Paralelização e Otimização da Simulação Discreta de Eventos em Processador *Multicore*

Tobias A. Gehlen[1], Nahri Moreano[1]

[1]Faculdade de Computação
Universidade Federal de Mato Grosso do Sul (UFMS)

tobias.gehlen@ufms.br, nahri@facom.ufms.br

Resumo. *A simulação é usada para estudar o comportamento de sistemas físicos e sua paralelização agiliza a solução, porém causa um overhead para assegurar o respeito às relações de causalidade do sistema. Este trabalho apresenta o desenvolvimento de um simulador paralelo de eventos discretos em uma arquitetura multicore de memória compartilhada. São analisadas questões como estrutura de dados da fila de eventos, impacto da carga de trabalho da aplicação, exploração de lookahead, avanço do horizonte, suspensão do gerenciador, desempenho e escalabilidade. O simulador paralelo obteve speedups de até 7,01 em relação ao simulador sequencial, em um processador com quatro núcleos e capaz de executar oito threads simultaneamente.*

Abstract. *Simulation is used to study the behavior of physical systems. Parallel simulation accelerates the solution of these systems, but it causes overhead to ensure respect for the causality relations of the system. This work presents the project and implementation of a parallel discrete event simulator in a multicore architecture with shared memory. We examine issues as data structure for the event queue, impact of application workload, lookahead exploitation, techniques for advancing the simulation horizon and suspending the manager mechanism, performance, and scalability. The parallel simulator achieved speedups up to 7.01, when compared to the sequential simulator, on a processor with four cores and capable of executing eight threads simultaneously.*

1. Introdução

O estudo de sistemas físicos se dá ao modelar, através de relações lógicas e matemáticas, as interações entre as entidades que atuam no sistema em questão. Essas relações permitem estimar como o sistema se comporta em diferentes circunstâncias. Em modelos simples, é possível resolvê-las de modo analítico, porém muitas vezes é necessário o uso da simulação por computador para estudar sistemas mais complexos e validar soluções analíticas. Aplicações de áreas como física, engenharia, computação, economia e militar podem ser analisadas através da simulação [Law and Kelton 2015, Sokolowski and Banks 2010].

Na simulação de eventos discretos, a aplicação modela o sistema físico sendo estudado como um conjunto de processos lógicos (PLs). O sistema físico evolui instantaneamente em instantes distintos no tempo, quando ocorrem eventos que representam as interações entre os PLs. Assim, a simulação avança com o tratamento de eventos que ocorrem em determinados instantes de tempo. Os eventos causam

modificações no estado do sistema, representado por variáveis de estado que descrevem a situação do sistema em determinado instante [Banks et al. 2010].

Devido à complexidade dos sistemas físicos modelados, a simulação executada sequencialmente pode apresentar longos tempos de execução, tornando inviável a sua utilização. Contudo, pela natureza do problema que divide o sistema em PLs, é possível paralelizar a execução a fim de reduzir o seu tempo de processamento e viabilizar o seu uso para problemas reais [Fujimoto 2000]. A exploração do paralelismo em uma simulação de eventos discretos é realizada através da execução concorrente dos PLs que compõem o sistema, usando para isso os vários processadores ou núcleos da arquitetura paralela. Entretanto, essa paralelização torna mais complexo garantir que as relações de causalidade do sistema físico sejam respeitadas no avanço da simulação [Fujimoto 2015].

A simulação paralela foi originalmente desenvolvida para plataformas de computação paralela com memória distribuída, onde a interação entre os PLs é realizada através da troca de mensagens usando uma rede de interconexão. Os algoritmos otimistas de simulação paralela não evitam a ocorrência de erros de causalidade, permitindo que diferentes PLs executem eventos fora de ordem, e se baseiam em mecanismos de detecção e recuperação de erros de causalidade. Esses mecanismos têm um impacto no ganho de desempenho obtido com a paralelização e demandam grandes recursos de memória [Jefferson 1985].

Algoritmos conservadores de simulação paralela mantêm o respeito às relações causais do sistema físico evitando completamente a possibilidade de um evento ser executado fora de ordem. Para isso, baseiam-se em estratégias para identificar eventos seguros de serem processados, e PLs que não possuem eventos seguros para executar são bloqueados, o que pode levar a situações de *deadlock*. Assim, mecanismos de prevenção ou detecção e recuperação de *deadlock* são utilizados. Tais estratégias e mecanismos também têm um impacto no desempenho [Chandy and Misra 1981, Misra 1986]. Entretanto, em uma plataforma de memória compartilhada, o compartilhamento de informações sobre a simulação entre os PLs permite a redução do *overhead* causado pelos mecanismos de avanço da simulação e tratamento de *deadlock* dos algoritmos conservadores.

Um simulador paralelo conservador, usando memória compartilhada e o sistema Synapse, é desenvolvido em [Wagner et al. 1989], obtendo *speedup* de 3,67 com 9 processadores. O mecanismo conservador de simulação paralela é implementado usando troca de mensagens e a linguagem Java em um processador *multicore* em [Ahmed et al. 2012], porém não são apresentados resultados de desempenho. Em [De Munck et al. 2010], um simulador paralelo conservador é desenvolvido baseado em troca de mensagens, usando a linguagem Java e obtendo *speedup* de quase 7 com dois processadores *multicore* com 4 núcleos.

