUINSON; ROSA; THIéRY, 2012).

2.3.2 GridSim

Este simulador foi construído para investigar o gerenciamento de recursos e algoritmos de escalonamento para sistemas de computação de larga escala, de tal forma que, suprisse as necessidades de pesquisadores e estudantes por simuladores que pudessem avaliar diferentes estratégias (BUYYA; MURSHED, 2002).

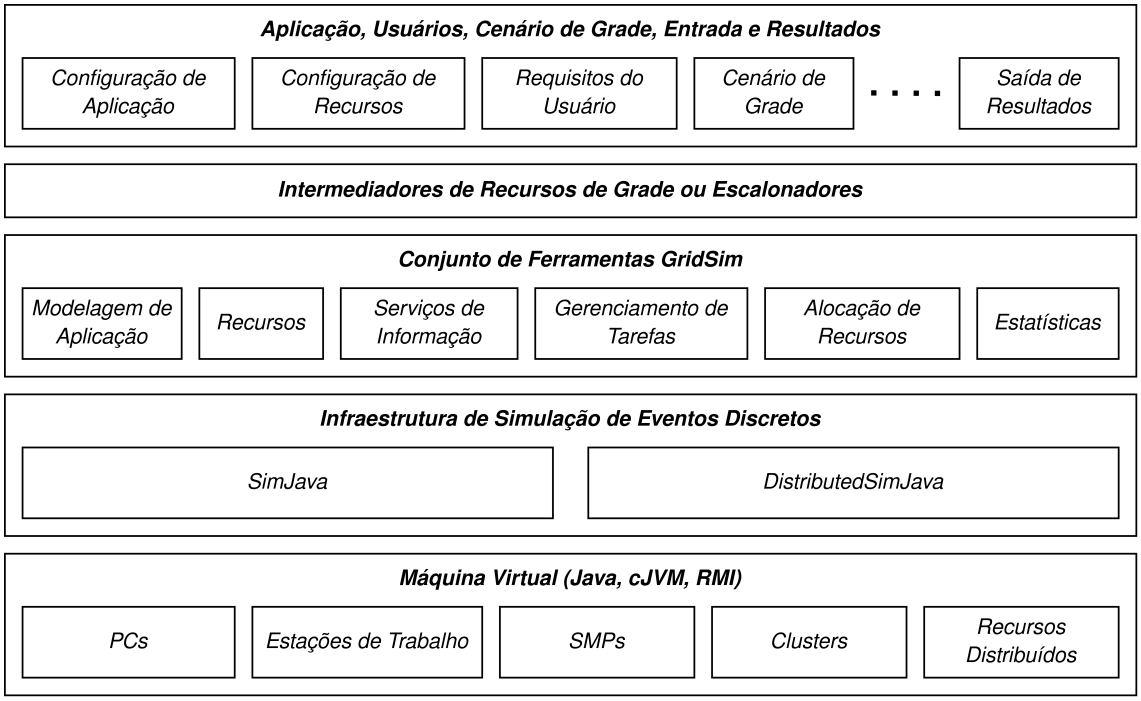
Ainda, o GridSim fornece recursos abrangentes para a simulação de diferentes classes de recursos heterogêneos, usuários, aplicativos, intermediários de recursos (*brokers*) e escalonadores. Ainda, este pode ser usado para simular escalonadores para sistemas de computação distribuídos de domínios administrativos únicos ou múltiplos, como *clusters* e grade computacional (BUYYA; MURSHED, 2002).

Além disso, o GridSim permite a alocação de recursos com compartilhamento no tempo como, por exemplo, sistemas multitarefas e, também seu compartilhamento no espaço como, por exemplo, sistemas em lote e, por fim, tal simulador também fornece a possibilidade do usuário de implementar suas próprias heurísticas de escalonamento.

Este, ainda, é construído utilizando-se a linguagem de programação Java, de tal forma que, fornece grande interoperabilidade e portabilidade entre diferentes arquiteturas de sistemas e sistemas operacionais podendo abranger, portanto, um maior número de usuários.

A seguir, na Figura (2.3.2) é apresentado a arquitetura modular da plataforma de simulação GridSim e os componentes de cada modulo.

Figura 2.3.2 – Arquitetura modular da plataforma de simulação GridSim.



Fonte: (BUYYA; MURSHED, 2002) [Adaptado].

2.3.2.1 Limitações

Embora, cada componente no GridSim é executado por um *thread* exclusiva para cada componente modelado, de modo que, por meio de utilização de *sockets* é possível a comunicação entre componentes permitindo, portanto, sua execução paralela, temos que existe um gargalo na escalabilidade do sistema por causa do custo de memória associado para o gerenciamento de cada *thread* (BUYYA; MURSHED, 2002).

Desta forma, mesmo que um computador paralelo possa ser usado para executar uma simulação distribuída, o custo de memória por nó de processamento será altíssimo, de tal forma que, a escalabilidade será limitada globalmente necessitando, portanto, mais recursos físicos computacionais.

Em suma, é importante notar que embora haja a possibilidade da simulação distribuída e que este simulador é implementado utilizando conceitos de simulação de eventos discretos, não foi utilizado qualquer protocolo de sincronização.

2.3.3 CODES

CODES (do inglês, **CO**-**D***esign of multilayer* **E***xascale* **S***torage* and data-intensive systems) é um *framework* construído sobre o simulador de propósito geral ROSS (do inglês, *Rensselaer's Optimistic Simulation System*) e, tem por finalidade de simular cargas de trabalho

de E/S e topologias de rede de sistemas de computação de alta performance (CAROTHERS, 2015).

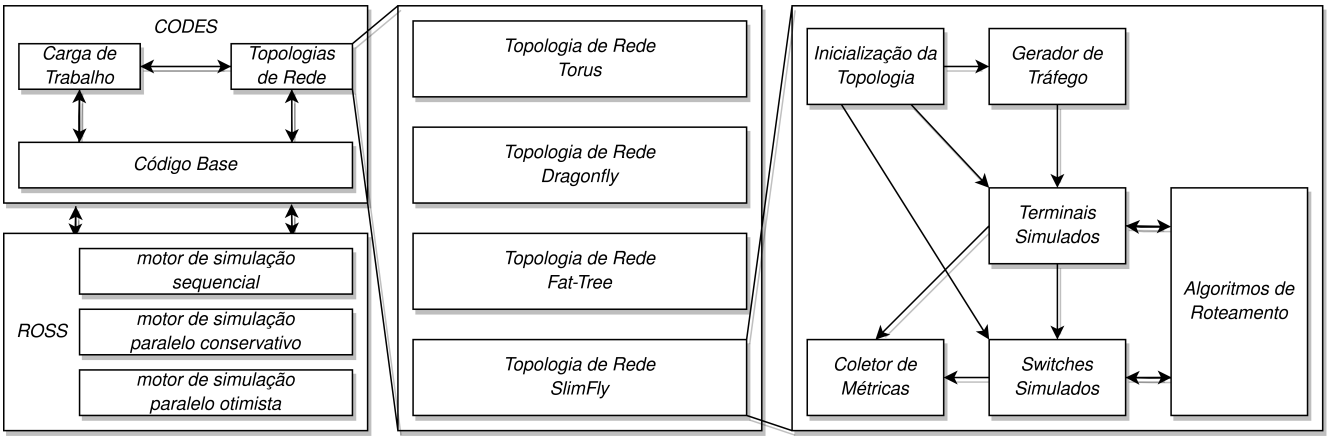
Este simulador fornece uma maneira conveniente e eficiente de usar bancos de traços de cargas de trabalho de rede e usá-las para simular a comunicação em uma topologia de rede específica. Além disso, CODES fornece uma variedade de topologias de rede para simular tais traços, entre elas: *dragonfly* (MUBARAK et al., 2012), *torus* (MUBARAK et al., 2014), *fat-tree* (JAIN et al., 2017) e *LogGP* (ALEXANDROV et al., 1995).

Além disso, como tal simulador é construído sobre o ROSS, a simulação distribuída é obtida a partir da utilização de protocolos de sincronização, tendo disponíveis os protocolos *YAWNS* e *Time Warp*, sendo o primeiro um protocolo conservativo e o último um protocolo otimista.

Desta forma, tal simulador é altamente escalável, sendo demonstrado, por exemplo, em simulações de uma topologia *torus* realizadas no supercomputador IBM Blue Gene/L obtiveram uma taxa de eventos de aproximadamente 12,36 bilhões de eventos por segundo sendo processados em 131.072 núcleos de processamento, por comparação, motores sequenciais, dificilmente, ultrapassam a marca de 2 milhões de eventos por segundo.

A seguir, é apresentado a arquitetura da plataforma de simulação CODES e seus componentes.

Figura 2.3.3 – Arquitetura da plataforma CODES.



Fonte: (LIU et al., 2015) [Adaptado].

2.3.3.1 Limitações

Embora tal simulador não possua uma limitação em sua escalabilidade, desde que adota protocolos de sincronização para simulações distribuídas, entre eles o *Time Warp*, temos que sua limitação está em sua gama de funcionalidades.

Isto posto, algumas das funcionalidades não fornecidas pelo *framework* CODES é a

possibilidade de configuração de políticas de escalonamento destas cargas de trabalho entre os diferentes nós de processamento e, a possibilidade de construir um modelo totalmente customizável de maneira conveniente, pois, atualmente, as topologias de rede são implementadas em código C/C++ como aplicações totalmente diferentes sendo, portanto, muito específicas.

Logo, por um lado tal especificidade é o que garante que o simulador possua tal altíssimo desempenho, pois extrai-se peculiaridades de cada topologia de rede, porém oferece certa inflexibilidade para modelos mais customizáveis e gerais.

2.4 Comparação dos Simuladores de Sistemas Distribuídos

Nesta seção, será realizado o comparativo entre as diferentes ferramentas para simulação de sistemas distribuídos. Para isso, foram escolhidos parâmetros para que tal análise fosse objetiva.

Desta forma, os principais parâmetros escolhidos foram.

- **Framework para o desenvolvimento do simulador**: O *framework* reflete, principalmente, nas questões de desempenho, usabilidade, portabilidade e interoperabilidade do simulador que será construído sobre esse.
- **Linguagem de programação**: A linguagem de programação reflete, principalmente, nas questões de desempenho, portabilidade e grau de dificuldade para o uso da ferramenta.
- **Licença de uso**: A licença de uso reflete, principalmente, na acessibilidade da ferramenta, no entendimento de seu funcionamento interno e na possibilidade de modificações.
- **Interface gráfica**: A interface gráfica reflete, principalmente, na facilidade que o usuário terá de utilizar a ferramenta, tornando-a mais acessível, de tal forma que, fornece ao usuário um meio para que este não tenha que codificar seu modelo.
- **Documentação**: A documentação reflete, principalmente, na facilidade com que um novato terá para aprender a usar o simulador.
- **Motor de simulação**: O motor de simulação reflete, principalmente, no desempenho que a simulação terá, em geral, motores de simulação paralelos e/ou distribuídos são mais eficientes com relação aos motores de simulação sequenciais, porém são extremamente mais complexos.
- **Escalabilidade**: A escalabilidade reflete, principalmente, o tamanho máximo do modelo a ser simulado.

A seguir, é disponibilizado uma tabela comparativa dos simuladores distribuídos abordados neste capítulo.

Tabela 1 – Comparação dos simuladores de sistemas distribuídos apresentados.

	SimGrid	GridSim	CODES
Framework	Nenhum	SimJava	ROSS
Linguagem	C/C++	Java	C/C++
Licença de Uso	*Open-Source*	*Open-Source*	*Open-Source*
Interface Gráfica	Não	Não	Não
Documentação	Muito Boa	Muito Boa	Boa
Motor	Sequencial	Paralelo	Distribuído
Escalabilidade	10.000	100	50 milhões

Fonte: Elaborado pelo autor.

Desta forma, efetuando uma análise concisa, temos que com relação ao uso de um *framework*, os simuladores GridSim e CODES possuem a necessidade do usuário ter conhecimento prévio sobre seus *frameworks* para utilizá-los, de tal forma que, será necessário um tempo maior de aprendizagem.

Ainda, com relação à linguagem de programação utilizada, o simulador GridSim possui uma maior portabilidade e interoperabilidade entre diferentes arquiteturas e sistemas operacionais, uma vez que tal simulador é construído utilizando a linguagem de programação Java, uma vez que esta possui uma máquina virtual que fornece a transparência de tais especificidades.

Além disso, com relação à interface gráfica, todos os simuladores citados não possuem qualquer interface gráfica, de modo que, é necessário que o usuário tenha conhecimento não só da linguagem de programação, mas também do *framework*, se houver, para a realização da simulação de seus modelos.

Por fim, com relação ao motor de simulação, temos que o SimGrid possui uma escalabilidade limitada, desde que simular sistemas computacionais *exascale* demanda uma imensa quantidade de poder computacional e de armazenamento primário. Por outro lado, o simulador CODES, construído sobre ROSS, possui protocolos de sincronização que permitem que sua simulação seja distribuída, de modo que, diferentes nós de processamento atuam diretamente para o avanço da simulação, de tal forma que, fornece uma maior escalabilidade.

2.5 Considerações Finais

Neste capítulo, apresentou-se os fundamentos essenciais para o entendimento da diferença entre uma simulação de eventos discreto sequencial e paralelo, o entendimento dos desafios da simulação distribuída como, por exemplo, a quebra de restrição de causalidade,

o entendimento do porquê é necessário os protocolos de sincronização, suas características e suas principais diferenças e, por fim, o entendimento profundo do protocolo de sincronização *Time Warp* que será o protocolo utilizado para a paralelização do motor de simulação neste trabalho.

A seguir, no próximo capítulo, este é dedicado à metodologia utilizada para realizar a implementação do protocolo otimista *Time Warp* no motor de simulação do iSPD (*iconic Simulator of Parallel and Distributed Systems*) que é o simulador utilizado neste trabalho, buscando, portanto, permitir a simulação de modelos *exascale* contendo centenas de milhares até dezenas de milhões de nós de processamento.

