Arthur Francisco Lorenzon

Um Estudo sobre o Desenvolvimento de Aplicações com Criação Dinâmica de Processos em MPI

Alegrete – RS

2013

Arthur Francisco Lorenzon

Um Estudo sobre o Desenvolvimento de Aplicações com Criação Dinâmica de Processos em MPI

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação da Universidade Federal do Pampa como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Orientador: Márcia Cristina Cera

Coorientador: Fábio Diniz Rossi

Universidade Federal do Pampa – UNIPAMPA

Campus Alegrete

Curso de Graduação em Ciência da Computação

Alegrete – RS

2013

Arthur Francisco Lorenzon

Um Estudo sobre o Desenvolvimento de Aplicações com Criação Dinâmica de Processos em MPI

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação da Universidade Federal do Pampa como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Trabalho de Conclusão de Curso defendido e aprovado em 04 de março de 2013

Márcia C. Cera

Márcia Cristina Cera
Orientador

Fábio Rossi

Fábio Diniz Rossi
Coorientador

Claudio Schepke

Claudio Schepke
Convidado 1

Ewerson Carvalho

Ewerson Carvalho
Convidado 2

Alegrete – RS
2013

Dedico este trabalho à Deus, minha família,
meus orientadores e amigos.

Agradecimentos

Primeiramente, gostaria de agradecer a Deus por mais esta conquista.

Agradeço de forma especial, à minha orientadora Prof. Márcia Cristina Cera que além de ser uma excelente mestre, foi uma grande amiga, permitindo que com seus conselhos, pudesse avançar na vida acadêmica. Obrigado por acreditar em mim e se preocupar com a evolução de seus alunos. Tenho certeza que colherá muitos frutos.

Agradeço de forma especial, ao meu coorientador Prof. Fábio Diniz Rossi, que acompanha minha trajetória acadêmica desde o início. Saiba que a forma com que conduziste teus ensinamentos, aliados ao grande amigo que tu és, foi essencial para que eu pudesse estar aqui hoje. Obrigado por confiar e acreditar em mim desde o início.

Agradeço a Universidade Federal do Pampa (UNIPAMPA), ao corpo docente do curso de Ciência da Computação, e em especial àqueles que contribuíram para minha formação. Estendo meus agradecimentos à todos os que mantive contato frequente, em especial ao GEMPP, que compartilhou saberes em inúmeras tardes de programação paralela e aos vigilantes, sempre prontos para uma térmica de café ou chimarrão.

Agradeço a Banca Examinadora, Claudio Schepke e Ewerson Carvalho pelas contribuições no desenvolvimento deste trabalho.

Agradeço, de forma especial, aos meus colegas Brandon, Jaline, Henrique, Sander e Thiarles por terem sido muito mais que colegas durante esta etapa. Formamos um grupo (família) em que a amizade, o respeito e a sabedoria de todos foram o combustível para muitas de nossas conquistas.

Agradeço aos meus amigos que conheci no decorrer desta etapa que se finaliza. Ao meu amigo Ademir, que me auxiliou nos momentos em que estava me estabelecendo em Alegrete. Aos meus amigos que dividiram a mesma morada, compartilhando saberes e contribuindo para minha formação, Alessandro, Uillian e Maicon. Ao meu amigo Jean e à sua família, que apesar do pouco tempo de convívio se tornaram pessoas as quais possuo grande respeito e admiração. Por fim, estendo meu agradecimento à todos meus amigos que contribuíram para a minha formação.

Por último, mas não menos importante, agradeço a minha família (Osvaldo, Senilda, Allyne, Jeferson e Matheus) por todo o apoio incondicional prestado durante esta etapa. Vocês foram de extrema importância, fornecendo o suporte necessário nos momentos de dificuldade. Peço desculpas pelas ausências nos momentos importantes.

"Disciplina é a ponte que liga
nossos sonhos às nossas realizações."
(Pat Tillman)

Resumo

O MPI é o padrão amplamente utilizado no desenvolvimento de programas paralelos para ambientes de memória distribuída. Ele é dividido em duas normas: MPI-1 e MPI-2. A primeira caracteriza-se por criar processos estaticamente, no início da execução da aplicação, enquanto que na segunda, a criação de processos ocorre de forma dinâmica, ou seja, em tempo de execução. Entretanto, criar processos dinamicamente gera um custo extra para a aplicação, quando comparado com a criação estática. Este trabalho realiza um estudo sobre o desenvolvimento de aplicações com criação dinâmica de processos, que objetiva investigar o impacto deste sobre-custo em aplicações MPI-2 e como ele pode ser contornado através do uso de técnicas de programação eficientes e características avançadas fornecidas pelo próprio MPI. Para isto, foram implementadas aplicações em MPI-1 e MPI-2, sob dois problemas (Jogo da Vida e *Skyline Matrix Solver*) da suíte *Cowichan Problems*. Nossa primeira análise, realizada sobre o Jogo da Vida, foi possível encontrar o custo extra da criação dinâmica de processos em aplicações com carga regular de trabalho e cujo tempo de computação é similar entre os processos. Já a segunda análise efetuada sobre o *Skyline Matrix Solver* possibilitou analisar o impacto da criação de processos em tempo de execução nas aplicações que possuem carga de trabalho irregular e tempo de computação diferente entre os processos. Nossos resultados mostram que para aplicações com carga de trabalho regular e com pouco tempo útil de computação (similares ao Jogo da Vida), no melhor dos casos, a inserção da criação dinâmica de processos impactou em apenas $0,2\%$ no tempo total da aplicação. Já para aplicações com carga de trabalho irregular e com tempo consideravelmente útil de computação (similares ao *Skyline Matrix Solver*), mostramos que é possível obter ganho com a criação dinâmica de processos, em que no melhor dos casos, a implementação com MPI-2 foi 6% mais eficiente que a implementação MPI-1.