Este trabalho apresenta o projeto e a implementação de um simulador paralelo conservador de eventos discretos para execução em uma arquitetura *multicore* com memória compartilhada. O objetivo é obter um simulador que permita a simulação de aplicações em um tempo reduzido de execução, em comparação com um simulador sequencial.

O texto é composto de quatro seções. A Seção 2 descreve detalhes do

desenvolvimento do simulador paralelo e de um simulador sequencial usado como referência, além das aplicações utilizadas para testes. A Seção 3 discute os resultados obtidos através de uma avaliação experimental de desempenho. Por fim, a Seção 4 conclui a discussão e apresenta possíveis trabalhos futuros.

2. Simuladores Desenvolvidos

Um importante aspecto do projeto dos simuladores sequencial e paralelo é a separação entre o mecanismo de simulação e a aplicação que modela o sistema físico. O mecanismo de simulação determina quais eventos são seguros de serem tratados, enquanto a aplicação gera e trata eventos que atualizam as variáveis de estado do sistema. Essa distinção permite que o simulador seja genérico e independente da aplicação, o que possibilita o seu uso para analisar o comportamento de diferentes sistemas físicos.

Os simuladores sequencial e paralelo foram implementados usando a linguagem C. O simulador paralelo utiliza o modelo de programação paralela OpenMP, que oferece uma API (*Application Programming Interface*) contendo um conjunto de diretivas de compilação, uma biblioteca de funções e variáveis de ambiente para criação e sincronização de *threads* [OpenMP Architecture Review Board 2020]. Devido à complexidade do mecanismo de simulação paralela, foi necessário o uso de semáforos para realizar sincronizações mais elaboradas.

2.1. Simulador Sequencial

Um simulador sequencial foi desenvolvido para fins de verificação da corretude e comparação de desempenho. Esse simulador possui uma fila de eventos a serem tratados, ordenada pelo tempo dos eventos, e um relógio com o tempo atual da simulação. A cada iteração, o evento de menor tempo é removido da fila, o relógio é atualizado e o evento é tratado pela aplicação. O tratamento do evento pode gerar novos eventos, que são inseridos na fila respeitando a ordem de tempo e inserção. A simulação é finalizada quando não há mais eventos a tratar. O simulador sequencial é descrito no Algoritmo 1 e sua organização ilustrada na Figura 1(a).

Algoritmo 1: Simulador sequencial

```
while filaEventos ≠ vazia do                    // Enquanto não terminou simulação
    evento := removeEvento(filaEventos)         // Remove evento de menor tempo
    relógio := evento.tempo                     // Atualiza relógio
    aplicação.trataEvento(evento)               // Pode gerar novos eventos
```

2.2. Simulador Paralelo

No simulador paralelo, para explorar paralelismo no tratamento dos eventos, a execução de cada PL é atrelada a uma *thread*. Cada PL possui uma fila de eventos destinados a ele, ordenada pelo tempo, e um relógio com o tempo atual do PL na simulação. Assim, cada PL pode estar em um tempo de simulação diferente, porém respeitando a ordem de tempo dos eventos. Uma *thread* adicional realiza a tarefa de gerenciador da simulação, que analisa o estado dos PLs e determina um tempo, denominado horizonte [Fujimoto 2000], tal que todos os eventos até esse tempo são seguros de serem tratados. A Figura 1(b) ilustra a organização do simulador paralelo.

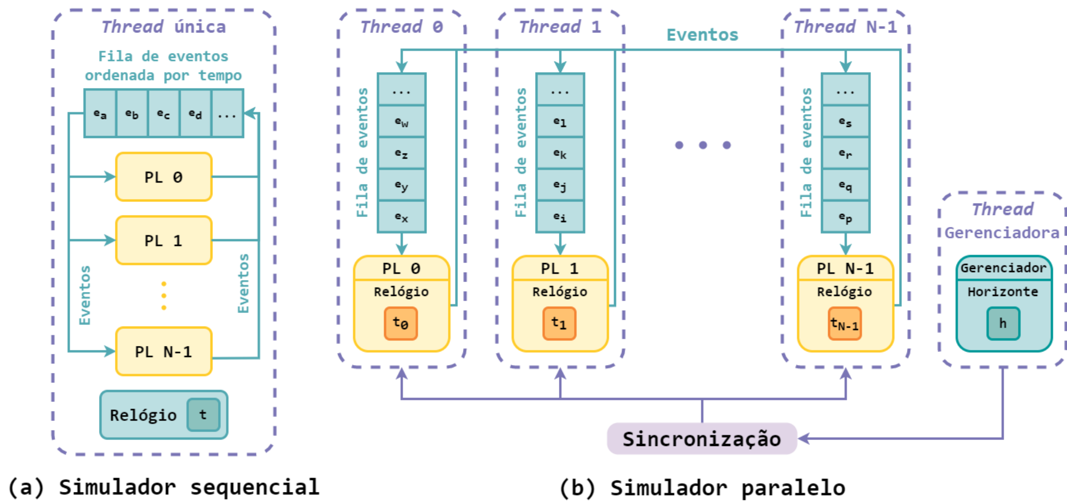

Figura 1. Organização dos simuladores sequencial e paralelo

Um evento de tempo t destinado a um PL_i é dito seguro se e somente se não existe a possibilidade do PL_i receber outro evento de tempo menor que t de qualquer PL, incluindo ele mesmo. Para avançar o horizonte, o simulador pode usar o *lookahead* L, que é um intervalo de tempo definido pela aplicação, tal que um PL com relógio t gera novos eventos com tempo pelo menos $t + L$ [Fujimoto 2000]. O gerenciador determina o horizonte somando o mínimo dentre os relógios de todos os PLs com L, garantindo que nenhum evento será gerado com tempo menor. Assim, todos os PLs podem tratar eventos com tempo menor ou igual ao horizonte.