3 Metodologia

Nesta capítulo, serão abordadas cada etapa da metodologia proposta para a paralelização do motor de simulação. Assim, é importante relembrar as etapas seguidas neste trabalho.

1. Revisão bibliográfica sobre simuladores de eventos discretos paralelo;
2. Adequação do modelo de simulação antes sequencial para um modelo paralelo;
3. Otimização do modelo de simulação paralelo;
4. Execução de testes de desempenho e comparação dos diferentes modos de execução.

Isto posto, podemos afirmar que a primeira etapa de nossa metodologia foi concluída a partir da dissecação dos conceitos fundamentais de simulação distribuída e trabalhos correlatos apresentados no Capítulo 2.

Desta forma, a Seção 3.1 apresentará a arquitetura atual do iSPD e a sua nova arquitetura, a Seção 3.2 apresentará o *framework* base utilizado, ROSS, para a paralelização do motor e a Seção 3.3 apresentará detalhes de implementação. Por fim, na Seção 3.4 será apresentado as considerações finais.

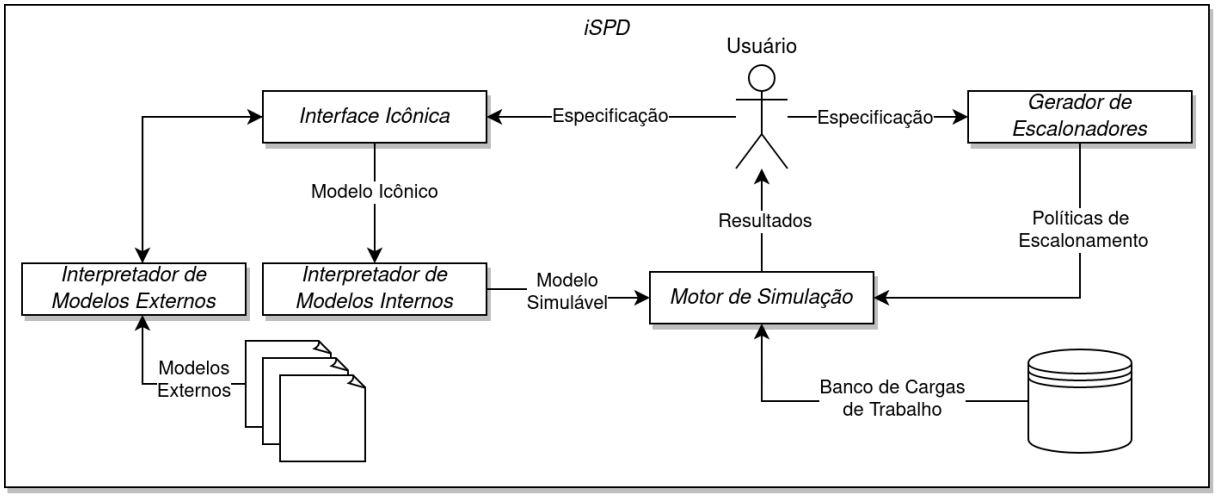
3.1 iSPD: *iconic Simulator of Parallel and Distributed Systems*

Nesta seção, apresenta-se uma visão geral do simulador iSPD (*iconic Simulator of Parallel and Distributed Systems*) (MANACERO et al., 2012), que é a ferramenta alvo deste trabalho. Desta forma, dado que o objetivo é paralelizar o motor de simulação, primeiramente, devemos entender os pontos chaves deste simulador e, assim, propor as modificações necessárias.

O simulador iSPD tem como principal finalidade oferecer uma maior usabilidade tornando o processo de modelagem e de simulação mais simples, eficiente e intuitivo e, para isso, fornece um conjunto de interfaces que auxiliam o usuário neste processo como, por exemplo, interfaces para modelagem, janelas de configuração de cargas de trabalho e geração de políticas de escalonamento (SILVA, 2015).

Desta forma, o iSPD é composto por um conjunto de módulos que desempenham funções bem definidas e, assim, a seguir na Figura (3.1.1) é apresentado o relacionamento de tais módulos dentro do simulador.

Figura 3.1.1 – Arquitetura do simulador iSPD.



Fonte: (SILVA, 2015) [Adaptado].

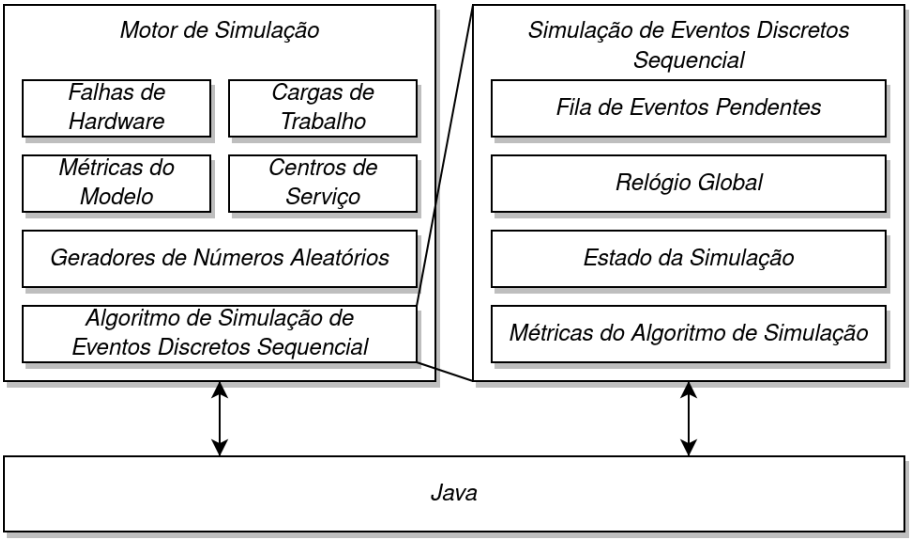
A seguir, é apresentado uma breve descrição de cada módulo.

- **Interface Icônica**: Este módulo é responsável por fornecer ao usuário interfaces que permitam a modelagem do sistema por meio de ícones e, posteriormente, sua configuração por meio de janelas.
- **Interpretador de Modelos Internos**: Este módulo é responsável por interpretar o modelo icônico especificado na etapa de modelagem, convertendo-o para um modelo simulável capaz de ser reconhecido pelo motor de simulação.
- **Interpretador de Modelos Externos**: Este módulo é responsável por interpretar modelos icônicos provenientes de outros simuladores de sistemas distribuídos, convertendo-os para o modelo simulável capaz de ser reconhecido pelo nosso motor de simulação.
- **Motor de Simulação**: Este módulo é responsável por realizar a simulação utilizando conceitos de eventos discretos e fornecer ao usuário as métricas obtidas a partir da simulação.
- **Gerador de Escalonadores**: Este módulo é responsável por gerenciar os escalonadores disponíveis para serem usados na simulação permitindo, também, a geração de novos escalonadores.

Portanto, como o foco deste trabalho é no módulo do motor de simulação, realizaremos uma dissecação neste, para entender, brevemente, seu funcionamento interno e propor as devidas alterações necessárias para a paralelização do motor de simulação.

A seguir, na Figura (3.1.2) é apresentado a arquitetura do motor de simulação anterior a este trabalho.

Figura 3.1.2 – Arquitetura do motor de simulação iSPD.



Fonte: Elaborado pelo autor.

Desta forma, a proposta para a paralelização do motor foi a sua reestruturação completa, pois tal motor de simulação desde sua concepção foi construída pensando-se sempre sobre o modelo de uma memória única e compartilhada, nesse caso, a memória principal do computador executando a simulação, de modo que, todos os módulos e seus algoritmos até então construídos são fundamentalmente sequenciais.

Isto posto, como a paralelização consiste no uso de diferentes nós de processamento de um sistema de computação interconectados em uma rede local (LAN, do inglês *Local Area Network*), temos que tal sistema de computação caracteriza-se por um sistema de memória distribuída em que os processos comunicam-se por meio da troca de mensagens (FLYNN, 1966), inviabilizando todos os algoritmos até então implementados. Assim, fica claro o motivo da proposta de reestruturação.

Com isso, uma vez que este trabalho foca no ganho de desempenho, optou-se por mudar a linguagem de programação de Java para C++. A seguir, é citado alguns motivos para tal escolha.

1. **Modelo de Execução**: Java é uma linguagem híbrida, isto é, o código fonte em Java é primeiro compilado em um código intermediário, denominado *bytecode*, e então é executado pela Máquina Virtual Java (JVM, do inglês *Java Virtual Machine*). Por outro lado, C++ é uma linguagem compilada, isto é, o código fonte é diretamente convertido na arquitetura da máquina alvo, sem a necessidade de uma etapa intermediária de geração de *bytecodes*. Assim, C++ possui uma vantagem no desempenho, desde que não há esse *overhead* adicional de interpretar *bytecodes* em tempo de execução.

2. **Gerenciamento de Memória**: Java é uma linguagem *garbage-collected*, de modo

que, introduz *overhead* na recuperação de memória durante a execução da aplicação. Por outro lado, C++ fornece um gerenciamento manual da memória, de modo que, o programador tem um maior controle e eficiência quando comparado com Java.

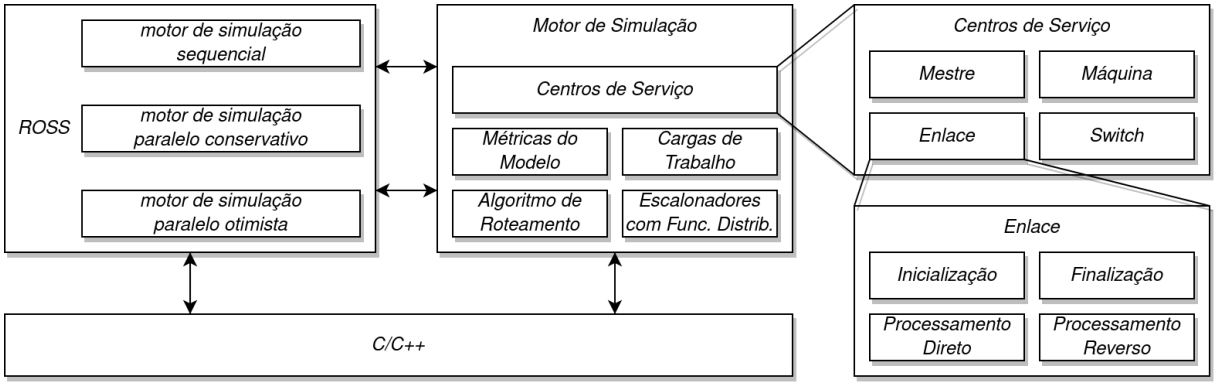
3. **MPI**: Embora, a linguagem de programação Java possua ligações para MPI (do inglês, *Message Passing Interface*), é mais simples e conveniente usá-la em conjunto com linguagens de programação como C ou C++, pois possuem integração nativa, flexibilidade de desenvolvimento, e maior suporte da comunidade quando comparado com ligações para MPI em Java.

Além da mudança da linguagem de programação, também optou-se por usar um simulador de eventos discretos paralelo de propósito geral, chamado ROSS (*Rensselaer's Optimistic Simulation System*), sobre o qual construiremos o nosso modelo de simulação.

Assim, para justificar tal escolha a Seção 3.2 foi dedicada a tal sistema de simulação, pois o entendimento de seu funcionamento interno e de sua importância no estado da arte foram fatores decisivos.

Por fim, por comparação com a arquitetura do motor de simulação apresentado na Figura (3.1.2), a seguir na Figura (3.1.3) é apresentado a nova arquitetura do motor de simulação paralelo proposta neste trabalho.

Figura 3.1.3 – Arquitetura do motor de simulação distribuído iSPD.



Fonte: Elaborado pelo autor.

3.2 ROSS: *Rensselaer's Optimistic Simulation System*

O *Rensselaer's Optimistic Simulation System* (ROSS) destaca-se como uma plataforma avançada e distribuída para simulação de eventos discretos. Sua versatilidade se evidencia ao fornecer suporte abrangente para execução paralela conservativa e otimista. Este sistema, detalhadamente explorado em (CAROTHERS; BAUER; PEARCE, 2000), oferece uma gama de funcionalidades que o tornam uma escolha robusta para cenários diversos de simulação.

Isto posto, enquanto para execução conservativa é utilizado o protocolo YAWNS (NICOL, 1993), para a execução otimista é utilizado o protocolo *Time Warp* (JEFFERSON, 1985), apresentado no Capítulo 2.

Desta forma, com relação ao protocolo otimista, este simulador alcança um desempenho paralelo excelente usando uma técnica denominada **computação reversa**, mostrando ser capaz de atingir a marca de 500 bilhões de eventos por segundo sendo processados em um modelo contendo mais de 250 milhões de processos lógicos em 120 *racks* do supercomputador Sequoia no LLNL (do inglês, *Lawrance Livermore National Laboratory*) (BARNES et al., 2013).

No caso da computação reversa, o mecanismo de *rollback* não é realizado pelas técnicas de salvamento de estados clássicas, porém, é realizado por permitir que os eventos sejam processados em ordem reversa até o instante em que ocorreu a quebra de causalidade. Assim, o consumo de memória da simulação reduz *drasticamente*.

Isto posto, na implementação do nosso motor de simulação iSPD foi necessário escrever tanto o processamento direto, quanto o processamento reverso de cada centro de serviço que são modelados como processos lógicos distintos. Para isso, foi necessário desenvolver um pensamento capaz de enxergar as operações sendo desfeitas e, ainda mantendo a restrição de obter o mesmo estado do processo lógico anterior à quebra de restrição de causalidade.