Palavras-chave: *Message-Passing Interface*. Criação Dinâmica de Processos, Programação Paralela Distribuída.

Abstract

The MPI standard is widely used in the development of parallel programs for distributed memory environments. It is divided into two standards: MPI-1 and MPI-2. The first is characterized by creating processes statically at the start of execution of the application, while in the second the creation of a process takes place dynamically, ie at runtime. However, processes create dynamically generates an extra cost to implement, when compared with static setting. This paper makes a study on the development of applications with dynamic creation of processes, which aims to investigate the impact of this overhead in MPI-2 applications and how it can be bypassed through the use of efficient programming techniques and advanced features provided by the MPI. For this, applications have been implemented with MPI-1 and MPI-2, under two problems (Game of Life and Skyline Matrix Solver) of the suite Cowichan Problems. Our first analysis performed on the Game of Life, it was possible to find the overhead of the dynamic creation of processes in applications with regular workload and whose computation time is similar in both cases. The second analysis performed on the Skyline Matrix Solver possible to analyze the impact of process creation at runtime in applications that have irregular workload and different computation time between processes. Our results show that for applications with regular workload and with little computing time (similar to the Game of Life), at best, the inclusion of dynamic creation of processes impacted by just $0,2\%$ in the total time of application . For applications with irregular work load considerably useful and time of computation (similar to Skyline Matrix Solver), we show that it is possible to succeed with the creation of dynamic processes, wherein at best, with the implementation of MPI-2 was 6% more efficient than the MPI-1 implementation version.

Key-words: Message-Passing Interface. Creating Dynamic Processes. Distributed Parallel Programming.

Lista de ilustrações

Lista de tabelas

Sumário

1 Introdução

De um modo geral, a programação paralela consiste na divisão de tarefas de uma aplicação com a finalidade de reduzir seu tempo total de execução (RAUBER; RÜNGER, 2010). Ela tem sido muito utilizada no desenvolvimento de aplicações científicas que necessitam grande poder computacional, como por exemplo, o cálculo da previsão do tempo. Em computadores convencionais (com um único processador), pode-se demorar dias para calcular a previsão do tempo para o dia seguinte. Desta forma, quando o resultado estiver pronto, ele não será mais válido. Já com o uso da programação paralela e distribuída, onde utiliza-se diferentes computadores trabalhando em conjunto para efetuar o cálculo da previsão do tempo, consegue-se reduzir este tempo para poucas horas.

A programação paralela em ambientes onde a memória é distribuída entre os computadores se dá através de padrões de troca de mensagens. MPI (*Message-Passing Interface*) (GROPP et al., 1998) é a biblioteca padrão para a transmissão de mensagens em arquiteturas paralelas de memória distribuída. Este padrão foi definido e é atualizado pelo *Message-Passing Interface Forum*. Em uma de suas atualizações, chamada de MPI-2 (GROPP et al., 2003), o padrão passou a definir a criação dinâmica de processos (ou seja, a criação de novos processos em tempo de execução), o acesso remoto a memória (RMA –*Remote Memory Access*) e a paralelização da entrada e saída de dados entre outras características.

Este trabalho realiza um estudo sobre o desenvolvimento de aplicações com criação dinâmica de processos em MPI. Esta característica possibilita a implementação de aplicações MPI-2 flexíveis ou adaptativas, ou seja, aplicações que possam se adaptar à eventuais mudanças da arquitetura em tempo de execução (i.e. exclusão ou inclusão de novos computadores). Tais aplicações podem ser usadas, por exemplo, para aumentar a taxa de utilização dos recursos de *clusters* de computadores. Assim, nós computacionais que estejam ociosos podem ser utilizados para computar aplicações científicas (CERA et al., 2010). Outra possibilidade é oferecer meios para a execução de aplicações MPI no contexto das Nuvens Computacionais. Por exemplo, o *framework Nebulous* (GALANTE; BONA, 2011), em fase de desenvolvimento, objetiva facilitar o processo de desenvolvimento e execução de aplicações científicas MPI em Nuvens e trata da flexibilização das aplicações para a execução neste tipo de plataforma.

Entretanto, criar processos em tempo de execução traz custos adicionais à aplicação. Isto porque a criação dinâmica acontece de forma síncrona e leva ao estabelecimento de um modelo de comunicação hierárquico entre os processos criados e seus criadores.

No entanto, o próprio padrão MPI oferece meios para reduzir tais perdas, permitindo sobrepor comunicação com computação útil.

Neste contexto, este trabalho utilizará dois problemas da suíte *Cowichan Problems* para estudar o uso da criação dinâmica de processos. Os *Cowichan Problems* são utilizados por diversos autores para a análise de sistemas de programação paralela (como por exemplo, Wilson e Bal (2007), Anvik et al. (2005) e Paudel e Amaral (2011)), isto porque seus problemas possibilitam analisar diferentes características destes sistemas. Portanto, optamos pela utilização de dois problemas que possuem características diferentes: o Jogo da Vida, com carga de trabalho regular e tempo de computação similar entre os processos; e o *Skyline Matrix Solver*, com carga de trabalho irregular e tempo de computação diferente entre os processos.

Este trabalho permitirá determinar o impacto da criação dinâmica de processos em aplicações que possuem carga regular e irregular de trabalho. Adicionalmente, será refletido sobre os resultados obtidos, a fim de determinar se o desafio que envolve a implementação de aplicações com criação de processos em tempo de execução é compensado pelo ganho obtido em questões de desempenho da aplicação.