Se um PL tem a fila de eventos vazia ou não possui eventos seguros para tratar, a *thread* correspondente é bloqueada com o uso de semáforos. É tarefa do gerenciador, atualizar o horizonte e verificar se existem eventos seguros e desbloquear as *threads* relacionadas. Como a execução da *thread* gerenciadora representa um *overhead* do mecanismo de simulação, ela é suspensa por um tempo de acordo com a quantidade de PLs em espera (bloqueados mas com fila de eventos não vazia) e um parâmetro H de hibernação determinado empiricamente, como mostrado na Equação 1.

$$\textit{Tempo suspensão} = H \times \frac{\textit{Número de PLs} - \textit{Número de PLs em espera}}{\textit{Número de PLs}} \quad (1)$$

Contudo, podem ocorrer situações em que nenhum PL possui evento seguro para tratar e não é possível avançar o horizonte pois um PL_i possui relógio com valor muito baixo e não recebeu mais eventos, portanto seu relógio permanece inalterado. Nesse caso, o mínimo dos relógios de todos os PLs é o relógio do PL_i, o que impede que o horizonte avance. Essa é uma situação de *deadlock*, em que os outros PLs aguardam que o PL_i avance seu relógio para que o gerenciador avance o horizonte, ao mesmo tempo que o PL_i aguarda eventos de outros PLs para avançar seu relógio. Para resolver o *deadlock*, o gerenciador determina o horizonte como sendo o menor tempo dos eventos a serem tratados dentre todas as filas de eventos. Como todos os PLs estão bloqueados, o evento de menor tempo é seguro.

O Algoritmo 2 representa os passos realizados por cada *thread* correspondente a um PL_i, enquanto as tarefas realizadas pelo gerenciador estão descritas no Algoritmo 3.

Algoritmo 2: Execução PL_i

```
while not fimSimulação do
    while háEventoSeguro(PL_i) do
        e := removeEvento(fila PL_i)
        relógio PL_i := e.tempo
        aplicação.trataEvento(PL_i, e)
    if not fimSimulação then
        bloqueiaPL(PL_i)
```

Algoritmo 3: Gerenciador

```
while not fimSimulação do
    avançaHorizonte()
    for each PL_i do
        if PLBloqueado(PL_i) then
            desbloqueiaPL(PL_i)
    fimSimulação := checaFimSimulação()
    if not fimSimulação then
        hibernaGerenciador(tempoSuspensão)
```

O simulador possui variáveis de controle como o horizonte, que são modificadas pelo gerenciador e lidas pelas *threads* que executam os PLs. Cada PL pode inserir eventos na fila de eventos de um outro PL, utilizando um *lock* na operação de manipulação da fila para garantir a exclusão mútua. Outras estruturas são de acesso exclusivo de cada *thread* para minimizar o *overhead* de sincronização.

O fim da simulação é detectado quando todos os PLs estão bloqueados e todas as filas de eventos estão vazias, portanto não existe nenhum evento para ser tratado, assim como é definido no simulador sequencial. O que distingue a situação de *deadlock* é que todos os PLs estão bloqueados, porém ainda há eventos a serem tratados.

2.3. Aplicação

Para teste e avaliação dos simuladores, uma aplicação foi modelada para ser simulada. A aplicação escolhida é uma rede de filas, em que cada nó da rede é composto por um servidor e sua fila de clientes, onde clientes aguardam em filas e são atendidos por diferentes servidores. Assim, cada servidor e sua respectiva fila de clientes é modelado como um PL do sistema. Esse modelo é simples de ser implementado, porém ao mesmo tempo é poderoso o suficiente porque com mínimas mudanças pode representar sistemas físicos complexos [Liu 2009]. A Figura 2 ilustra uma rede de filas, em que o cliente de um servidor é enviado para a fila de clientes do próximo servidor.

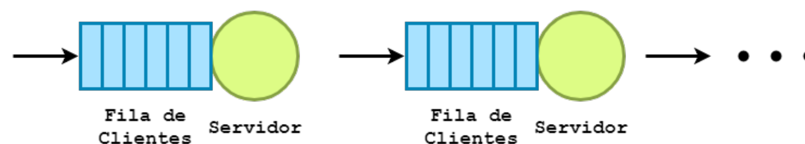

Figura 2. Sistema de rede de filas de servidores

Nessa aplicação existem somente dois tipos de eventos, Chegada de Cliente (CC) e Fim de Serviço (FS). A chegada de cliente em um PL_i no instante t é tratada da seguinte maneira: se o servidor estiver ocupado com um atendimento, o cliente é inserido no final da fila de clientes. Caso contrário, o cliente é atendido imediatamente e o PL_i gera um evento de FS para ele mesmo com tempo $t +$ *tempo de serviço*. Ao final, o PL_i gera um outro evento de CC para ele mesmo com tempo $t +$ *tempo de interchegada*, alimentando a rede de filas com clientes até certo instante final predefinido pela aplicação.