No entanto, uma propriedade especial da computação reversa é que a maioria das operações realizadas no processamento direto de um evento são *construtivas* por natureza, isto é, não requerem nenhuma história necessária para serem desfeitas. Por outro lado, existem casos em que tais operações não são construtivas como, por exemplo, situações condicionais, para isso, é necessário o uso de uma estrutura de dados fornecidas pelo próprio simulador, denominada *bitfield*, que fornece maneiras de indicar qual ramo condicional foi satisfeito no processamento direto.

A seguir, há um exemplo do processamento direto no Algoritmo (2) e um exemplo do processamento reverso no Algoritmo (3) do escalonador Round-Robin implementado na versão distribuída.

Algoritmo 2 Processamento Direto

```
1: worker_id ← workers[next_worker_index]
2: next_worker_index++
3: se next_worker_index = #workers então
4:     bitfield.c0 ← 1
5:     next_worker_index ← 0
6: fim se
7: retorne worker_id
```

Algoritmo 3 Processamento Reverso

```
1: se bitfield.c0 = 1 então
2:     bitfield.c0 ← 0
3:     next_worker_index ← #workers − 1
4: fim se
```

Neste exemplo, podemos ver que no processamento direto se o próximo índice da máquina operária a ser escolhida é igual ao número total de operários, digamos #workers, então o próximo índice será definido como 0, indicando que voltou-se para o início da fila, de

modo que, mantém-se a propriedade de circularidade da escolha. No entanto, uma vez que tal condição é satisfeita, definimos que o primeiro *bit* da estrutura de dados *bitfield* como 1, com a finalidade de indicar para o processamento reverso que tal condição no processamento direto tinha sido satisfeita.

Assim, mostra-se um exemplo da implementação do escalonador Round-Robin utilizando-se a técnica de computação reversa, sendo esta necessária para a utilização do simulador ROSS executando no modo otimista.

Por fim, ROSS não só permite que executemos paralelamente no modo otimista, mas também permite a execução conservativa, de tal forma que, poderemos comprar o desempenho de ambos os protocolos de otimização em diferentes modelos sendo simulados. Assim, poderemos entender o comportamento do nosso simulador com diferentes protocolos de sincronização.

3.3 Implementação

Nesta seção, apresentaremos os módulos do motor de simulação que passaram por um processo de reimplementação e/ou adaptação, com o propósito de construir um motor de um sistema de simulação distribuída, bem como para simular atributos inerentes a sistemas de alta performance *exascale*.

Portanto, as subseções subsequentes estão reservadas para a apresentação destes módulos, fundamentais para o pleno funcionamento do simulador. Consequentemente, a modelagem da carga de trabalho será abordada na Subseção (3.3.1), enquanto a modelagem da geração de carga de trabalho será detalhada na Subseção (3.3.2), entre outros tópicos correlacionados.

3.3.1 Módulo de Modelagem de Carga de Trabalho

Nesta subseção, apresentaremos como as cargas de trabalho são modeladas em nosso simulador. Além disso, tal subseção foi dedicada, pois a modelagem das cargas de trabalho na versão distribuída de nosso simulador difere ligeiramente daquela fornecida na versão sequencial, tal modificação deve-se ao fato da adição de um atributo inerente para a simulação de arquiteturas de GPUs.

Assim, na versão distribuída, podemos afirmar que o espaço de tarefas $\mathbb{T}$[1] pode ser definido como segue.

[1] Por questões de comparação, na versão sequencial, o espaço de tarefas era definido por $\tilde{\mathbb{T}} = \{(t_1, t_2) \in \mathbb{R}^2 \,|\, t_1 > 0,\, t_2 > 0\}$

Definição 3.3.1 (Espaço de Tarefas)**:** *O espaço de tarefas* $\mathbb{T}$ *é o conjunto de todas as tarefas possíveis de serem representadas, e define-se o espaço de tarefas pelo conjunto a seguir.*

$$\mathbb{T} = \{(t_1, t_2, t_3) \in \mathbb{R}^3 \,|\, t_1 > 0,\, t_2 > 0,\, t_3 \in [0, 1]\}$$

em que, cada componente t_1, t_2 e t_3 da tarefa é descrito a seguir na ordem listada.

- **Tamanho de Processamento (MFLOP)**: Componente t_1 que refere-se ao custo computacional da tarefa nos centros de processamento em MFLOP. Assim, este componente está diretamente relacionado ao tempo necessário para o processamento de uma tarefa em um centro de processamento.
- **Tamanho de Comunicação (Mbits)**: Componente t_2 que refere-se ao custo de comunicação da tarefa nos centros de comunicação em Mbits. Assim, este componente está diretamente relacionado ao tempo necessário para a comunicação de uma tarefa em um centro de comunicação.
- **Offloading Computacional (%)**: Componente t_3 que refere-se à porcentagem do custo computacional que será transferido para ser processado em arquiteturas de unidades de processamento gráfico (GPUs).

Assim, o atributo adicional denominado como **offloading computacional**, cujo propósito é especificar a porção da tarefa a ser direcionada para unidades de processamento gráfico (GPUs), foi incorporado com o intuito de habilitar o usuário a simular o processamento de tarefas em GPUs.

Ademais, a título de ilustração, consideremos uma tarefa com um custo computacional de 50 GFLOP/s e um atributo de *offload computacional* igual a 0,75. Portanto, podemos afirmar que 37,50 GFLOP/s (75%) serão alocados para as unidades de processamento gráfico (GPUs), enquanto 12,50 GFLOP/s (25%) serão destinados às unidades de processamento central (CPUs).

Além disso, é importante ressaltar que a introdução do offloading computacional não apenas oferece ao usuário a capacidade de simular o processamento em GPUs, mas também amplia significativamente a flexibilidade e a precisão da simulação como um todo. Essa funcionalidade permite que os usuários ajustem a alocação de recursos de acordo com as características específicas de suas tarefas e sistemas, otimizando assim o desempenho da simulação em cenários que envolvam arquiteturas heterogêneas de hardware. Além disso, a inclusão desse atributo abre portas para investigações mais detalhadas sobre os impactos do offloading computacional em diversas métricas de desempenho e eficiência, promovendo

um campo de pesquisa em constante evolução no âmbito da simulação de sistemas de alto desempenho.

Finalmente, a operação interna da aplicação deste atributo será explorada na Subseção (3.3.4), na qual trataremos do processamento das tarefas pelos centros de processamento.

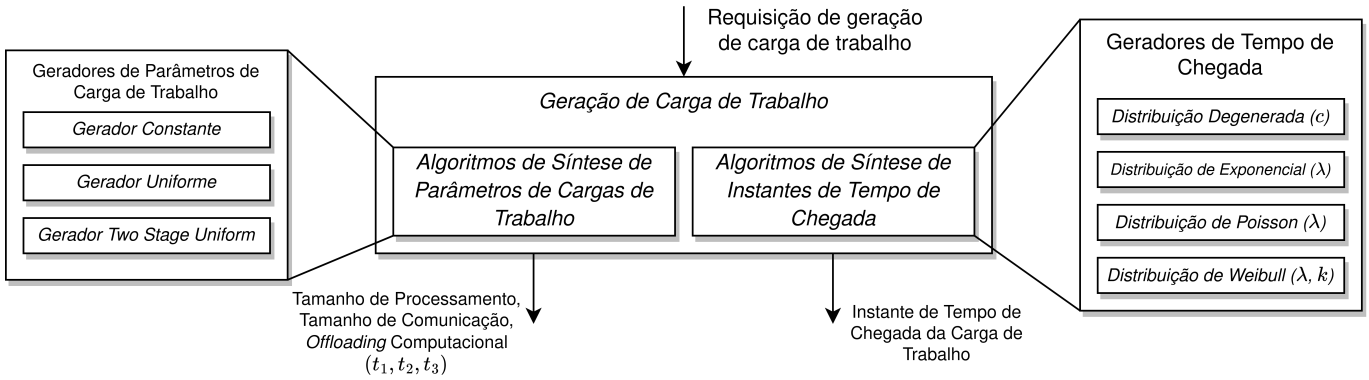
3.3.2 Módulo de Geração de Carga de Trabalho

O módulo responsável pela geração da carga de trabalho compreende uma série de algoritmos destinados à síntese de tarefas, abrangendo não apenas a especificação dos parâmetros de tamanho de processamento, tamanho de comunicação e offloading computacional, mas também a modelagem da distribuição temporal das chegadas dessas tarefas.

O módulo encarregado da geração da carga de trabalho é composto por dois submódulos distintos. O primeiro submódulo tem como finalidade a síntese dos parâmetros das tarefas, conforme definidos em (3.3.1). O segundo submódulo, por sua vez, se dedica à geração dos instantes de chegada das tarefas, nos quais o intervalo de tempo entre essas chegadas é determinado com base em uma distribuição de probabilidades previamente definida.

A seguir, a Figura (3.3.1) exibe o diagrama do módulo de geração de cargas.

Figura 3.3.1 – Diagrama do módulo de geração de cargas de trabalho do iSPD.



Fonte: Elaborado pelo autor.

Assim, com relação ao módulo de síntese dos parâmetros da carga de trabalho, temos os seguintes geradores.

- **Gerador Constante**: Refere-se a um gerador cujas tarefas geradas terão os mesmos parâmetros t_1, t_2 e t_3, previamente definidos pelo usuário.
- **Gerador Uniforme**: Refere-se a um gerador cujos parâmetros t_1, t_2 e t_3 das tarefas geradas seguem distribuições uniformes contínuas independentes, isto é, $t_i \sim U(a_i, b_i)$, $i = 1, 2, 3$, tal que $a_i \leq t_i \leq b_i$.

- **Gerador Uniforme de Dois Estágios**: Refere-se a um gerador no qual os parâmetros t_1, t_2, e t_3 das tarefas geradas seguem distribuições uniformes de dois estágios independentes. Em outras palavras, para cada $t_i \sim TSU(l_i, m_i, h_i, p_i), i = 1, 2, 3$, se um valor $x_i \sim U(0, 1)$ gerado aleatoriamente no momento da geração da tarefa satisfizer $x_i \leq p_i$, então t_i seguirá a distribuição $t_i \sim U(l_i, m_i)$. Caso contrário, t_i seguirá a distribuição $t_i \sim U(m_i, h_i)$.

Por outro lado, com relação ao módulo de síntese dos instantes de chegada das tarefas, temos os seguintes tipos de geradores com as seguintes distribuições.

- **Distribuição Degenerada (c)**: Refere-se a um gerador cujos instantes de chegadas t_i e t_j gerados consecutivamente satisfazem a condição $t_j - t_i = c$, $c > 0$.
- **Distribuição Exponencial (λ)**: Refere-se a um gerador cujos instantes de chegadas t_i e t_j gerados consecutivamente satisfazem a distribuição exponencial, isto é, $(t_j - t_i) \sim Exp(\lambda)$.
- **Distribuição de Poisson (λ)**: Refere-se a um gerador cujos instantes de chegadas t_i e t_j gerados consecutivamente satisfazem a distribuição de Poisson, isto é, $(t_j - t_i) \sim Poisson(\lambda)$.
- **Distribuição de Weibull (λ, k)**: Refere-se a um gerador cujos instantes de chegadas t_i e t_j gerados consecutivamente satisfazem a distribuição de Weibull, isto é, $(t_j - t_i) \sim Weibull(\lambda, k)$.

Na versão anterior, já se encontravam disponíveis tanto geradores de carga, como o gerador uniforme de dois estágios, quanto geradores de instantes de tempo de chegada, como a distribuição exponencial. Contudo, com a finalidade de aprimorar a diversidade de opções, foram adicionados novos geradores, ampliando significativamente a capacidade do usuário em simular uma extensa variedade de comportamentos das cargas de trabalho em sistemas de alto desempenho.

Considerando esse contexto, a incorporação referida tem o objetivo de substancialmente ampliar a versatilidade na elaboração de cenários, resultando em um ambiente de simulação mais expansivo e adaptável às demandas particulares de análises e experimentos e, portanto, enriquecendo assim a capacidade de modelagem e proporcionando maior precisão na representação de diferentes contextos e situações simuladas.

Por fim, no contexto de um motor de simulação distribuído que utiliza um protocolo otimista, apresentamos nos Algoritmo (4) e (5) um exemplo simplificado de código relacionado ao processamento direto e reverso para a geração de carga de trabalho, baseado no gerador uniforme de dois estágios.

Algoritmo 4 Processamento Direto do Gerador Uniforme de Dois Estágios

Entrada: O estado X_k do processo lógico chamador
Saída: A tarefa (t_1, t_2, t_3)

1: **para** $i \leftarrow 1$ **até** 3 **faça**
2: $x_i \leftarrow$ tw_rand_unif(X_{k+i-1}) ▷ $x_i \sim U(0,1)$
3: **se** $x_i \leq p_i$ **então** ▷ $t_i \sim U(l_i, m_i)$
4: $t_i \leftarrow l_i +$ tw_rand_unif(X_{k+i}) $\cdot (m_i - l_i)$
5: **caso contrário** ▷ $t_i \sim U(m_i, h_i)$
6: $t_i \leftarrow m_i +$ tw_rand_unif(X_{k+i}) $\cdot (h_i - m_i)$
7: **fim se**
8: **fim para**
9: remaining_tasks $\leftarrow$ remaining_tasks $- 1$
10: **retorne** (t_1, t_2, t_3)

Dessa forma, é possível observar que, em cada iteração do laço, a função **tw_rand_unif**, responsável por gerar um valor aleatório uniforme no intervalo unitário, é invocada duas vezes. Consequentemente, o laço executará um total de três iterações, resultando em um total de seis chamadas para a função.

Portanto, quando ocorrer um *rollback*, é necessário corrigir o estado do processo lógico em relação à geração de números pseudoaleatórios. Para isso, segue o Algoritmo (5).