1.1 Objetivo

O presente trabalho tem como objetivo principal estudar o desenvolvimento de aplicações com criação dinâmica de processos através da norma MPI-2. Para isto, foi escolhido um grupo de problemas-alvo da suíte *Cowichan Problems*. Com o intuito de alcançar o objetivo principal, definimos alguns objetivos específicos:

1. Comparar o tempo de processamento entre as implementações MPI-1 e MPI-2;
2. Identificar o impacto da criação dinâmica de processos em aplicações que possuem carga regular e irregular de trabalho;
3. Identificar o impacto causado pelas comunicações que ocorrem em uma organização hierárquica de comunicadores.
4. Explorar o uso das características avançadas do MPI, com a finalidade de propor alternativas para compensar o custo extra da criação dos processos em tempo de execução.

1.2 Estrutura do Texto

Este trabalho está estruturado da seguinte forma: O Capítulo 2 apresenta os conceitos da programação paralela em sistemas de memória distribuída. Para isto, apre-

sentamos a classificação das arquiteturas paralelas, com ênfase nos Multicomputadores e a forma de programar paralelo neste tipo de arquitetura. Também é apresentado o padrão MPI, destacando as principais características e diferenças das normas MPI-1 e MPI-2. A suíte de problemas *Cowichan Problems* também é descrita neste Capítulo, em que apresentamos os dois problemas-alvo. Por fim, será contextualizado sobre os trabalhos que fazem uso da criação dinâmica de processos.

O estudo do desenvolvimento de aplicações com criação dinâmica de processos está dividido em duas partes. A primeira parte contempla o estudo e análise das implementações paralelas do Jogo da Vida. Desta forma, o Capítulo 3 apresenta o Jogo da Vida e discute o desenvolvimento das aplicações paralelas com MPI-1 e MPI-2. Por fim, são apresentados os resultados e conclusões sobre a análise realizada sobre este problema.

Na segunda parte, o Capítulo 4 apresenta o estudo e análise das implementações paralelas do *Skyline Matrix Solver*. Assim, inicialmente é contextualizado sobre o problema, em que são destacadas as principais características do problema e desafios encontrados na paralelização deste problema em sistemas de memória distribuída. Após, são discutidas as implementações paralelas envolvendo as normas MPI-1 e MPI-2 e apresentados os resultados obtidos.

Por fim, o Capítulo 5 discute os resultados obtidos nas análises sobre o Jogo da Vida e o *Skyline Matrix Solver*, seguido das referências bibliográficas utilizadas no desenvolvimento deste trabalho.

2 Referencial Teórico

Este capítulo objetiva contextualizar aspectos da programação paralela em sistemas de memória distribuída e caracterizar a suíte de problemas *Cowichan Problems*. Para isto, a Seção 2.1 contém a classificação das arquiteturas paralelas de acordo com suas características principais (fluxo de instruções/dados e organização de memória). Já na Seção 2.2 serão abordadas as características fundamentais da programação paralela distribuída utilizando as normas MPI-1 e MPI-2. A Seção 2.3 apresenta a suíte de problemas *Cowichan Problems* e, por fim, a Seção 2.4 contém os trabalhos relacionados.

2.1 Classificação das Arquiteturas Paralelas

Os computadores paralelos surgiram após um intenso processo evolutivo com o objetivo de ampliar o número de problemas que podem ser resolvidos de maneira eficiente, além de proporcionar o aumento de desempenho das aplicações. Estes computadores são caracterizados como uma coleção de elementos de processamento que podem comunicar-se e cooperar para resolver grandes problemas em tempo hábil (ALMASI; GOTTLIEB, 1994).

Conforme Aspray (1990), são três os principais componentes de um computador: a **Unidade de Controle**, que gerencia o fluxo de dados e coordena todas as operações do computador, ou seja, informa à Unidade Funcional qual operação deverá ser executada; a **Unidade Funcional**, que executará determinadas instruções sobre os dados, as quais poderão ser operações aritméticas e/ou lógicas; e por fim, a **Memória**, que armazena internamente toda a informação (conjunto de instruções e os dados) que é manipulada pelo computador. Devido as diversas formas com que estes componentes podem estar organizados, faz-se necessária a classificação dos computadores com a finalidade de uniformizar de maneira coerente as características dos diferentes sistemas computacionais.

De acordo com Rauber e Rünger (2010), eles são classificados através de suas características principais, e um modelo de classificação simples e bastante difundido é a Taxonomia de *Flynn* (FLYNN, 1972). Ela caracteriza os computadores de acordo com o fluxo de instruções e na forma de como os dados são tratados. Assim, *Flynn* organizou os computadores em quatro classes, conforme apresentado a seguir:

***Single Instruction, Single-Data* (SISD):** Este grupo representa os computadores que possuem uma única unidade de processamento, também chamados de *single-*

core. A Figura 1a ilustra esta classe de computadores onde tem-se uma única unidade de Controle e uma única unidade Funcional operando sobre uma posição da Memória. Nestas condições, esta arquitetura processa somente uma instrução e um fluxo de dados em cada momento. Um exemplo clássico desta classe é a Arquitetura de Von Neumann.