O tratamento de um evento de fim de serviço em um PL_i no instante t consiste em: o PL_i gera um evento de CC com tempo t para outro PL_j, enviando o cliente que

terminou o atendimento para um próximo servidor de acordo com a topologia da rede. Se houver algum cliente esperando na fila de clientes do PL_i, ele é retirado da fila e atendido. Nesse caso, o PL_i gera um evento de FS para ele mesmo com tempo $t + \textit{tempo de serviço}$.

Foi desenvolvido um gerador de redes de filas, que produz a entrada fornecida aos simuladores com os parâmetros da aplicação, que são o tempo de serviço e tempo de interchegada de cada servidor, a topologia da rede dos servidores (especificando quais PLs podem gerar eventos para quais outros PLs), além do tempo máximo da simulação. Todos esses parâmetros são gerados aleatoriamente para cada rede de filas gerada, de maneira que podem ser simuladas diferentes redes, ou diferentes instâncias de uma mesma topologia de rede, o que permite avaliar os simuladores em diferentes cenários. Para permitir uma comparação precisa dos resultados dos simuladores paralelo e sequencial, a aleatoriedade restringe-se à etapa de geração da rede e não é aplicada durante a simulação.

3. Resultados e Discussão

Uma avaliação de desempenho foi realizada para comparar os simuladores sequencial e paralelo e avaliar otimizações propostas para eles. Diferentes topologias de redes de fila (linear, circular, completamente conexa e parcialmente conexa) foram simuladas. Os resultados apresentados referem-se à rede parcialmente conexa que tem sua estrutura gerada aleatoriamente e permite uma avaliação mais abrangente dos simuladores.

Na Seção 3.6 compara-se a melhor versão dos simuladores sequencial e paralelo. Para isso, foram realizadas cinco repetições de cada simulação, isto é, foram geradas aleatoriamente cinco redes com as mesmas configurações. Os resultados de *speedup* obtidos apresentam variação média de 0,43% e máxima de 2,02%. Assim, julgou-se suficiente realizar somente uma execução de cada simulação nos demais experimentos.

A plataforma de execução dos simuladores é composta por um processador Intel Core i5-9300H com quatro núcleos físicos com *hyper-threading*, permitindo a execução simultânea de oito *threads*, com frequência de *clock* 2,4GHz. Cada núcleo possui memórias *cache* L1I e L1D de 32 KB, e L2 de 256KB, compartilhando *cache* L3 de 8MB e memória RAM de 8GB. Foi utilizado o Intel® C++ Compiler 19.1 no sistema operacional Windows 10 Home de 64 bits.

3.1. Estrutura de Dados para a Fila de Eventos

Experimentos preliminares mostraram que a inserção e remoção de eventos da fila de eventos, única e simplesmente encadeada, do simulador sequencial representa em média 99,6% do tempo total de execução. Portanto, foram analisadas quatro diferentes estruturas de dados para a fila de eventos, todas ordenadas pelo tempo do evento:

- Uma única fila simplesmente encadeada com inserção a partir do início (inicial).
- Uma única fila duplamente encadeada com inserção a partir do final.
- Múltiplas filas simplesmente encadeadas com inserção a partir do início.
- Múltiplas filas duplamente encadeadas com inserção a partir do final.

A Figura 3(a) apresenta o tempo de execução do simulador sequencial (em escala logarítmica) utilizando essas estruturas, para redes com diferentes número de PLs. A Figura 3(b) detalha o tempo gasto (em escala logarítmica) com as operações de inserção e remoção de eventos na execução do simulador sequencial com 128 PLs. Observou-se

que eventos novos são em geral inseridos da metade para o fim da fila, logo a inserção do evento percorre pelo menos metade dos elementos da fila. Assim, foi implementada a fila duplamente encadeada que inicia a inserção pelo fim. Essa mudança produz um tempo de execução médio 1,75 vezes pior que o obtido com a estrutura inicial.

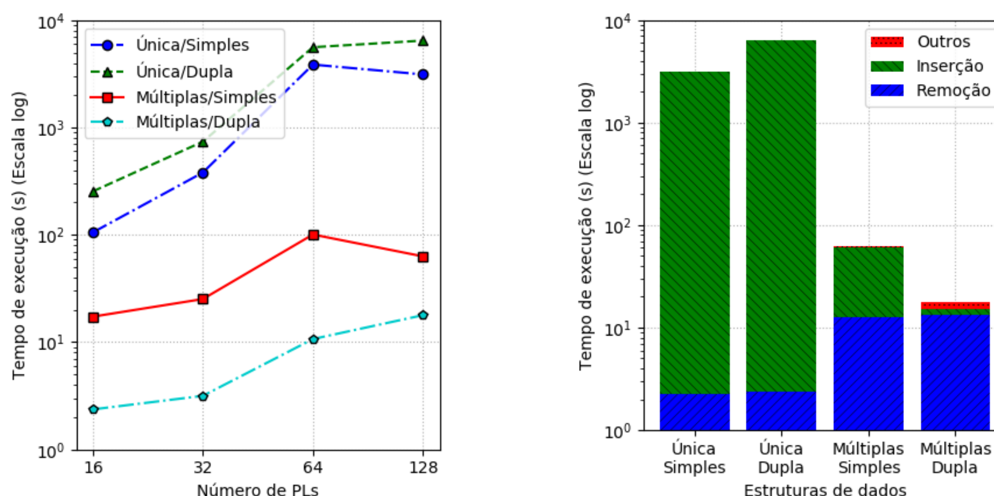

Figura 3. (a) Simulador sequencial com diferentes estruturas de dados para fila de eventos; (b) Tempo de execução das operações de inserção e remoção de eventos para 128 PLs

De forma análoga ao simulador paralelo, foram implementadas múltiplas filas no simulador sequencial, uma para cada PL. Com isso, o número de eventos em cada fila é menor, o que reduz o tempo de inserção de eventos. Essa estrutura proporciona, em média, um tempo de execução 36 vezes melhor que o inicial, mesmo com o maior tempo de remoção, justificado pela busca pelo evento de menor tempo dentre todas as filas. Por fim, quando combina-se múltiplas filas duplamente encadeadas, o melhor resultado é obtido, com o tempo de execução, em média, 219 vezes melhor que o da estrutura inicial. Essa estrutura é portanto a utilizada em todos os demais experimentos.