Algoritmo 5 Processamento Reverso do Gerador Uniforme de Dois Estágios

Entrada: O estado X_k do processo lógico chamador
Saída: Nenhuma

1: **para** $i \leftarrow 1$ **até** 6 **faça**
2: tw_rand_reverse_unif(X_{k+i-1})
3: **fim para**
4: remaining_tasks $\leftarrow$ remaining_tasks $+ 1$

3.3.3 Módulo de Roteamento

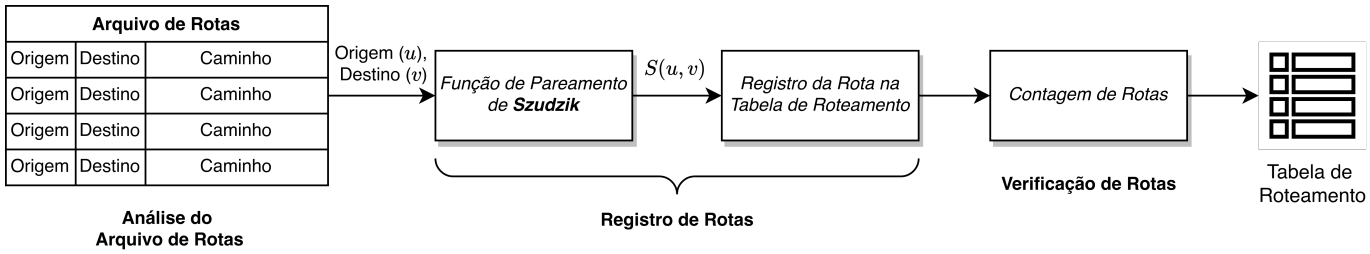
O módulo responsável pelo registro das rotas entre diferentes centros de serviços, a checagem destas rotas e o mecanismo necessário para o roteamento dos clientes entre tais centros de serviços são responsabilidades deste módulo.

Anteriormente, poucas topologias de redes de interconexão eram viáveis devido às limitações na implementação do algoritmo de roteamento e na capacidade de centros de serviço habilitados para encaminhar tarefas intermediárias.

Assim, na versão atual, foi implementado um novo módulo de roteamento que possibilita a simulação de uma ampla variedade de topologias de interconexão. Um dos principais avanços obtidos com esse novo módulo de roteamento é a melhoria da eficiência e a capacidade de suportar várias rotas entre o mesmo par de centros de serviços.

A seguir, a Figura (3.3.2) exibe o diagrama representativo do fluxo do módulo de roteamento.

Figura 3.3.2 – Diagrama do fluxo do módulo de roteamento que permite o registro de múltiplas rotas entre pares de centros de serviço e, portanto, permite a simulação de uma maior quantidade de topologias.



Fonte: Elaborado pelo autor.

Dessa forma, a seguir, será feita uma breve conclusão sobre a funcionalidade de cada seção no fluxo do módulo de roteamento.

1. **Análise do Arquivo de Rotas**: A primeira etapa consiste na análise do arquivo de rotas, isto é, um arquivo contendo linhas formatadas com informações de origem, destino e caminho é recebido e posteriormente analisado. Essas linhas são processadas com o objetivo de extrair essas informações, que serão utilizadas para construir a entrada da próxima etapa neste fluxo de registro de rotas.

2. **Registro de Rotas**: A segunda etapa envolve o recebimento das informações de cada rota e a utilização delas para construir uma tabela de roteamento. Essa tabela é implementada como uma tabela de espalhamento. No entanto, como uma tabela de espalhamento é baseada no conceito de uma única chave, não seria possível usar diretamente as informações de origem e destino para consultar a rota.

   Para resolver essa situação, faz-se uso da função de pareamento de **Szudzik**, que recebe dois números naturais como entrada e os mapeia para um número natural exclusivo. Essa função é definida da seguinte maneira.

$$S(x, y) = \begin{cases} x^2 + x + y & \text{se } x \geq y, \\ y^2 + x & \text{c.c.} \end{cases}$$

   Neste caso, as entradas consistem nos identificadores dos centros de serviço de origem e destino, permitindo assim que a consulta à rota seja realizada em tempo médio $O(1)$.

3. **Verificação de Rotas**: A terceira etapa consiste na verificação da ausência de rotas necessárias, ou seja, se existem rotas que deveriam ter sido registradas, mas não o foram,

bem como na identificação de rotas registradas em excesso. Essa etapa desempenha um papel fundamental, pois permite que o usuário faça as correções necessárias antes do início da simulação.

3.3.4 Módulo de Centro de Serviços

O módulo de centro de serviços é uma parte fundamental do motor de simulação, pois sua função principal é simular os centros de serviços, ou seja, replicar os mecanismos e responsabilidades de cada centro de serviço, entre eles mestres, máquinas, enlaces e switches.

Neste módulo, os centros de serviços são implementados com base na teoria de filas. Isso significa que o modelo sendo simulado representa a interconexão entre esses centros de serviço, o que chamamos de **rede de filas**. Além disso, há a presença dos chamados **clientes**, que correspondem às tarefas e mensagens que percorrem as filas, aguardando para serem atendidos.

Assim, em comparação com a versão anterior do motor de simulação, podemos afirmar que este módulo passou por mudanças significativas. Isso se deve ao fato de que este módulo desempenha um papel crucial nas principais implementações da **computação reversa**, uma vez que os centros de serviço são responsáveis pelo gerenciamento da chegada e envio de eventos.

Além disso, em comparação com a capacidade de simulação de sistemas de alto desempenho, o novo motor de simulação possibilita a simulação de arquiteturas heterogêneas, com foco especial em GPUs. Além disso, houve uma modificação significativa no modelo de simulação do consumo de energia. Anteriormente baseado em um modelo constante, o novo modelo segue uma abordagem linear semelhante à utilizada no SimGrid.

Dessa forma, a seguir, apresentaremos os mecanismos e as responsabilidades de cada centro de serviço, com foco principal nas modificações essenciais realizadas em comparação com a versão anterior do motor. Além disso, faremos uma classificação dos diferentes tipos de centros de serviço.

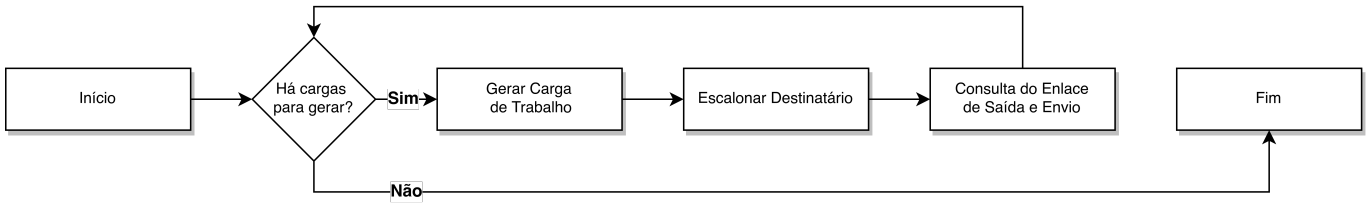
Centros de Processamento

Os centros de serviço incluídos neste grupo são dedicados ao processamento de tarefas, ao cálculo de métricas de processamento e à capacidade de tomar decisões sobre o roteamento de tarefas que chegam a esse centro, mesmo que essas tarefas não sejam originalmente destinadas a ele. Entre esses centros, podemos destacar os seguintes exemplos:

- **Mestre**: O papel do mestre engloba a recepção das tarefas geradas durante a criação da carga de trabalho, o escalonamento conforme a política definida pelo usuário e a recepção dos resultados das tarefas que foram concluídas após o escalonamento.

No entanto, a principal modificação relevante (além do fato da computação reversa) está relacionada ao processo de geração da carga de trabalho cujo diagrama é representado na Figura (3.3.3). Agora, o mestre realiza essa geração durante a execução da simulação. Em comparação com a abordagem anterior, em que toda a carga de trabalho era gerada no início da simulação, essa mudança resultou em uma significativa economia de memória RAM. Anteriormente, essa abordagem gerava um consumo massivo de memória RAM, limitando a quantidade máxima de tarefas que poderiam ser geradas. Agora, com a geração de tarefas ocorrendo dinamicamente durante a execução da simulação, o consumo de memória RAM foi drasticamente reduzido, permitindo a geração de uma quantidade muito maior de tarefas.

Figura 3.3.3 – Diagrama do fluxo de execução da geração de cargas de trabalho pelo mestre. Neste caso, o mestre gerará as cargas à medida que são necessárias, em vez de gerar toda a carga da simulação no início.



Fonte: Elaborado pelo autor.

- **Máquina**: O papel da máquina abrange a recepção das tarefas destinadas ao processamento, o cálculo das métricas pertinentes e o subsequente envio dos resultados de volta ao mestre, caso a máquina seja o destino da tarefa. Alternativamente, a máquina também pode encaminhar a tarefa para o próximo enlace de saída, caso ela sirva como intermediário em uma rota específica.

  No entanto, uma das principais modificações neste centro de serviço é a capacidade de simular arquiteturas heterogêneas, com um enfoque especial em GPUs. Para viabilizar essa funcionalidade, algumas fórmulas utilizadas para o cálculo de métricas foram ajustadas a fim de garantir a obtenção de resultados satisfatórios.

  A seguir, apresentamos a nova fórmula para o cálculo do tempo de processamento de uma tarefa $(t_1, t_2, t_3) \in \mathbb{T}$ em uma máquina m.

$$\Delta T_m(t) = \frac{N_1(1-t_3)t_1}{(1-L_m)CP_m} + \frac{N_2 t_3 t_1}{GPU_m} + \frac{t_2}{IB_m} \tag{3.3.1}$$

  em que CP_m é o poder computacional da CPU, GPU_m é o poder computacional da GPU, IB_m é a largura de banda de interconexão CPU-GPU, L_m e o fator de carga da máquina, N_1 é a quantidade de núcleos de CPU e N_2 é a quantidade de núcleos de GPU.

Por último, outra modificação significativa foi feita no modelo de potência energética de uma máquina m. Anteriormente, a potência era definida como uma constante K, de forma que fatores como a carga de trabalho na máquina não eram considerados. Abaixo, apresentamos a fórmula atualmente utilizada para o modelo de potência energética quando uma dada tarefa esta sendo processada:

$$P_{m,t} = P_{m,\mathsf{idle}} + \frac{1}{N_1}\left(P_{m,\mathsf{max}} - P_{m,\mathsf{idle}}\right) \tag{3.3.2}$$

em que $P_{m,\mathsf{idle}}$ é a potência da máquina sem qualquer carga de trabalho, e $P_{m,\mathsf{max}}$ é a potência da máquina com carga de trabalho máxima.

Centros de Comunicação

Os centros de serviço incluídos neste grupo são dedicados a comunicação dos dados, ao cálculo de métricas relativo à comunicação dos dados e a capacidade de tomar decisões de encaminhamento de tarefas. Entre esses centros, podemos destacar os seguintes exemplos:

- **Enlace**: O papel do enlace é mimetizar a transmissão de dados (por exemplo, tarefas) entre dois centros de serviço, envolvendo o registro e cálculo de atrasos relacionados à latência de transmissão, o tempo de espera na fila de entrada e o tempo necessário para transmissão.

  Dito isso, algumas das principais alterações incluíram a introdução da computação reversa e a otimização na quantidade global de eventos necessários para replicar o processo de transmissão de dados.

- **Switch**: O papel do switch é mimetizar o repasse de dados entre os diferentes centros de serviço que estão conectados ao switch, envolvendo o registro e cálculo de atrasos relacionados à latência de transmissão e o tempo de espera na fila.

  Além disso, a alteração central nos switches envolveu a capacidade de configuração manual por parte dos usuários, além da já mencionada implementação da computabilidade reversa.

3.3.5 Módulo de Coleção de Métricas

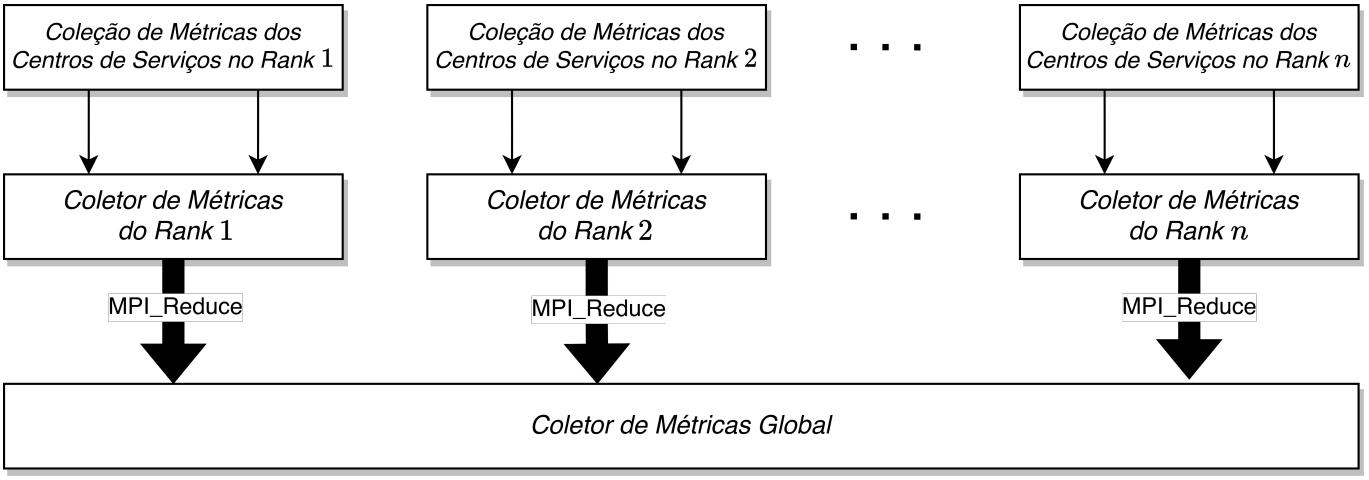
Este módulo corresponde a uma das principais funcionalidades do simulador, que consiste em coletar e prover as métricas obtidas a partir da simulação do modelo especificado. Entretanto, este módulo sofreu intensas modificações, pois as métricas dos centros de serviço e métricas relacionadas estão dispersas na memória distribuída.