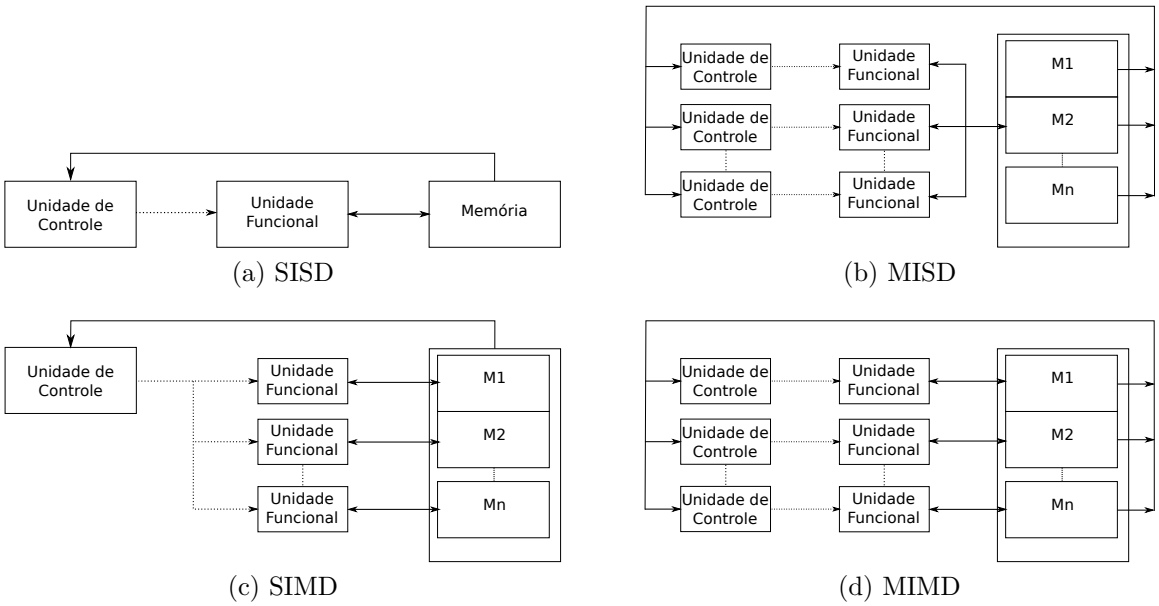
***Multiple-Instruction, Single-Data* (MISD):** Esta classe é representada pelos computadores que possuem múltiplas Unidades de Controle coordenando múltiplas Unidades Funcionais sobre o mesmo dado, conforme pode ser visto na Figura 1b. Tecnicamente, isto é impraticável, pois múltiplas unidades de processamento estariam operando sobre a mesma posição de memória ao mesmo tempo, executando instruções diferentes. Assim, não há nenhuma implementação deste tipo de arquitetura.

Os computadores paralelos concentram-se nas duas classes seguintes:

***Single-Instruction, Multiple-Data* (SIMD):** Esta classe é caracterizada pela execução de um único fluxo de instruções ao mesmo tempo sobre múltiplos dados. Devido possuir uma única Unidade de Controle, conforme pode ser visualizado na Figura 1c, todos os processadores executam o mesmo programa em paralelo sobre diferentes fluxos de dados. Compõe esta classe os processadores vetoriais.

***Multiple-Instruction, Multiple-Data* (MIMD):** Esta classe é caracterizada pela execução simultânea de múltiplos fluxos de instruções, dos quais, cada um opera sobre seus dados de forma síncrona. Isto é possível pois para cada Unidade Funcional

Figura 1 – Ilustração da classificação proposta por Flynn



(a) SISD (b) MISD

(c) SIMD (d) MIMD

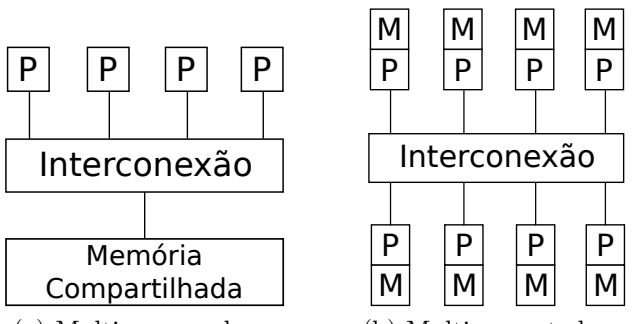
existe uma Unidade de Controle e ambas acessam posições diferentes na Memória, conforme ilustrado na Figura 1d. Nela, várias unidades de processamento operam de forma cooperativa e/ou concorrente na execução de uma ou várias aplicações. Esta classe representa as arquiteturas paralelas atuais, enquadrando-se nela, os processadores *multi-core* e os agregados de computadores dedicados (ex: *clusters* de alto desempenho).

A classificação proposta por *Flynn* é amplamente difundida, porém genérica. Portanto, os computadores paralelos que estão enquadrados na classe MIMD, podem ainda, conforme proposto em Tanenbaum (1976), ser classificados de acordo com a organização física da memória. Esta classificação leva em consideração a forma em que o acesso à memória entre os processadores está organizado e divide-os em Multiprocessadores e Multicomputadores.

Computadores onde todos os processadores compartilham uma memória central são chamados de **Multiprocessadores**. A Figura 2a ilustra este tipo de arquitetura onde os processadores (representados por P) estão conectados a uma memória global através de uma rede de interconexão. Nesta arquitetura, o paradigma de comunicação é o de memória compartilhada, onde a troca de dados ocorre através do uso de regiões de memória compartilhadas na memória global.

Os **Multicomputadores**, são assim chamados, pois consistem de um conjunto de computadores (onde cada computador é chamado de nó) interligados através de uma rede de interconexão (ex: *Ethernet* ou *InfiniBand*). Nesta arquitetura, ilustrada na Figura 2b, cada processador (representado por P) possui uma memória local privada (representada por M), à qual só ele tem acesso. Neste tipo de arquitetura, a troca de dados ocorre através da transmissão de mensagens. Este modelo de comunicação será apresentado com maiores detalhes na seção seguinte.