3.2. Mecanismo de Avanço do Horizonte

A versão inicial do simulador paralelo, denominada versão base, utiliza um único horizonte de eventos seguros para todos os PLs. Contudo, uma versão estendida foi desenvolvida, onde cada PL possui seu próprio horizonte. Nessa implementação, o simulador tem conhecimento da topologia da rede de PLs, portanto o gerenciador pode calcular o horizonte de cada PL_i tendo como base somente os relógios e os eventos dos PL_j que podem enviar eventos para o PL_i. A Figura 4 mostra o tempo de execução das duas versões do simulador paralelo, variando o número de PLs.

Em execuções com mais PLs e portanto mais *threads*, a versão base obteve menor tempo de execução. Devido ao alto custo de se calcular horizontes individuais para cada PL, a versão estendida obteve um desempenho cada vez pior, quase se equiparando à execução do melhor simulador sequencial da Seção 3.1.

3.3. Exploração de *Lookahead*

Para avançar mais o horizonte de eventos e permitir que mais eventos sejam tratados pela aplicação antes da *thread* do PL se bloquear, foi implementado o uso de *lookahead* nas

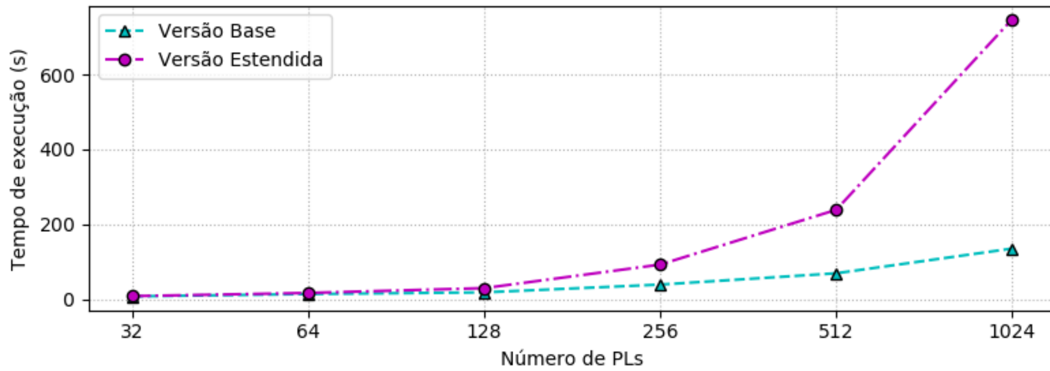

Figura 4. Simuladores paralelos com avanço do horizonte sem e com informação da topologia da rede

duas versões do simulador paralelo. A Figura 5 mostra o tempo de execução das duas versões do simulador paralelo, sem e com a exploração de *lookahead*, para redes com diferentes números de PLs.

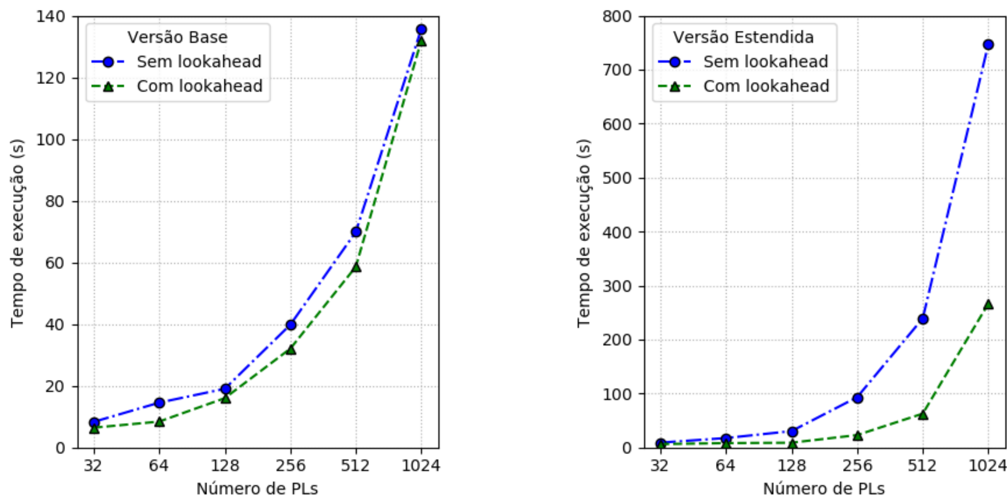

Figura 5. Impacto do uso de *lookahead* nas versões (a) base e (b) estendida do simulador paralelo

Como na versão base há somente um valor de horizonte, há também apenas um valor de *lookahead*. O *lookahead* é geralmente pequeno, sendo o menor tempo de tratamento de evento (tempo de serviço e tempo de interchegada) dentre todos os PLs, o que reduz levemente o tempo de execução, como mostra a Figura 5(a).