Nesse contexto, as principais modificações implementadas neste módulo incluíram a introdução de um sistema que permite a coleta distribuída das métricas à medida que a

simulação progride, juntamente com a coordenação eficaz da coleta de métricas entre os núcleos de simulação em operação.

A seguir, a Figura (3.3.4) exibe o diagrama do módulo de coleção de métricas.

Figura 3.3.4 – Diagrama do módulo de coleção de métricas distribuído.



Fonte: Elaborado pelo autor.

Dessa forma, com base na Figura (3.3.4), é possível segmentar a coleta de métricas em três níveis distintos e descrever suas respectivas responsabilidades da seguinte maneira:

- **Nível de Centro de Serviço**: Nesse nível, o coletor de métricas específico do centro de serviço assume a responsabilidade pelo cálculo e registro das métricas associadas a esse centro.

  Dessa forma, podemos afirmar que neste nível, cada coletor de métricas possui uma visão exclusiva do centro de serviço ao qual está associado. Em outras palavras, existem coletores de métricas deste nível dedicados a cada centro de serviço individualmente.

- **Nível de Rank**: Nesse nível, o coletor de métricas é específico de um *rank*, assim, possui visão de todos os centros de serviço que estão sendo simuladores neste *rank*.

  Assim, podemos afirmar que, neste nível, cada coletor de métricas mantém uma visão exclusiva do conjunto de centros de serviço simulados por seu *rank* específico. Isso implica que já possuímos uma visão parcial das métricas globais a partir dessa perspectiva.

- **Nível Global**: Nesse nível, o coletor de métricas possui a visão geral sobre as métricas obtidas durante a simulação. Assim, neste nível o coletor de métricas é responsável por calcular as métricas globais e gerar o relatório de métricas para o usuário.

Portanto, com o objetivo de ilustrar e demonstrar uma das métricas principais do simulador, apresentaremos os procedimentos necessários para calcular o Tempo de Simulação

Global (GST, do inglês *Global Simulation Time*) em um motor de simulação distribuído. Esta métrica indica o tempo requerido para a execução da carga de trabalho de um determinado modelo.

Para isso, primeiramente definimos o conceito de último tempo de atividade (LAT, do inglês *Last Activity Time*).

Definição 3.3.2: *(Último Tempo de Atividade) O último tempo de atividade é o instante de tempo da última ação realizada por um centro de serviço. Assim, para um centro de serviço l dizemos que seu último tempo de atividade é LAT_l.*

Desta forma, é responsabilidade do coletor de métricas em nível de centro de serviço calcular o último tempo de atividade do centro de serviço associado, em que, posteriormente, esse valor será utilizado pelo coletor de métricas em nível de *rank*.

Além disso, é importante destacar que o conceito de "último tempo de atividade"difere do "tempo local virtual"do processo lógico. Em primeiro lugar, o tempo local virtual registra apenas o momento de chegada de um evento, não necessariamente a última ação executada pelo centro de serviço. Em segundo lugar, é crucial observar que $LAT_l < +\infty$, enquanto o tempo local virtual é considerado infinito quando o processo lógico não possui mais eventos a serem processados.

Isto posto, também definimos o conceito de tempo de simulação de *rank* (RST, do inglês *Rank Simulation Time*).

Definição 3.3.3: *(Tempo de Simulação de Rank) Seja r um rank, então o tempo de simulação do rank r é o último tempo de atividade dentre os centros de serviço simulados por este rank, ou seja.*

$$RST_r = \max_{l \in \mathcal{LP}(r)} LAT_l$$

em que $\mathcal{LP}(r)$ é o conjunto de todos os processos lógicos pertencentes ao *rank* r.

Por fim, uma vez que o coletor de métricas global toma conhecimento dos tempos parciais de simulação de cada *rank*, este calculará o tempo máximo, o qual corresponderá ao que chamaremos de tempo de simulação global (GST, do inglês *Global Simulation Time*).

Definição 3.3.4: *(Tempo de Simulação Global) O tempo de simulação global é o valor máximo entre todos os tempos de simulação de rank, ou seja.*

$$\begin{aligned} GST &= \max_{1 \leq r \leq n} RST_r \\ &= \max_{1 \leq r \leq n} \left\{ \max_{l \in \mathcal{LP}(r)} LAT_l \right\} \end{aligned}$$

em que n é a quantidade total de *ranks* atuantes na simulação.

3.4 Considerações Finais

Neste capítulo, apresentou-se a metodologia para a obtenção de um motor de simulação distribuído especificando as diferenças entre o motor anterior e o novo motor distribuído, apresentou-se também alguns detalhes de implementação dos módulos componentes do motor distribuído e as principais modificações realizadas para a construção de um simulador de eventos discretos distribuídos.

A seguir, no próximo capítulo, este é dedicado à apresentação dos resultados obtidos a partir da implementação do motor de simulação utilizado neste trabalho comparando-o com o motor anterior e com modelos *exascale*.

4 Resultados

Neste capítulo serão apresentados os testes e os resultados obtidos com o trabalho desenvolvido. Assim, esses resultados permitem a avaliação da implementação do novo motor de simulação distribuído que tem por objetivo não só verificar o comportamento da nova implementação comparando-a com a implementação anterior, mas também a validação de seus resultados.

Assim, este capítulo é dividido em duas seções que tratarão de aspectos distintos do trabalho desenvolvido. Isto posto, a Seção (4.1) abordará uma análise comparativa do novo motor com o motor antigo, e a Seção (4.2) apresentará uma análise de desempenho do motor desenvolvido neste trabalho.

4.1 Análise Comparativa e Validação da Implementação do Motor de Simulação na Versão Sequencial

Nesta seção, trataremos da análise comparativa das versões sequenciais entre a implementação distribuída do motor de simulação desenvolvido neste trabalho com a implementação antiga do motor de simulação construído pelo grupo.

Além disso, o principal objetivo é exibir que o suporte adicionado ao simulador para realizar simulações de sistemas *exascale* mantém os resultados previamente obtidos com a já validada implementação do simulador em Java, ou seja, a adição deste suporte não afeta as métricas previamente obtidas.

Por fim, a seguir a Subseção (4.1.1) apresentará o ambiente de testes em que se realizou os testes para a obtenção de resultados, a Subseção (4.1.2) apresentará a configuração do modelo utilizado no simulador e, por fim, a Subseção (4.1.3) apresentará os resultados.

4.1.1 Ambiente de Testes

Os experimentos conduzidos para a análise comparativa e validação da implementação do motor de simulação com relação à implementação validada do simulador em Java foram realizados na máquina com as seguintes configurações.

- **Sistema Operacional**: Arch Linux x86_64 — 6.5.5-arch-1
- **Processador**: Intel i7-9700K (8) @ 3.6 GHz
- **Memória RAM**: 16 GB

Por fim, devemos observar que ambos testes utilizaram apenas um núcleo de processamento, pois neste caso estamos focados na análise comparativa do desempenho das versões sequenciais de ambos os simuladores e da obtenção das métricas.

4.1.2 Configurações do Modelo

A seguir, temos a configuração do modelo do sistema distribuído de pequeno porte utilizado na obtenção dos resultados das métricas em ambas implementações dos simuladores.

- **Máquina**: São um total de oito máquinas com as seguintes configurações:
  - **Poder de Processamento**: 50.000 MFLOPS
  - **Fator de Carga**: 0 (ocioso)
  - **Quantidade de Núcleos**: 8 núcleos
- **Enlace**: São um total de oito enlaces com as seguintes configurações:
  - **Largura de Banda**: 100 Mbits/s
  - **Fator de Carga**: 0 (ocioso)
  - **Latência**: 5 ms
- **Mestre**: É um total de um mestre com a seguinte configuração:
  - **Escalonador**: Round-Robin
  - **Modelagem de Tarefas**: Tarefas com 10.000 MFLOP de tamanho de processamento e 1 Kbit de tamanho de comunicação.
  - **Distribuição Entre Chegadas**: Fixo (5 segundos)

Por fim, deve-se notar que no subtópico **modelagem de tarefas** não foi especificado a quantidade de tarefas, pois esta irá variar conforme cada caso de teste que será abordado a seguir.

4.1.3 Resultados

A seguir, a Tabela (2) exibe os resultados obtidos a partir da execução das simulações em ambas implementações. Além disso, escolhemos o tempo de simulação global (GST) como a principal métrica a ser analisada, pois esta é computada corretamente se outras métricas também forem, de modo que, a comparação do tempo de simulação global mostra-se segura.

Desta forma, a partir da Tabela (2) nota-se que o erro absoluto entre as métricas do tempo de simulação global (GST) e da máxima quantidade de comunicação (MAC) é zero, ou seja, ambas implementações fornecem o mesmo valor destas métricas. Estes, de

Tabela 2 – Resultados do tempo de simulação global (GST) em segundos e da quantidade máxima de comunicação (MAC) em Mbits realizada por um centro de comunicação em ambos simuladores.

	iSPD-Java		iSPD-Exa			
# Tarefas	GST	MAC	GST	MAC	Erro GST	Erro MAC
100	$5,016 \times 10^{2}$	$1,270 \times 10^{-2}$	$5,016 \times 10^{2}$	$1,270 \times 10^{-2}$	0	0
1.000	$5,001 \times 10^{3}$	$1,221 \times 10^{-1}$	$5,001 \times 10^{3}$	$1,221 \times 10^{-1}$	0	0
10.000	$5,000 \times 10^{4}$	$1,221 \times 10^{0}$	$5,000 \times 10^{4}$	$1,221 \times 10^{0}$	0	0
100.000	$5,000 \times 10^{5}$	$1,221 \times 10^{1}$	$5,000 \times 10^{5}$	$1,221 \times 10^{1}$	0	0
1.000.000	$5,000 \times 10^{6}$	$1,221 \times 10^{2}$	$5,000 \times 10^{6}$	$1,221 \times 10^{2}$	0	0

fato, eram os resultados desejados, pois a adição do suporte a simulações de sistemas de alto desempenho *exascale* não devem afetar a simulação de sistemas de pequeno e médio porte. Assim, validamos o simulador desenvolvido neste trabalho para situações de pequeno e médio porte comparando os resultados obtidos com a então validada implementação do motor antigo.

Além disso, segue a análise de desempenho das versões sequenciais de ambos simuladores.

Tabela 3 – Resultados da análise de desempenho constando valores da média, mediana e desvio padrão dos tempos de execução (ms) e *speedup* obtidos a partir da execução de 10 vezes de cada caso.

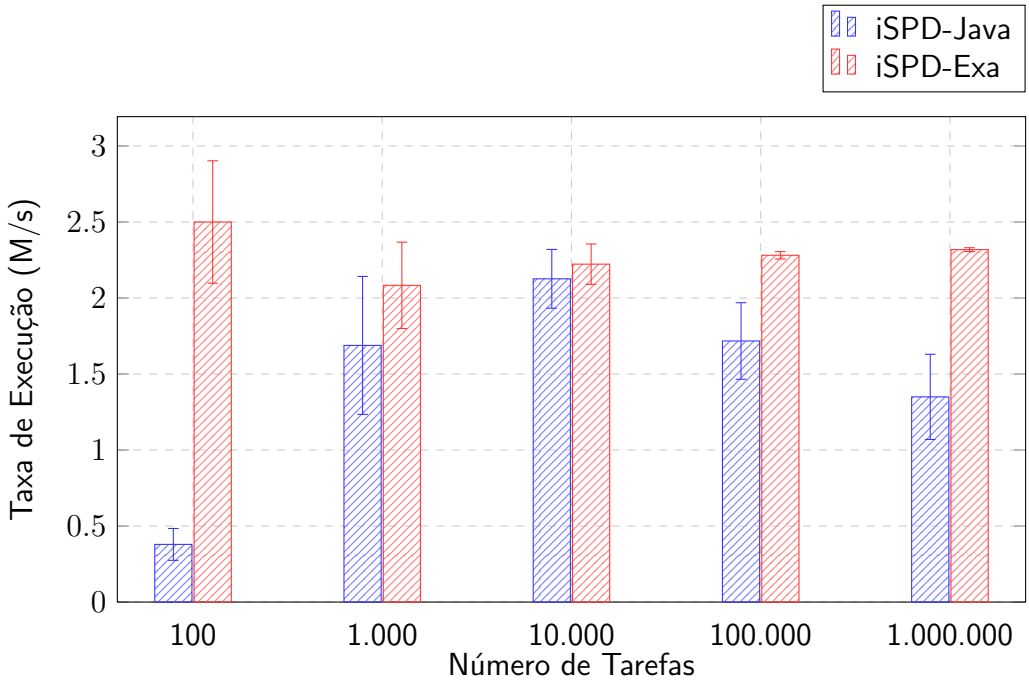
	iSPD-Java			iSPD-Exa			
# Tarefas	Média	Mediana	Desvio Padrão	Média	Mediana	Desvio Padrão	*Speedup*
100	4,2000	3,5000	1,8135	**0,2300**	0,2000	0,0483	**19,500**
1.000	8,4000	8,5000	1,7763	**2,6400**	2,4000	0,4142	**3,2892**
10.000	69,400	68,000	7,2141	**22,970**	22,500	1,4697	**3,0403**
100.000	844,40	841,50	123,26	**219,17**	219,20	2,3391	**3,8530**
1.000.000	10.810	10.708	2348.0	**2160.1**	2156.8	11,682	**5,0055**

Assim, a partir da Tabela (3) é possível identificar que a implementação do motor desenvolvida neste trabalho obtém menores tempos de execução em todos os casos de testes realizados, fato este devido a mudança de linguagem de Java para C++ e no uso de algoritmos e estruturas de dados mais eficientes. Além disso, nota-se que a variabilidade dos tempos de execução são muito menores no iSPD-Exa (nova implementação) com relação ao iSPD-Java. Desse modo, temos que o *speedup* são consideráveis em todos os casos, possuindo um pico de 19,5×. Contudo, nota-se que à medida que cresce o número de tarefas, temos que o *speedup* aproxima-se de 3× a 5×.