Figura 2 – Ilustração da arquitetura MIMD conforme organização da memória



(a) Multiprocessadores (b) Multicomputadores

2.2 Programação Paralela Distribuída

A execução de programas paralelos em ambientes onde o espaço de endereçamento é distribuído, ocorre através da troca de mensagens e da cooperação entre os processos. Isto é possível com o uso de primitivas de envio e recebimento. Vejamos o ambiente apresentado na Figura 2b. Nela, suponhamos que exista um processo p alocado em cada processador P. Quando o processo p_1 necessitar enviar um dado para o processo p_2, o processo p_1 realizará uma chamada a uma primitiva do tipo `send`, enquanto que p_2, faz uma chamada a uma primitiva do tipo `receive`. O inverso pode ocorrer caso o processo p_2 queira enviar um dado para p_1.

A operação de troca de mensagens no nível de programação é facilitada através do uso de padrões. O principal objetivo deles é oferecer portabilidade e proporcionar alto nível de abstração, simplificando assim, a tarefa de comunicação entre os processos. Tal simplicidade permite facilitar o desenvolvimento de aplicações paralelas para memória distribuída.

Dentre os padrões especificados, o MPI é amplamente utilizado em computadores paralelos, especialmente em Computadores Paralelos Escaláveis com memória distribuída e em Redes de Estações e Trabalho (SNIR et al., 1998). Outras bibliotecas surgem como alternativa ao MPI, como, por exemplo, o PVM (*Parallel Virtual Machine*) (GEIST et al., 1994). Entretanto, além da característica da criação dinâmica de processos, que é o foco deste trabalho, o MPI fornece ainda um nível de segurança e confiabilidade superior ao PVM nas rotinas de troca de mensagens (GEIST; KOHLA; PAPADOPOULOS, 1996).

O MPI é um padrão de fato e foi definido através da participação da comunidade de fabricantes e pesquisadores da área de Processamento de Alto Desempenho (PAD). Ele consiste numa interface de troca de mensagens e provê funções para linguagem C, C++ e sub-rotinas para Fortran-77 e Fortran-95. Um programa MPI é definido como uma coleção de processos que podem trocar mensagens (GROPP et al., 1998).

Ainda conforme Gropp et al. (1998), o MPI pode ser utilizado para implementação de aplicações SPMD (*Single Program, Multiple Data*), onde todos os processos executam partes diferentes do mesmo programa, selecionadas através dos identificadores únicos dos processos. Este estilo é o mais utilizado, embora também seja adequado ao estilo MPMD (*Multiple Program, Multiple Data*), onde cada processo segue um caminho de execução distinto através do mesmo código, ou códigos diferentes. De acordo com Wilkinson e Allen (2005) e Foster (1995), modelos de programação paralela podem ser utilizados para auxiliar no desenvolvimento de uma aplicação MPI. Diferentes modelos são definidos e, cada um, representa uma classe de algoritmos que possuem o mesmo tipo de controle.

2.2.1 Modelos de Programação Paralela

Um modelo muito utilizado por quem está começando a programar em paralelo é o modelo **Mestre/Trabalhador**. Ele é caracterizado por possuir um processo responsável (Mestre) capaz de comandar a execução dos Trabalhadores (demais processos), os quais recebem uma quantia de trabalho distribuída pelo Mestre e realizam a computação destes dados. Ao final da computação, retornam os resultados obtidos ao Mestre. Por ser um modelo onde a distribuição das tarefas é realizada de forma centralizada, geralmente, consegue-se obter um bom balanceamento de carga. Porém, em muitos casos, o Mestre se torna o gargalo da aplicação. Portanto, o programador deve tomar cuidado para não sobrecarregar o Mestre.

O modelo **Divisão e Conquista** é aconselhado à ser seguido em problemas que possam ser sub-divididos em pequenas frações de computação. Neste modelo, a Divisão é caracterizada pelo fracionamento recursivo do trabalho até que as frações tenham uma granularidade fina o suficiente para que possam ser computadas sequencialmente. A etapa da Conquista consiste em integrar os resultados obtidos pelo cálculo das pequenas frações. Este modelo é utilizado na paralelização dos algoritmos de ordenação *Quick Sort* (BROWN; XIONG, 1993) e *Merge Sort* (JEON; KIM, 2003), em que o montante de dados é dividido em pequenas frações de dados, os quais são distribuídos entre os processos/*threads*. Assim, cada processo/*thread*, realizará a ordenação destas pequenas frações e retornará estes dados ordenados na etapa da Conquista. Entretanto, o programador deve tomar cuidado ao realizar o balanceamento de carga na etapa da divisão de tarefas/dados, o que pode afetar de forma negativa na eficiência da aplicação, pois processos podem receber carga de trabalho maior que outros.

Nas aplicações onde diversas tarefas podem ser executadas em paralelo, o modelo de **Fases Paralelas** é o mais aconselhado. Ele é caracterizado por possuir duas etapas bem definidas e separadas: etapa da computação concorrente e outra da comunicação. Através do sincronismo na comunicação, o programador deve tomar cuidado com o balanceamento de carga de trabalho, pois processos podem ficar ociosos aguardando a etapa de comunicação caso concluam antes seu trabalho. Outro ponto que deve ser considerado, é o sobre-custo da comunicação, já que muitos processos vão se comunicar no mesmo período de tempo.

Diferente dos modelos apresentados, o ***Pipeline*** é caracterizado por dividir o problema em tarefas que devem ser completadas uma depois da outra. Aplicações que realizam diversas operações (tarefas) com a finalidade de melhorar a qualidade geral de uma imagem, utilizam este paradigma (SCHEPKE, 2009). Nelas, cada operação (tarefa) à ser realizada depende do resultado obtido pela operação (tarefa) anterior. Neste modelo, existe fluxo contínuo de dados, em que ocorre sobreposição de comunicação com a computação.

A escolha do paradigma à ser usado leva em consideração a característica da aplicação-alvo, da arquitetura em questão e as características da interface/biblioteca de programação paralela. Um programa MPI pode implementar os paradigmas descritos anteriormente e para isto, as duas subseções seguintes abordam conceitos de programação paralela utilizando o MPI-1 e MPI-2.