Na versão estendida, cada PL_i possui um horizonte e um *lookahead*$_i$, que é o menor tempo de tratamento de evento dos PL_j que podem enviar eventos para o PL_i. Isso resulta em uma redução significativa no tempo de execução, como mostra a Figura 5(b). Porém, esse tempo ainda é superior ao da versão base. Como o uso de *lookahead* se mostrou benéfico para as duas versões, nos demais experimentos os simuladores paralelos utilizam o *lookahead*.

3.4. Mecanismo de Suspensão do Gerenciador

O gerenciador representa um *overhead* na execução do simulador paralelo pois realiza a sincronização da execução das *threads*. A Figura 6 mostra o impacto do parâmetro de

hibernação H da Equação 1 no tempo de execução do melhor simulador paralelo, para diferentes números de PLs.

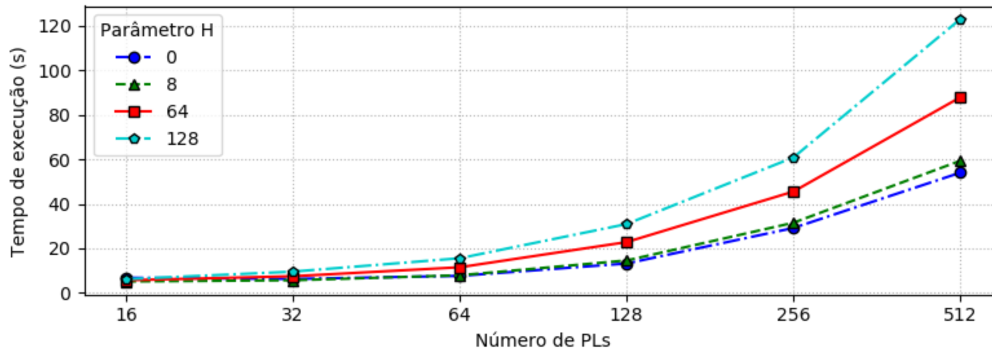

Figura 6. Simulador paralelo variando o parâmetro de hibernação H do gerenciador

O melhor tempo de execução é obtido para $H = 0$. Isso mostra que não é necessário forçar a *thread* gerenciadora a hibernar, possivelmente porque o tempo gasto em trocas de contexto no processador já é suficiente para a simulação ser executada sem atrasos. Também mostra como o avanço da simulação é dependente da sincronização provida pelo gerenciador, porque quanto menos ele está ativo, maior foi o tempo total de execução, indicando que muitas *threads* estavam bloqueadas por um longo período de tempo. Portanto, para os demais experimentos realizados, o parâmetro H tem valor 0 e não há suspensão forçada do gerenciador.

3.5. Carga de Trabalho da Aplicação

O tratamento dos eventos da aplicação de rede de filas desenvolvida é pouco custoso, pois cada evento calcula apenas o tempo e destino do próximo evento. Para mostrar como a execução paralela dos PLs traz ganhos significativos no tempo de execução, foi analisado um parâmetro de Carga de Trabalho, em que a rotina de tratamento de evento deixa a *thread* do PL em espera ocupada para emular aplicações de sistemas físicos complexos.

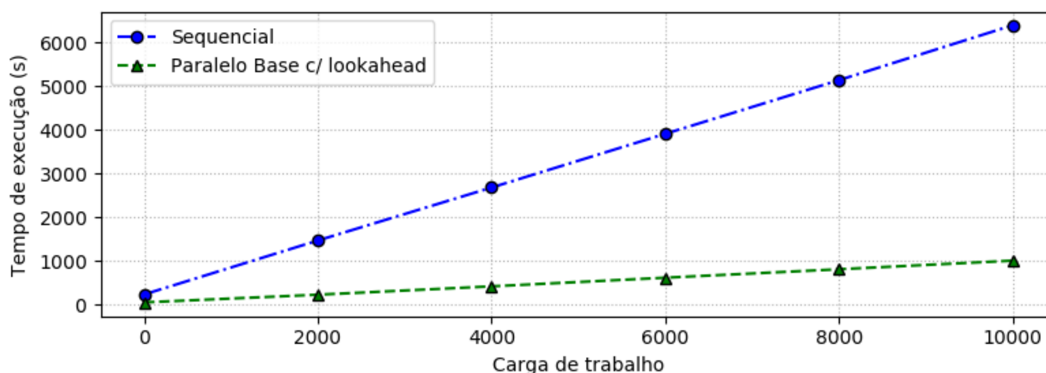

Figura 7. Simuladores sequencial e paralelo com variação da carga de trabalho para 512 PLs

A Figura 7 mostra o tempo de execução dos melhores simuladores sequencial e paralelo, para 512 PLs, variando a carga de trabalho da aplicação. O tempo de execução

cresce muito mais rápido com o simulador sequencial, expondo como a execução concorrente dos PLs é benéfica para a simulação de sistemas físicos complexos, onde cada evento deve atualizar várias variáveis de estado com operações custosas, como por exemplo aplicação de transformadas integrais e manipulação de matrizes.