Ainda, a Figura (4.1.1) exibe a comparação das velocidades, isto é a taxa de execução em milhões de eventos por segundo, da implementação de ambos os simuladores em suas versões sequenciais, de tal forma que, permitirá uma visão geral do desempenho dos simula-

dores com relação às informações apresentadas na Tabela (3).

Figura 4.1.1 – Avaliação comparativa da taxa de execução de eventos (milhões de eventos por segundo) entre a implementação iSPD-Java e iSPD-Exa, proporcionando uma análise detalhada do desempenho entre as duas implementações.

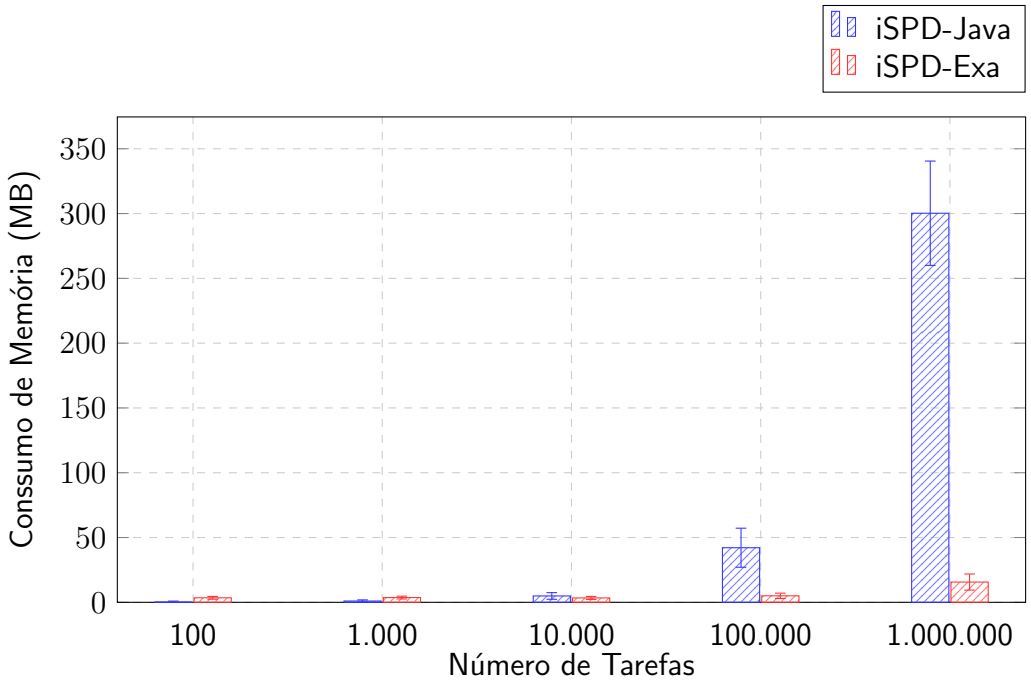


Desta forma, a partir da Figura (4.1.1), pode-se afirmar que as taxas de execução da nova implementação (iSPD-Exa) mostraram-se superiores em todos os casos de testes tomados. Além disso, a nova implementação mostrou-se extremamente mais rápida com uma pequena quantidade de tarefas e muito rápida com grandes quantidades de tarefas. Ainda, para complementar, esta mostrou-se ter pouca variabilidade em suas taxas de execução. Logo, podemos afirmar que as taxas de execução são mais consistentes entre os diferentes casos de testes.

Por outro lado, com relação à implementação antiga (iSPD-Java), é possível ver que há uma grande variação nas taxas de execução à medida que aumenta-se o número de tarefas, e que sua velocidade é lenta em casos pequenos, e, além disso, a partir de 10.000 (caso de maior taxa de execução) tarefas sua taxa de execução diminui progressivamente.

Além disso, a Figura (4.1.2) oferece uma análise comparativa do consumo de memória principal (em MB) nas implementações sequenciais de ambos os simuladores. Esse gráfico proporciona uma visão abrangente do desempenho dos simuladores em relação ao tamanho do modelo simulado, permitindo uma avaliação detalhada da escalabilidade. Ao examinar a representação visual, é possível obter percepções significativas sobre como os simuladores lidam com modelos de diferentes dimensões, facilitando uma compreensão mais profunda do comportamento em termos de consumo de memória em cenários diversos.

Figura 4.1.2 – Comparação do consumo de memória durante as execuções das simulações, medido em megabytes, entre as implementações iSPD-Java e iSPD-Exa, oferecendo uma análise detalhada das demandas de memória principal.



Assim, a partir da Figura (4.1.2), pode-se afirmar que o consumo de memória principal é mínimo e similar durantes os três primeiros casos de testes (100, 1.000 e 10.000 tarefas). No entanto, podemos afirmar que há um crescimento exponencial no consumo de memória principal pela implementação antiga (iSPD-Java), enquanto que a nova implementação consome muito pouco.

Isto posto, o consumo vagaroso da nova implementação deve-se ao fato de que os eventos não são alocados continuamente, pois existe um *memory pool* que será utilizado para o reúso da memória corresponde aos eventos que já foram processados. Assim, permitindo a obtenção da memória de eventos mais rápidos (desde que não irá haver alocação dinâmica) e menor consumo de memória (pois, estaremos realizando o reúso da memória destes eventos).

Por fim, estes resultados eram de fato o esperado devido as modificações e adições cuidadosamente tomadas na implementação do novo motor de simulação que foram citadas na Seção (3.3), produzindo, portanto, um motor de simulação veloz e menos dispendioso em recursos computacionais.

4.2 Análise de Desempenho da Implementação do Motor de Simulação na Versão Distribuída

Nesta seção, trataremos da análise de desempenho da versão distribuída, o qual é o foco da nova implementação do simulador, permitindo, portanto, ver os benefícios da

simulação distribuída e do protocolo de sincronização otimista *Time Warp*.

Por fim, a seguir a Subseção (4.2.1) apresentará o ambiente de testes em que se realizou as medições para a obtenção dos resultados, a Subseção (4.2.2) apresentará a configuração do modelo *exascale* utilizado na simulação e, por fim, a Subseção (4.2.3) apresentará os resultados.

4.2.1 Ambiente de Testes

Os experimentos conduzidos para a análise de desempenho do motor de simulação na versão distribuída foram realizados no *cluster* do GSPD (Grupo de Sistemas Paralelos e Distribuídos) com as seguintes configurações.

- **Número de Nós**: 6 nós de processamento
- **Sistema Operacional**: Debian Bullseye x86_64 — 5.10.0
- **Processador**: Intel i7-3370K (8) @ 3.9 GHz
- **Memória RAM**: 16 GB
- **Memória Secundária**: 500 GB
- **MPI**: OpenMPI 4.1.0

Por fim, utilizamos apenas seis nós de processamento, de modo que, temos um total de 48 núcleos de processamento, 96 GB de memória principal e 3 TB de memória secundária, e os casos de testes serão realizados em 1, 8, . . . , 48 núcleos de processamento.

4.2.2 Configurações do Modelo

A seguir, temos a configuração do modelo de um sistema *exascale* utilizado na obtenção dos resultados das métricas de desempenho do simulador.

- **Rack**: É um total de 56 racks com as seguintes configurações:
  - **Poder de Processamento**: 21 PFLOPS
  - **Fator de Carga**: 0 (ocioso)
  - **Quantidade de Núcleos**: 192.000 núcleos
- **Enlace**: É um total de 169.540 enlaces com as seguintes configurações:
  - **Largura de Banda**: 1 Gbits/s
  - **Fator de Carga**: 0 (ocioso)

  - **Latência**: 1 μs

- **Mestre**: É um total de 56 mestres com a seguinte configuração:
  - **Escalonador**: Round-Robin
  - **Modelagem de Tarefas**: Tarefas com 500 GFLOP de tamanho de processamento e 100 Mbits de tamanho de comunicação.
  - **Quantidade de Tarefas**: 84 milhões.
  - **Distribuição Entre Chegadas**: Poisson ($\lambda = 0.1$)

Assim, podemos afirmar que tal configuração constitui um supercomputador com mais de 10 milhões de núcleos de processamento possuindo, neste modelo, um poder computacional de 1,176 exaFLOPS. Por fim, os componentes constituintes deste modelo estão dispostos em uma topologia Dragonfly.

4.2.3 Resultados

A seguir, é especificado os significados dos parâmetros obtidos que serão exibidos nos resultados posteriormente.

- **Tempo de Execução**: Representa o tempo em segundos necessário para completar a simulação. Para este parâmetro, quanto menor melhor.
- **Taxa de Eventos**: Representa a quantidade de eventos que são processados por segundo. Isto é, indica a velocidade do simulador. Para este parâmetro, quanto maior melhor.

$$\text{taxa de eventos} = \frac{\#\text{eventos totais}}{\text{tempo de execução}} \tag{4.2.1}$$

- **Eventos Totais Líquidos**: Representa a quantidade de eventos que de fato deveriam ter sido processados. Em outras palavras, é a quantidade de eventos processados quando utiliza-se o modo de execução sequencial. Para este parâmetro, não existe um valor ideal, porém existe a situação ideal em que, em diferentes execuções do modelo, todas elas devem produzir o mesmo valor de eventos totais líquidos, ou seja.

$$\#\text{eventos totais líquidos}_{\text{sequencial}} = \#\text{eventos totais líquidos}_{\text{otimista}} \tag{4.2.2}$$

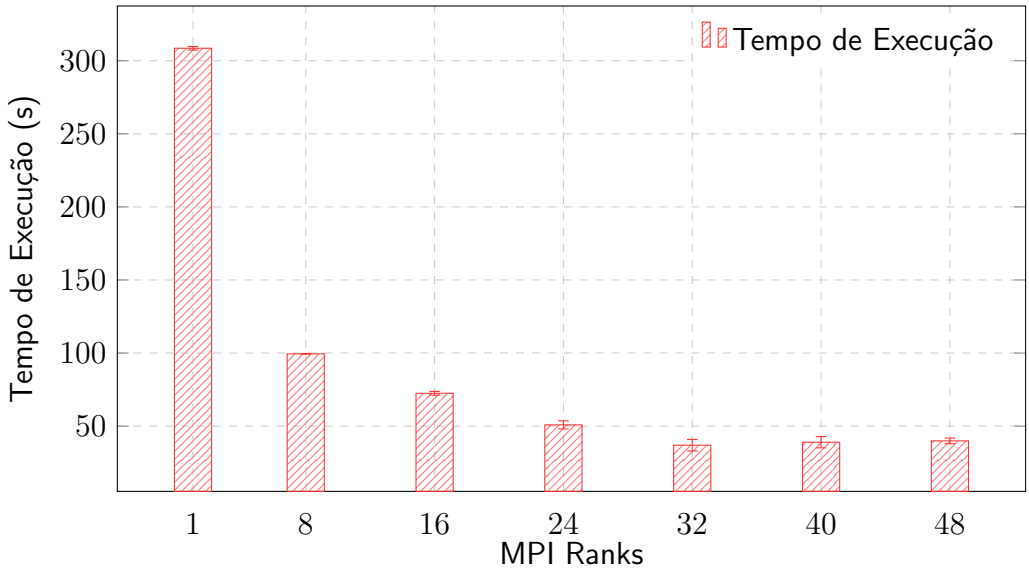
  Neste caso, a quantidade de eventos totais líquidos L_s no modo sequencial é igual a quantidade de eventos totais processados no modo sequencial T_s. Por outro lado, a quantidade de eventos totais líquidos L_o no modo otimista é igual a quantidade

de eventos totais processados no modo otimista T_o menos a quantidade de eventos processados de modo reverso, R, de modo que, ambas parcelas líquidas devem ser iguais, $L_s = L_o$, em outras palavras,

$$T_s = T_o - R \tag{4.2.3}$$

A seguir, a Figura (4.2.1) exibe os tempos de execução (em segundos) necessários para completar a simulação do sistema *exascale*, comparando diferentes números de núcleos atuantes na simulação.

Figura 4.2.1 – Comparação dos tempos de execução (em segundos) do modelo *exascale* com diferentes quantidades de núcleos atuantes na simulação no modo distribuído da nova implementação.



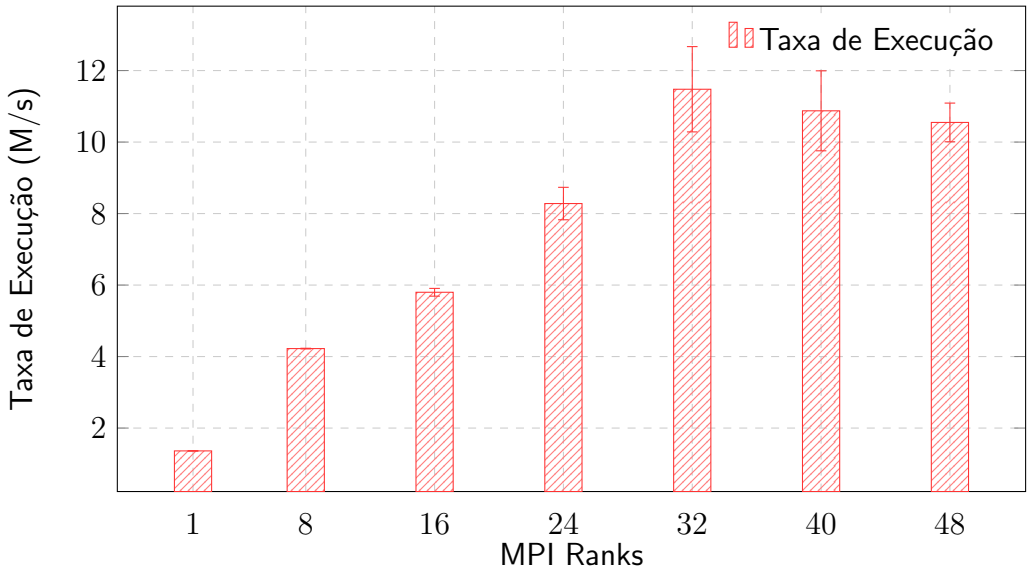
Desta forma, a partir da Figura (4.2.1), pode-se afirmar que à medida que aumenta-se a quantidade de núcleos atuantes na simulação, há uma redução progressiva do tempo de execução até 32 núcleos. Assim, indicando que a distribuição da simulação de eventos discretos, de fato, reduz o tempo total da simulação.