2.2.2 MPI-1

A concepção do padrão MPI envolveu um processo de padronização englobando um grupo de 60 pessoas de 40 organizações, principalmente dos Estados Unidos e da Europa (WALKER, 1993). Conforme Snir et al. (1998), nominou-se MPI-1 a primeira norma proposta, que específica operações de comunicação ponto a ponto (troca de dados entre dois processos) e coletiva (troca de dados envolvendo N processos), dentre outras características.

Um programa desenvolvido utilizando o padrão MPI-1 cria estaticamente todos os processos no início da sua execução, portanto a quantidade de processos não sofre alteração durante a execução do programa. Ao iniciar o programa, cada processo executa a função de inicialização do ambiente de execução MPI, o `MPI_Init()`. Já a finalização de um processo MPI ocorre através da chamada à função `MPI_Finalize()`. O Apêndice A apresenta informações detalhadas de todas as funções MPI utilizadas neste trabalho.

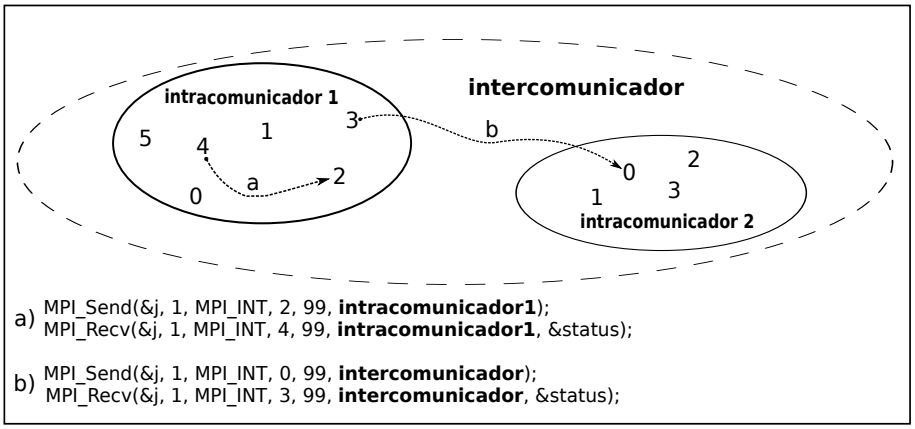
Após inicializar o ambiente de execução, cada processo MPI recebe o seu identificador (*rank*) dentro do comunicador através da função `MPI_Comm_rank()`. Um comunicador identifica um grupo de processos (conjunto ordenado de N processos) e representa os canais de comunicação por onde os dados são transmitidos entre os processos MPI. Por padrão, existe um comunicador pré-definido, denominado `MPI_COMM_WORLD`. O número total de processos de um determinado comunicador é obtido através da função `MPI_Comm_size()`.

Os comunicadores podem ser divididos em **intracomunicadores**, quando a comunicação é interna à um grupo de processos de um mesmo comunicador; e **intercomunicadores**, quando a comunicação é realizada entre processos envolvendo intracomunicadores diferentes. A Figura 3 ilustra a comunicação entre processos utilizando **intracomunicadores** (comunicação representada em (a)) e **intercomunicadores** (comunicações representada em (b)).

Ainda na Figura 3, a comunicação entre os processos ocorre através do uso de primitivas de envio/recebimento. Nela, são apresentados dois cenários (a e b) de comunicação ponto a ponto utilizando as primitivas `MPI_Send()` para envio e `MPI_Recv()` para recebimento.

O cenário a corresponde a comunicação ponto a ponto dentro do `intracomuni-`

Figura 3 – Comunicadores em MPI-1



Exemplo de comunicação entre a) processos internos a um intracomunicador (p_4 envia para p_2) e b) utilizando intercomunicador em intracomunicadores diferentes (p_3 do intracomunicador1 envia para p_0 do intracomunicador2)
Fonte: (MAILLARD; CERA, 2010)

`cador1`. Nele, o processo 4 envia um dado para o processo 2. Portanto, o processo 4 executará a função `MPI_Send()`, e o processo 2 a função `MPI_Recv()`. Já o cenário *b*, corresponde a comunicação entre dois intracomunicadores diferentes (`intracomunicador1` e `intracomunicador2`) através do `intercomunicador`. Nele, o processo 3 do `intracomunicador1` envia um dado através da função `MPI_Send()` para o processo 0 do `intracomunicador2`, que o recebe através da função `MPI_Recv()`.

Estas funções (utilizadas na Figura 3) realizam operações de comunicação do tipo bloqueante e exigem sincronismo entre os processos envolvidos, o que significa que a função irá ficar bloqueada até que todos os dados tenham sido enviados ou copiados para o *buffer* de memória. Após a operação estar completa, cada processo segue seu fluxo de execução. Entretanto, operações bloqueantes podem se tornar o gargalo da aplicação, já que mantém os processos bloqueados até a comunicação estar concluída. Esta limitação pode ser contornada através do uso de operações não-bloqueantes.

A operação não bloqueante (`MPI_Isend()`) retorna após a mensagem ser copiada para o *buffer* de envio. Desta forma, enquanto a mensagem está sendo transmitida, o processo pode realizar computação. Similarmente, a operação de recebimento não bloqueante (`MPI_Irecv()`) inicia a operação, mas não a completa. Neste caso, a função completará quando a mensagem estiver armazenada no *buffer* de recebimento. Enquanto isto, o processo pode computar sobre outros dados. Entretanto, em algum momento da execução, deve ser realizado uma verificação, a fim de saber se a operação está completa ou não. Assim, as funções `MPI_Wait()` e `MPI_Waitany()` podem ser utilizadas neste contexto, indicando que o processo só seguirá seu fluxo de execução quando concluir as comunicações iniciadas. As operações bloqueantes e não-bloqueantes podem ser utilizadas em conjunto (i.e. `MPI_Send()` e `MPI_Irecv()`), e a eficiência da aplicação em termos de comunicação

é decorrente do uso adequado destas funções.