Quando comparada à carga de trabalho 0, a carga de valor 100 causa aumento no tempo de execução de 75s e 5s, no simulador sequencial e paralelo, respectivamente. Para representar situações reais, em que a aplicação realiza várias operações ao tratar cada evento, esse valor foi escolhido como carga de trabalho para os demais experimentos.

3.6. Desempenho e Consumo de Memória dos Simuladores Sequencial e Paralelos

Compara-se aqui o melhor simulador sequencial (com múltiplas filas duplamente encadeadas) e as versões base e estendida do simulador paralelo, sem e com o uso de *lookahead*. A Figura 8(a) mostra o tempo de execução dos simuladores variando o número de PLs. A Figura 8(b) exibe o consumo máximo de memória na execução do melhor simulador sequencial e melhor paralelo variando o número de PLs.

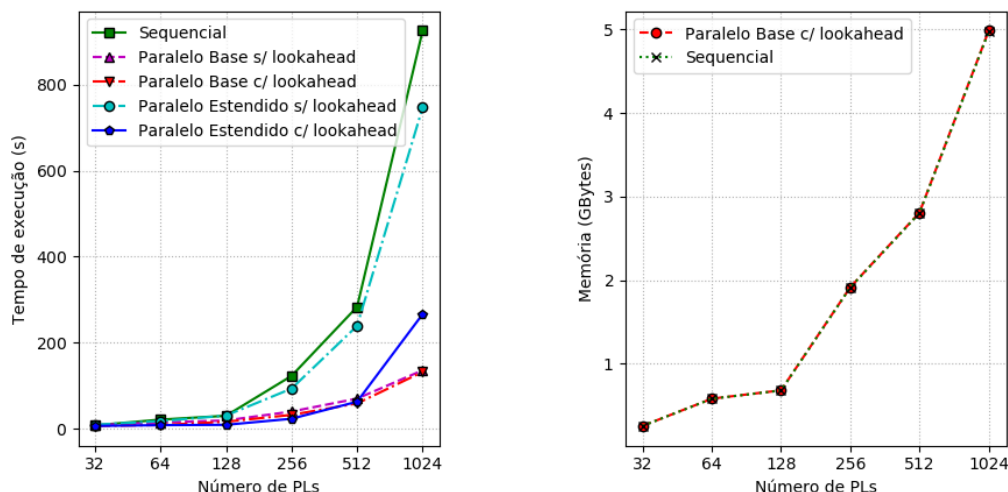

Figura 8. (a) Tempo de execução dos vários simuladores; (b) Consumo máximo de memória dos simuladores

O simulador sequencial apresenta os maiores tempos de execução, porém o simulador paralelo versão estendida sem *lookahead* apresenta um comportamento similar. Isso mostra que não justifica ter o *overhead* de manter um horizonte para cada PL, se o *lookahead* não é explorado. A versão base do simulador paralelo produz os menores tempos de execução. A diferença no consumo de memória entre os simuladores sequencial e paralelos não é significativa. Com um maior número de PLs, mais eventos são gerados e as filas de eventos crescem, causando o aumento do consumo de memória.

O melhor simulador paralelo é a versão base com somente um horizonte e uso do *lookahead*. A Figura 9 mostra o *speedup* obtido por esse simulador em relação ao melhor simulador sequencial, variando o número de PLs.

A partir de 8 PLs já há ganho de desempenho, que aumenta à medida que o número de PLs cresce. O *speedup* máximo de 7,01 é obtido para 1024 PLs. O *speedup* ideal de 8, não é alcançado pois os mecanismos de sincronização representam um *overhead* do simulador paralelo. Os vales para 32 e 128 PLs são resultado da aleatoriedade na geração das redes de filas usadas na entrada dos simuladores.

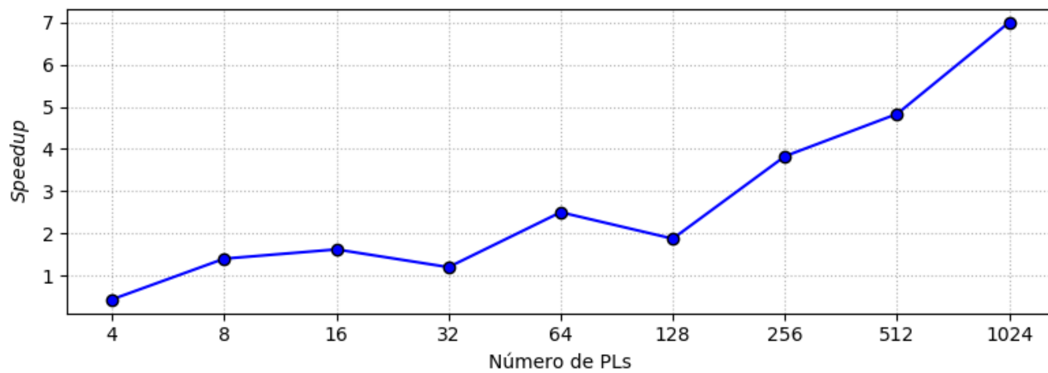

Figura 9. *Speedup* do melhor simulador paralelo em relação ao melhor sequencial

3.7. Escalabilidade do Simulador Paralelo

Para avaliar a escalabilidade do simulador paralelo, foi controlado o número máximo de núcleos virtuais disponíveis para a execução utilizando ferramentas do compilador. As Figuras 10(a) e (b) mostram, respectivamente, o tempo de execução e o *speedup* (em relação ao melhor sequencial) do melhor simulador paralelo, variando o número de núcleos usados, para 1024 PLs.