Além disso, devemos notar que o tempo mínimo de execução é obtido com $n = 32$, e este aumenta vagarosamente para os casos de testes seguintes. Isto deve-se ao fato de que à medida que o número de núcleos atuantes na simulação aumenta, a quantidade de trabalho útil por núcleo de simulação diminui (pois, o modelo tem o mesmo tamanho), de tal forma que, o custo de sincronização da simulação entre os núcleos é maior. Contudo, isto é esperado devido ao fato de ser um experimento *strong scaling*.

Ainda, a Figura (4.2.2) exibe a comparação das velocidades, isto é a taxa de execução em milhões de eventos por segundo, da nova implementação no modo distribuído otimista,

de modo que, permitirá uma visão geral do desempenho deste modo nos diferentes casos de teste tomados.

Figura 4.2.2 – Comparação das taxas de execução (em milhões de eventos por segundo) da simulação do modelo *exascale*, com diferentes quantidades de núcleos atuantes na simulação no modo distribuído otimista da nova implementação.

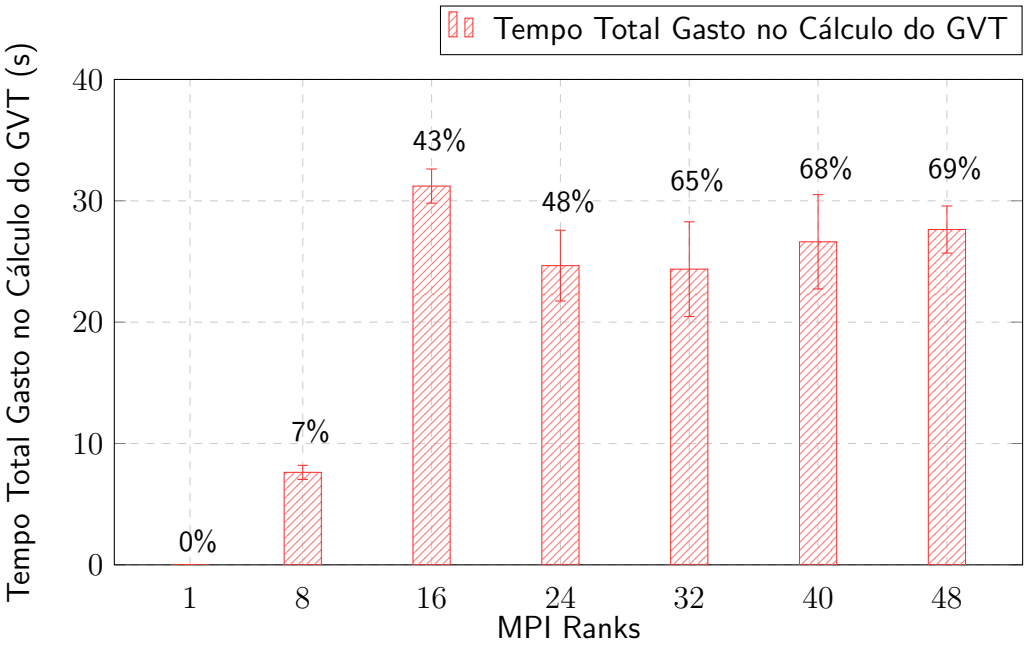


Desta forma, a partir da Figura (4.2.2), pode-se afirmar que as taxas de execução da nova implementação (iSPD-Exa) no modo distribuído otimista, cresce rapidamente e chega no seu pico em $n = 32$, em que atinge aproximadamente 11,5 milhões de eventos por segundo, e diminui vagarosamente a partir disso. Isto deve-se ao mesmo motivo, previamente, citado para o caso do aumento do tempo de execução para os casos de testes posteriores a $n = 32$.

No entanto, é importante notar a diferença das taxas de execução entre o modo distribuído otimista apresentado na Figura (4.2.2), e o modo sequencial apresentado na Figura (4.1.1). Neste caso, a principal diferença está no fato de que no modo sequencial, dificilmente, obtém-se uma taxa de execução maior que 3 milhões de eventos por segundo em nosso ambiente de testes, enquanto que no modo distribuído otimista foi possível obter uma taxa de aproximadamente 11,5 milhões de eventos por segundo. Logo, isto exibe como a distribuição da simulação de eventos discretos utilizando o protocolo otimista *Time Warp*, permite que a taxa de execução de eventos aumente significativamente com relação ao modo sequencial.

Para aprofundar a explicação sobre o custo de sincronização mencionado anteriormente, a Figura (4.2.3) apresenta o custo associado ao cálculo do tempo global virtual (custo de sincronização global), variando conforme o número de núcleos envolvidos na simulação. Dessa maneira, a análise detalhada dessa figura, juntamente com a explicação subsequente, fornecerá uma base sólida para compreender esses casos específicos, elucidando o impacto do número de núcleos na eficiência do cálculo do tempo global virtual e, por conseguinte, na sincronização global.

Figura 4.2.3 – Comparação dos tempos despendidos para o cálculo do tempo global virtual (GVT) (em segundos) da simulação do modelo *exascale*, com diferentes quantidades de núcleos atuantes na simulação no modo distribuído otimista da nova implementação.



Desta forma, a partir da Figura (4.2.3), temos que para $n = 1$ o custo do cálculo do tempo global virtual é zero, pois, neste caso, há somente um núcleo atuante na simulação, de modo que, não há necessidade para tal cálculo. Além disso, é importante notar que a variação deste custo entre os casos $n = 1$ e $n = 8$ é diferente do que entre os casos $n = 8$ e $n \geq 16$, pois para $n \in \{1, 2, \ldots, 8\}$, estamos utilizando somente um nó de processamento, de tal forma que, toda a comunicação é realizada pelo barramento interno.

Por outro lado, para $n \geq 16$ estamos tratando com mais de um nó, de modo que, toda a comunicação para o cálculo do GVT é realizada por meio de uma rede de interconexão, muito mais lenta do que o barramento interno, de tal forma que, é visível o aumento nos custos do cálculo do GVT na Figura (4.2.3).

Ainda, na Figura (4.2.3) sobre cada medição, temos um valor que indica a porcentagem do tempo total de execução que corresponde ao tempo gasto calculando o GVT. Desta forma, é evidenciado o aumento da presença do cálculo do GVT para os casos $n = 40$ e $n = 48$ em relação a $n = 32$, mostrando-se ser um dos fatores que influenciaram o aumento do tempo de execução.

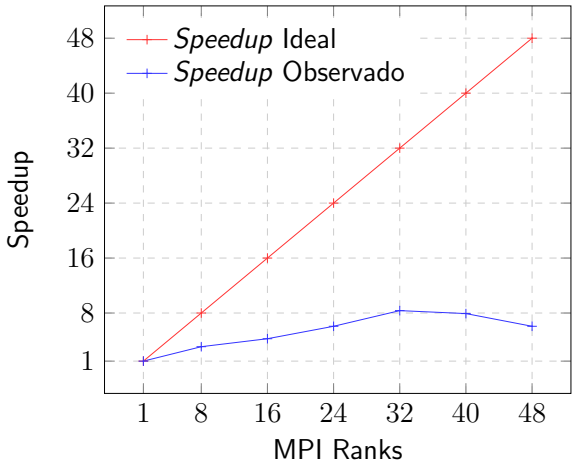
No entanto, outro fator é o desbalanceamento de carga entre os nós, isto é, processos lógicos em nós com menor carga tendem a avançar mais rapidamente seu tempo lógico virtual (LVT), assim, requisitam o cálculo do GVT antecipadamente e têm que esperar os outros processos lógicos chegarem no estado para o cálculo do GVT. Isto é, os processos lógicos que poderiam estar realizando trabalho útil estão esperando o cálculo do GVT por parte dos

processos lógicos que ainda estão realizando algum processamento.

Neste caso, isto deve-se ao fato de que o algoritmo implementado pelo ROSS para o cálculo do GVT, trata-se de um algoritmo síncrono, em que de maneira subjacente utiliza o MPI_AllReduce que é bloqueante e introduz uma sincronização coletiva entre os diferentes processos lógicos ocasionando, portanto, o bloqueio dos processos lógicos que calcularam a sua parte no processo do cálculo do GVT.

Isto posto, a Figura (4.2.4) a seguir apresenta a comparação do *speedup* ideal com o *speedup* observado.

Figura 4.2.4 – Comparação do *speedup* ideal com o *speedup* observado, este obtido com os resultados apresentados.



A partir da análise da Figura (4.2.4), é possível afirmar que o *speedup* máximo observado alcançou 8,35×, apresentando um comportamento *sublinear* conforme previsto pelos fatores mencionados anteriormente. Este resultado indica que um sistema de eventos discretos paralelo revela limitações, especialmente nas interfaces de entrada e saída, bem como em aspectos além do escopo da implementação do motor, em detrimento do processamento efetivo dos eventos. Essa conclusão destaca a necessidade de considerar esses fatores ao avaliar o desempenho do sistema em cenários práticos e complexos.

Por fim, para complementar a validação do motor de simulação distribuído, a Tabela (4) expõe a quantidade de eventos totais líquidos processados em cada caso de teste. É esperado que todos os valores sejam idênticos, independentemente do modo de execução. Essa uniformidade é crucial, conforme especificado pela equipe do ROSS[1], sendo uma etapa fundamental na validação de qualquer motor de simulação implementado. A observação desses resultados reforça a confiabilidade do motor distribuído, assegurando que a consistência na quantidade de eventos processados seja mantida em diferentes cenários de teste.

[1] https://ross-org.github.io/model-dev/debugging.html

Tabela 4 – Quantidade de eventos totais líquidos processados pelo motor de simulação nos diferentes casos de testes. Neste caso, $n = 1$ representa o modo sequencial, enquanto todos os outros representam o modo distribuído otimista.

iSPD-Exa		
# Núcleos	# Eventos Totais Líquidos	Erro
1	$4,2 \times 10^8$	Não Aplicável
8	$4,2 \times 10^8$	0
16	$4,2 \times 10^8$	0
24	$4,2 \times 10^8$	0
32	$4,2 \times 10^8$	0
40	$4,2 \times 10^8$	0
48	$4,2 \times 10^8$	0

Desta forma, a partir da Tabela (4), podemos afirmar que o nosso motor de simulação implementado está de acordo com as etapas de validação proposto pela equipe do ROSS, e, portanto, a escolha do modo de execução não afetará os resultados obtidos.

4.3 Considerações Finais

Neste capítulo foram exibido os testes e resultados obtidos com a implementação de um motor de simulação distribuído. Primeiramente, foram realizados uma comparação detalhada entre as versões sequenciais de ambos os simuladores, com a finalidade de garantir que a nova implementação ainda permite a simulação de sistemas de pequeno e médio porte como realizados na implementação antiga, obtendo-se os mesmos resultados.

Desse modo, apresentou-se também os resultados de desempenho do simulador a partir da simulação de um modelo *exascale* com diferentes quantidades de núcleos, que mostraram-se satisfatórios, exibindo que a distribuição da simulação por múltiplos núcleos, de fato, impacta no tempo de execução, tornado-o veloz. Além disso, exibiu-se que, independentemente, do modo de execução, seja este sequencial ou distribuído, o motor é **determinístico** e **reprodutível**.

Por fim, podemos afirmar que os objetivos foram alcançados com a implementação do motor de simulação distribuído otimista utilizando o protocolo *Time Warp*.

5 Conclusões

Neste trabalho, implementou-se um motor de simulação distribuído que fosse capaz de simular muitos núcleos de processamento de um sistema distribuído em tempo hábil, utilizando-se da distribuição da simulação de eventos discretos em múltiplos nós de processamento e do protocolo de sincronização otimista *Time Warp*, com a finalidade de manter a ordem de causalidade entre os eventos, de forma que, obtivéssemos resultados compatíveis com o esperado.

Além disso, este trabalho apresentou os conceitos fundamentais de sistemas de eventos discretos paralelos que surgiram para tornar a execução de simulações velozes por meio da distribuição da simulação. Ainda, apresentou a definição do problema da sincronização e os protocolos de sincronização existentes na literatura que têm por finalidade manter a ordem de causalidade entre os eventos. Desse modo, apresentou-se também o funcionamento do protocolo conservativo *CMB* e do protocolo otimista *Time Warp*, em que o último é o foco deste trabalho.

Ainda, este trabalho apresentou a metodologia adotada, conjuntamente, com os passos necessários para a implementação de um motor de simulação distribuído. Dessa forma, foram apresentados os motivos para cada adição ou modificação, com o intuito de, exibir ao leitor do por quê das decisões tomadas. Isto posto, foram apresentados os módulos essenciais para o funcionamento do motor de simulação e algumas especificidades de seu funcionamento.

Igualmente, apresentou-se os resultados obtidos com a nova implementação. Primeiramente, comparamos os resultados da simulação com a antiga implementação e a nova implementação, com a finalidade de, exibir que o suporte adicionado para simular paralelamente, não altera os resultados obtidos, independentemente, do modo de execução escolhido no novo motor de simulação. Além disso, foram apresentados os resultados de desempenho da nova implementação, os quais mostraram-se satisfatórios, exibindo que o tempo de execução reduzia consideravelmente à medida que o número de núcleos de simulação aumenta.