As comunicações coletivas, são aquelas em que todos ou um conjunto de processos pertencentes a um comunicador participam. Exemplos de operações coletivas básicas: `MPI_Bcast()`, em que um processo envia dados para todos os processos do mesmo grupo; e operações de redução: `MPI_Reduce()`, na qual todos os processos enviam uma mensagem e somente um recebe, aplicando alguma operação sobre os dados recebidos, por exemplo, uma soma.

O MPI é um padrão que contém mais de 100 funções disponíveis, porém através da utilização das funções apresentadas acima, diversas aplicações podem ser eficientemente paralelizadas. Com o passar dos anos, através da participação da comunidade da área de Processamento de Alto Desempenho no MPI Forum[1], novas características foram adicionadas ao padrão, surgindo assim, a norma MPI-2.

2.2.3 MPI-2

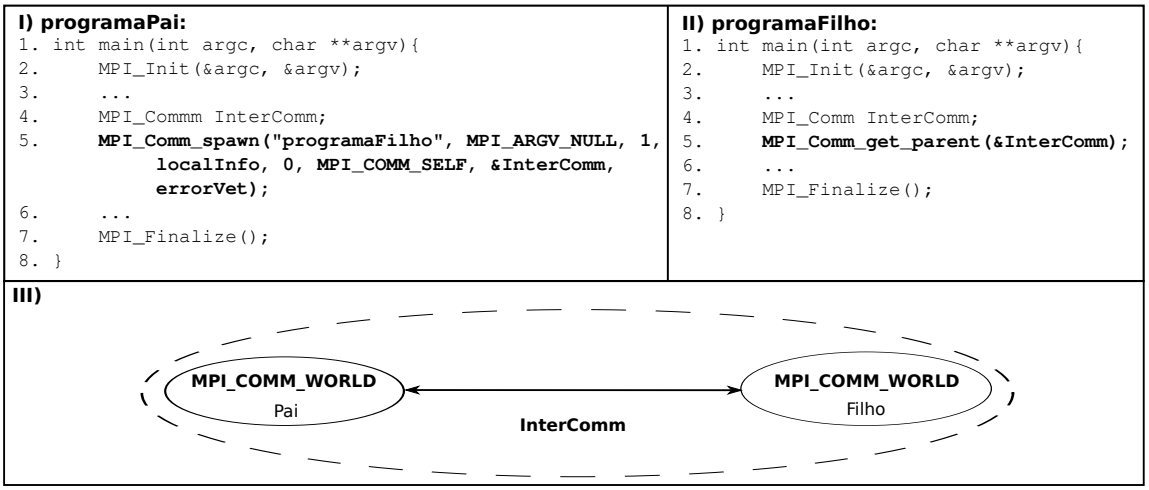
A norma MPI-2 define um conjunto de novos tópicos ao padrão MPI. Segundo Gropp et al. (2003), MPI-2 se diferencia do MPI-1 nas seguintes características: criação dinâmica de processos, operações de entrada e saída paralela, comunicações assimétricas e coletivas estendidas. No contexto deste trabalho, a mais importante delas é a criação dinâmica de processos. Esta característica adiciona dinamicidade à aplicação, possibilitando gerenciar (criar e/ou destruir) processos em tempo de execução.

Uma aplicação implementada com MPI-2 geralmente inicia a execução com um único processo. A criação dinâmica de processos é implementada através da primitiva `MPI_Comm_spawn()`. Ela deve ser invocada por um dos processos da aplicação MPI, o qual será chamado de pai. A invocação da primitiva leva a criação de um novo processo, chamado filho, o qual não precisa ser idêntico ao pai. Quando um processo filho é criado, ele irá pertencer a um intracomunicador diferente do pai e a comunicação entre eles ocorrerá através de um intercomunicador. No processo filho, o intercomunicador que o liga com o pai é retornado após a execução da função `MPI_Comm_get_parent()`. Já no processo pai, o intercomunicador que o liga ao filho é retornado na execução da função `MPI_Comm_spawn()`.

A Figura 4 exemplifica a criação de um processo. No retângulo I, é descrito o pseudocódigo denominado `programaPai` que será executado pelo processo pai. Nele, na linha 2 e 7 ocorre respectivamente a inicialização e finalização do ambiente MPI. Já na linha 5, ocorre a criação do processo filho que irá executar o programa chamado de `programaFilho`, que é definido no primeiro parâmetro. A quantidade de processos que serão criados é definida no terceiro parâmetro (neste caso, 1) e o sétimo indica o

[1]Disponível em: *http://www.mpi-forum.org/*

Figura 4 – Relacionamento hierárquico resultante da criação de um único processo



I) Pseudocódigo do programa executado pelo processo pai; **II)** Pseudocódigo do programa executado pelo processo filho; e **III)** Ilustração do relacionamento resultante entre pai e filho após a criação do processo filho.

intercomunicador que será utilizado para comunicações entre o pai e o(s) filho(s) (neste caso, `InterComm`). Os demais parâmetros são explicados no Apêndice A. Note que a inicialização das variáveis, bem como outras definições foram omitidas neste e nos demais pseudocódigos apresentados nesta Seção.