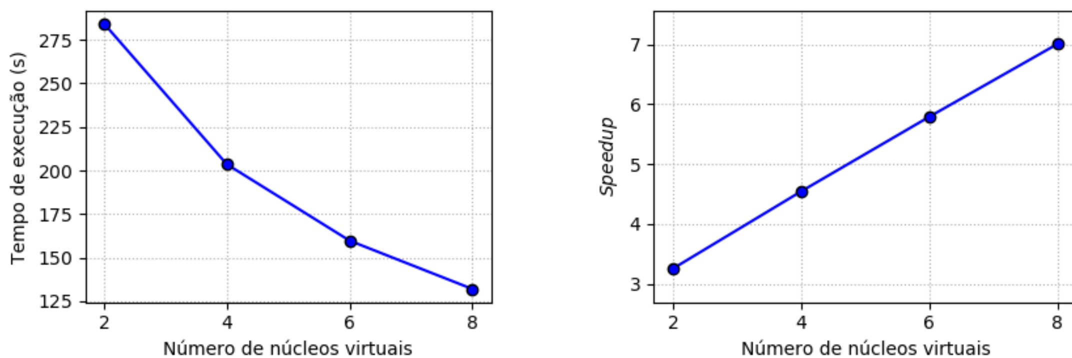

Figura 10. (a) Tempo de execução e (b) *speedup* (em relação ao simulador sequencial) do melhor simulador paralelo, ao variar número de núcleos, para 1024 PLs

A Figura 10(a) reforça a redução do tempo de execução do simulador paralelo quando mais núcleos são usados, isto é, mais paralelismo é explorado. A Figura 10(b) mostra que o *speedup* cresce quase que linearmente com o número de núcleos, representando uma ótima escalabilidade. O *speedup* superlinear de 4,54 para 4 núcleos é causado, provavelmente, pelo maior espaço disponível em memórias *cache*, quando mais núcleos são usados [Ristov et al. 2016].

4. Conclusões

O simulador paralelo desenvolvido nesse trabalho permite o estudo do comportamento de diferentes sistemas físicos complexos. Basta que a aplicação seja modelada como um conjunto de PLs que interagem entre si através de eventos. Foram analisadas questões como estrutura de dados adotada para a fila de eventos, avanço do horizonte da simulação, exploração de *lookahead*, suspensão do mecanismo gerenciador e impacto da carga de trabalho da aplicação.

Como o processador utilizado na avaliação de desempenho possui quatro núcleos e pode executar oito *threads* simultaneamente, o *speedup* máximo obtido de 7,01 é extremamente positivo e evidencia que a paralelização da simulação de eventos discretos em sistemas de memória compartilhada é eficaz e explora a capacidade de execução concorrente dos PLs. O simulador apresenta escalabilidade em relação ao aumento dos núcleos disponíveis e da complexidade do sistema físico modelado.

Como trabalhos futuros, o uso de uma estrutura de árvore balanceada ou *heap* para a fila de eventos tem potencial de melhorar o desempenho dos simuladores. Outra possibilidade é definir o número total de *threads* de acordo com o número de núcleos, com cada *thread* responsável pela simulação de vários PLs, diminuindo o *overhead* da troca de contexto das *threads* no processador. A investigação de mecanismos otimistas de simulação paralela em sistemas de memória compartilhada também seria interessante.

Referências

Ahmed, M. G., Pattalwar, S. V., and Thakare, V. M. (2012). Parallel discrete event simulation based on multi-core platform – a new approach. *International Journal of Computer Applications – Proceedings of the National Conference on Innovative Paradigms in Engineering and Technology*, NCIPET(11):12–16.

Banks, J. et al. (2010). *Discrete-event system simulation*. Prentice Hall, 5 edition.

Chandy, K. M. and Misra, J. (1981). Asynchronous distributed simulation via a sequence of parallel computations. *Communications of the ACM*, 24(4):198–206.

De Munck, S., Vanmechelen, K., and Broeckhove, J. (2010). Design and performance evaluation of a conservative parallel discrete event core for GES. In *Proceedings of the International Conference on Simulation Tools and Techniques*, pages 1–10.

Fujimoto, R. M. (2000). *Parallel and distributed simulation systems*. John Wiley & Sons.

Fujimoto, R. M. (2015). Parallel and distributed simulation. In *Proceedings of the Winter Simulation Conference*, pages 45–59.

Jefferson, D. R. (1985). Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425.

Law, A. M. and Kelton, W. D. (2015). *Simulation modeling and analysis*. McGraw-Hill.

Liu, J. (2009). Parallel discrete-event simulation. Technical report, School of Computing and Information Sciences, Florida International University.

Misra, J. (1986). Distributed discrete-event simulation. *ACM Computing Surveys*, 18(1):39–65.

OpenMP Architecture Review Board (2020). The OpenMP API specification for parallel programming. `https://www.openmp.org/`. (accessado em janeiro 2020).

Ristov, S., Prodan, R., Gusev, M., and Skala, K. (2016). Superlinear speedup in HPC systems: why and when? In *Proceedings of the Federated Conference on Computer Science and Information Systems*, pages 889–898.

Sokolowski, J. and Banks, C., editors (2010). *Modeling and simulation fundamentals*. John Wiley & Sons.

Wagner, D. B., Lazowska, E. D., and Bershad, B. N. (1989). Techniques for efficient shared-memory parallel simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation*, pages 29–37.