Contudo, notou-se que existem fatores decisivos envolvidos da distribuição da simulação que afetam o desempenho do simulador, como citado, anteriormente, no Capítulo 4. Assim, embora, resultados satisfatórios foram obtidos, nem todos os modelos têm a característica de serem paralelizáveis ou distribuídos, de tal forma que, o custo incorrido pela sincronização da distribuição, possa sobressair no tempo de execução total da simulação, independentemente, do protocolo de sincronização utilizado.

Por fim, podemos afirmar que os métodos adotados para a implementação do motor de simulação e os resultados obtidos, a partir desta, foram satisfatórios exibindo que a distribuição da simulação de eventos discretos utilizando o protocolo otimista *Time Warp* reduz o

tempo de execução da simulação.

5.1 Trabalhos Futuros

Como consequência da realização deste trabalho, é possível apresentar direções de trabalhos futuros.

- **Protocolos de Sincronização**: Implementação de outros protocolos de sincronização, tanto conservativos quanto otimistas, para o estudo dos seus benefícios e como estes podem afetar o desempenho do simulador.
- **Métodos de Distribuição dos Processos Lógicos**: Atualmente, a distribuição dos processos lógicos segue ou uma distribuição estaticamente definida dentro do motor de simulação. Contudo, tal distribuição estaticamente definida, nem sempre será o melhor mapeamento dos processos lógicos entre os núcleos de simulação, de tal forma que, isto afetará o desempenho do simulador. Assim, métodos de distribuição automática baseada no modelo que será simulado pode ajudar a aumentar o desempenho do simulador.
- **Algoritmos de Cálculo de GVT Assíncrono**: Atualmente, o algoritmo utilizado para o cálculo do tempo global virtual (GVT) é síncrono, e produz uma sincronização coletiva entre os processos lógicos, podendo bloqueá-los por mais tempo do que o necessário. Desta forma, algoritmos assíncronos podem mitigar tal problema, podendo aumentar o desempenho do simulador.

5.2 Lista de Publicações e Apresentações

Nesta seção, lista-se as apresentações e as publicações resultantes a partir do desenvolvimento deste trabalho nos seguintes eventos.

1. **Simulação de Sistemas Exascale usando Time Warp**: Artigo aceito e publicado. Este trabalho foi apresentando oralmente no evento científico **XIV Escola Regional de Alto Desempenho de São Paulo** ocorrida de 17/07/2023 a 19/07/2023 na **Universidade Federal de São Paulo (UNIFESP), Campus de São José dos Campos - Unidade Parque Tecnológico**.
2. **Simulação de Sistemas Exascale usando Time Warp**: Trabalho apresentado e aprovado. Este trabalho foi apresentado oralmente no evento científico **XXXV Congresso de Iniciação Científica** ocorrida de 03/10/2023 a 05/10/2023 na **Universidade Estadual Paulista "Júlio de Mesquita Filho" (UNESP), Campus de São José do Rio Preto - IBILCE**.

Referências

ALEXANDROV, A. et al. *LogGP: Incorporating Long Messages into the LogP Model — One Step Closer towards a Realistic Model for Parallel Computation*. USA, 1995.

BANKS, J. Introduction to simulation. In: *Proceedings of the 31st Conference on Winter Simulation: Simulation—a Bridge to the Future - Volume 1*. New York, NY, USA: Association for Computing Machinery, 1999. (WSC '99), p. 7–13. ISBN 0780357809. Disponível em: ⟨https://doi.org/10.1145/324138.324142⟩.

BARNES, P. D. et al. Warp speed: Executing time warp on 1,966,080 cores. In: *Proceedings of the 1st ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. New York, NY, USA: Association for Computing Machinery, 2013. (SIGSIM PADS '13), p. 327–336. ISBN 9781450319201. Disponível em: ⟨https://doi.org/10.1145/2486092.2486134⟩.

BAUER, D. et al. Seven-o'clock: A new distributed gvt algorithm using network atomic operations. In: *Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation*. USA: IEEE Computer Society, 2005. (PADS '05), p. 39–48. ISBN 0769523838. Disponível em: ⟨https://doi.org/10.1109/PADS.2005.27⟩.

BRYANT, R. E. *SIMULATION OF PACKET COMMUNICATION ARCHITECTURE COMPUTER SYSTEMS*. USA, 1977.

BUYYA, R.; MURSHED, M. *GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing*. 2002.

CAROTHERS, C. Enabling co-design of multi-layer exascale storage architectures. 8 2015. Disponível em: ⟨https://www.osti.gov/biblio/1311761⟩.

CAROTHERS, C.; BAUER, D.; PEARCE, S. Ross: a high-performance, low memory, modular time warp system. In: *Proceedings Fourteenth Workshop on Parallel and Distributed Simulation*. [S.l.: s.n.], 2000. p. 53–60.

CASANOVA, H. et al. Versatile, scalable, and accurate simulation of distributed applications and platforms. *Journal of Parallel and Distributed Computing*, Elsevier, v. 74, n. 10, p. 2899–2917, jun. 2014. Disponível em: ⟨http://hal.inria.fr/hal-01017319⟩.

CASANOVA, H.; LEGRAND, A.; QUINSON, M. Simgrid: A generic framework for large-scale distributed experiments. In: *Tenth International Conference on Computer Modeling and Simulation (uksim 2008)*. [S.l.: s.n.], 2008. p. 126–131.

CHANDY, K. M.; MISRA, J. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Trans. Softw. Eng.*, IEEE Press, v. 5, n. 5, p. 440–452, sep 1979. ISSN 0098-5589. Disponível em: ⟨https://doi.org/10.1109/TSE.1979.230182⟩.

CHANG, C. et al. Simulations in the era of exascale computing. *Nature Reviews Materials*, v. 8, n. 5, p. 309–313, 03 2023.

CHETLUR, M.; WILSEY, P. A. Causality representation and cancellation mechanism in time warp simulations. In: *Proceedings of the Fifteenth Workshop on Parallel and Distributed Simulation*. USA: IEEE Computer Society, 2001. (PADS '01), p. 165–172. ISBN 076951104X.

DAS, S. R. Adaptive protocols for parallel discrete event simulation. In: *Proceedings of the 28th Conference on Winter Simulation*. USA: IEEE Computer Society, 1996. (WSC '96), p. 186–193. ISBN 0780333837. Disponível em: ⟨https://doi.org/10.1145/256562.256602⟩.

DICKENS, P. M. et al. Analysis of bounded time warp and comparison with yawns. *ACM Trans. Model. Comput. Simul.*, Association for Computing Machinery, New York, NY, USA, v. 6, n. 4, p. 297–320, oct 1996. ISSN 1049-3301. Disponível em: ⟨https://doi.org/10.1145/240896.240913⟩.

FLYNN, M. Very high-speed computing systems. *Proceedings of the IEEE*, v. 54, n. 12, p. 1901–1909, 1966.

FUJIMOTO, R. M. Parallel discrete event simulation. *Commun. ACM*, Association for Computing Machinery, New York, NY, USA, v. 33, n. 10, p. 30–53, oct 1990. ISSN 0001-0782. Disponível em: ⟨https://doi.org/10.1145/84537.84545⟩.

FUJIMOTO, R. M. Research challenges in parallel and distributed simulation. *ACM Trans. Model. Comput. Simul.*, Association for Computing Machinery, New York, NY, USA, v. 26, n. 4, may 2016. ISSN 1049-3301. Disponível em: ⟨https://doi.org/10.1145/2866577⟩.

JAFER, S.; LIU, Q.; WAINER, G. Synchronization methods in parallel and distributed discrete-event simulation. *Simulation Modelling Practice and Theory*, v. 30, p. 54–73, 2013. ISSN 1569-190X. Disponível em: ⟨https://www.sciencedirect.com/science/article/pii/S1569190X12001244⟩.

JAIN, N. et al. Predicting the performance impact of different fat-tree configurations. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. New York, NY, USA: Association for Computing Machinery, 2017. (SC '17). ISBN 9781450351140. Disponível em: ⟨https://doi.org/10.1145/3126908.3126967⟩.

JAIN, R. *The art of computer systems performance analysis - techniques for experimental design, measurement, simulation, and modeling*. [S.l.]: Wiley, 1991. I-XXVII, 1-685 p. (Wiley professional computing). ISBN 978-0-471-50336-1.

JEFFERSON, D. R. Virtual time. *ACM Trans. Program. Lang. Syst.*, Association for Computing Machinery, New York, NY, USA, v. 7, n. 3, p. 404–425, jul 1985. ISSN 0164-0925. Disponível em: ⟨https://doi.org/10.1145/3916.3988⟩.

JEFFERSON, D. R.; SOWIZRAL, H. Fast concurrent simulation using the time warp mechanism. part i. local control. In: . [S.l.: s.n.], 1982.

LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, Association for Computing Machinery, New York, NY, USA, v. 21, n. 7, p. 558–565, jul 1978. ISSN 0001-0782. Disponível em: ⟨https://doi.org/10.1145/359545.359563⟩.

LIU, N. et al. Fattreesim: Modeling large-scale fat-tree networks for hpc systems and data centers using parallel and discrete event simulation. In: *Proceedings of the 3rd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. New York, NY, USA: Association for Computing Machinery, 2015. (SIGSIM PADS '15), p. 199–210. ISBN 9781450335836. Disponível em: ⟨https://doi.org/10.1145/2769458.2769474⟩.

MANACERO, A. et al. Ispd: An iconic-based modeling simulator for distributed grids. In: *Proceedings of the 45th Annual Simulation Symposium*. San Diego, CA, USA: Society for Computer Simulation International, 2012. (ANSS '12). ISBN 9781618397843.

MISRA, J. Distributed discrete-event simulation. *ACM Comput. Surv.*, Association for Computing Machinery, New York, NY, USA, v. 18, n. 1, p. 39–65, mar 1986. ISSN 0360-0300. Disponível em: ⟨https://doi.org/10.1145/6462.6485⟩.

MUBARAK, M. et al. Modeling a million-node dragonfly network using massively parallel discrete-event simulation. In: *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. USA: IEEE Computer Society, 2012. (SCC '12), p. 366–376. ISBN 9780769549569. Disponível em: ⟨https://doi.org/10.1109/SC.Companion.2012.56⟩.

MUBARAK, M. et al. A case study in using massively parallel simulation for extreme-scale torus network codesign. In: *Proceedings of the 2nd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. New York, NY, USA: Association for Computing Machinery, 2014. (SIGSIM PADS '14), p. 27–38. ISBN 9781450327947. Disponível em: ⟨https://doi.org/10.1145/2601381.2601383⟩.

NICOL, D. M. The cost of conservative synchronization in parallel discrete event simulations. *J. ACM*, Association for Computing Machinery, New York, NY, USA, v. 40, n. 2, p. 304–333, apr 1993. ISSN 0004-5411. Disponível em: ⟨https://doi.org/10.1145/151261.151266⟩.

NKETSA, A.; KHALIFA, N. B. Timed petri nets and prediction to improve the chandy-misra conservative-distributed simulation. *Appl. Math. Comput.*, Elsevier Science Inc., USA, v. 120, n. 1–3, p. 235–254, may 2001. ISSN 0096-3003. Disponível em: ⟨https://doi.org/10.1016/S0096-3003(99)00244-1⟩.

PERUMALLA, K. S. Parallel and distributed simulation: Traditional techniques and recent advances. In: *Proceedings of the 38th Conference on Winter Simulation*. [S.l.]: Winter Simulation Conference, 2006. (WSC '06), p. 84–95. ISBN 1424405017.

PRINCIPE, M. et al. *Transparent Distributed Cross-State Synchronization in Optimistic Parallel Discrete Event Simulation*. [S.l.], 2017.

QUAGLIA, F.; CORTELLESSA, V.; CICIANI, B. Trade-off between sequential and time warp-based parallel simulation. *IEEE Transactions on Parallel and Distributed Systems*, v. 10, n. 8, p. 781–794, 1999.

QUINSON, M.; ROSA, C.; THIéRY, C. Parallel simulation of peer-to-peer systems. In: *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*. [S.l.: s.n.], 2012. p. 668–675.

RAJAN, R.; RADHAKRISHNAN, R.; WILSEY, P. Dynamic cancellation: Selecting time warp cancellation strategies at runtime. *VLSI Design*, v. 9, 01 1999.

RONNGREN, R. et al. A comparative study of state saving mechanisms for time warp synchronized parallel discrete event simulation. In: *Proceedings of the 29th Annual Simulation Symposium*. [S.l.: s.n.], 1996. p. 5–14.

SILVA, D. T. d. Abordagem icônica para modelagem e simulação de ambientes de computação em nuvem. UNESP, São José do Rio Preto, SP, Brazil, p. 1–98, aug 2015. Disponível em: ⟨https://repositorio.unesp.br/handle/11449/138453⟩.

SPOLON, R. Um método para avaliação de desempenho de protocolos de sincronização otimistas para simulação distribuída. USP, São Carlos, SP, Brazil, p. 1–249, jul 2001. Disponível em: ⟨https://www.teses.usp.br/teses/disponiveis/76/76132/tde-19042002-162157/pt-br.php⟩.

VEE, V.-Y.; HSU, W. J. Parallel discrete event simulation: A survey. In: . [S.l.: s.n.], 2007.

WOLFE, N. et al. Modeling a million-node slim fly network using parallel discrete-event simulation. In: *Proceedings of the 2016 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. New York, NY, USA: Association for Computing Machinery, 2016. (SIGSIM-PADS '16), p. 189–199. ISBN 9781450337427. Disponível em: ⟨https://doi.org/10.1145/2901378.2901389⟩.

WOLFE, N. et al. Modeling large-scale slim fly networks using parallel discrete-event simulation. *ACM Trans. Model. Comput. Simul.*, Association for Computing Machinery, New York, NY, USA, v. 28, n. 4, aug 2018. ISSN 1049-3301. Disponível em: ⟨https://doi.org/10.1145/3203406⟩.