Ainda na Figura 4, porém no retângulo II, é apresentado o pseudocódigo do processo filho. Após a inicialização do ambiente MPI, ele executará a função `MPI_Comm_get_parent()` localizada na linha 5. Esta função retornará o intercomunicador `InterComm` que ligará o intracomunicador do processo filho com o do pai. Por fim, a ilustração do relacionamento hierárquico resultante após a criação do processo filho é encontrada no retângulo III da mesma figura. Nela, cada processo está interno ao seu intracomunicador (elipse) e interligado através do intercomunicador representado pela elipse vazada.

A função `MPI_Comm_spawn()` pode criar um ou vários processos à cada chamada. Dependendo da forma utilizada, dois modelos hierárquicos de comunicação podem ser criados. No primeiro, todos os processos são criados em uma única chamada à função, desta forma, todos os processos criados pertencerão ao mesmo intracomunicador. A Figura

Figura 5 – Criação de 3 processos em único `MPI_Comm_spawn()`

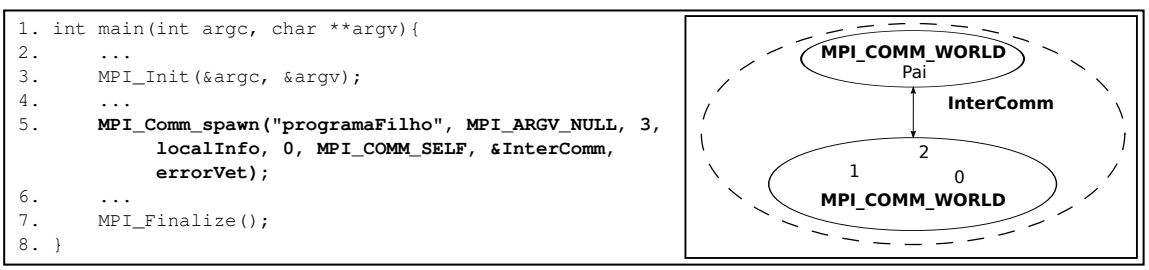
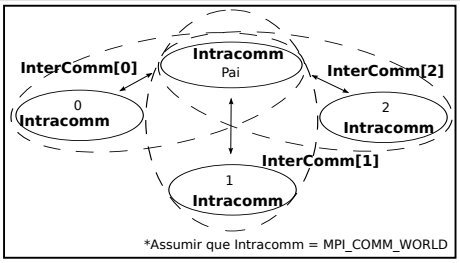


Figura 6 – Criação de 3 processos em múltiplos `MPI_Comm_spawn()`

```
1. int main(int argc, char **argv){
2.     ...
3.     MPI_Init(&argc, &argv);
4.     ...
5.     for(i = 0; i < 3; i++)
6.       MPI_Comm_spawn("programaFilho", MPI_ARGV_NULL,
           1, localInfo, 0, MPI_COMM_SELF, &InterComm[i],
           errorVet);
7.     ...
8.     MPI_Finalize();
9. }
```



5 apresenta o pseudocódigo para criação de três processos.

Ainda na Figura 5, na linha 5, ocorre a chamada para a função que criará os processos filhos. O consequente modelo hierárquico de comunicação é apresentado no retângulo ao lado do pseudocódigo. A principal vantagem deste método é possibilitar a comunicação entre todos os filhos criados, pois todos pertencem ao mesmo intracomunicador. Entretanto, como a primitiva `MPI_Comm_spawn()` é bloqueante, o processo pai fica bloqueado até concluir a criação de todos os filhos.

No segundo modelo possível, um processo é criado a cada chamada à função. Na Figura 6, é apresentado um pseudocódigo para criação de 3 processos em chamadas diferentes. Neste exemplo, a criação dos 3 processos foi realizada interna à um laço de repetição (linha 6). Neste caso, como definido no terceiro parâmetro da função `MPI_Comm_spawn()`, será criado somente um processo de cada vez. Com isto, cada processo criado pertencerá à um intracomunicador diferente e a comunicação entre eles deverá ser realizada através dos intercomunicadores que ligam os filhos ao pai e vice-versa. O modelo hierárquico gerado é encontrado no retângulo ao lado do pseudocódigo.

Além das possíveis comunicações ponto a ponto (utilizando inter e intracomunicadores) e coletivas (utilizando intracomunicadores) descritas na Seção 2.2.2, o MPI-2 permite ainda comunicações coletivas através de intercomunicadores. Portanto, um processo que pertence a um intracomunicador pode comunicar-se com N processos filhos que estão em outro intracomunicador através do intercomunicador que os liga.

A criação dinâmica de processos fornecidas pelo MPI-2 permite ampliar a gama de problemas que podem ser resolvidos através do padrão MPI, já que o modelo de aplicação deixou de ser estritamente SPMD, passando a suportar também o MPMD. Entretanto, a criação de processos em tempo de execução gera um custo extra para a aplicação, o qual pode ser compensado através da utilização de características avançadas do MPI. Este trabalho procura analisar o impacto do uso da criação dinâmica de processos. Para isto, implementaremos dois problemas da suíte de problemas *Cowichan Problems*, conforme detalhado na próxima seção.

Tabela 1 – Características do grupo 1 de Cowichan

Característica / Problemas	*Granularidade*	*Comunicação*	*Balanceamento de Carga*	*Busca em Árvore*
Active Chart Parsing	x			
Image Thinning		x		
kece				x
Polygon Overlay and Display			x	
Skyline Matrix Solver		x	x	
The Turing Ring	x	x	x	

2.3 *Cowichan Problems*

A suíte de problemas *Cowichan Problems* consiste em mais de 20 problemas voltados à análise de Sistemas de Programação Paralela. A suíte é classificada em dois grupos de acordo com a complexidade dos problemas. Wilson e Bal (1996) definiram o grupo 1, que contém os principais problemas. Estes, exigem maior conhecimento sobre técnicas avançadas de programação paralela. Já o grupo 2, que possui problemas trivialmente paralelizáveis é descrito em Wilson (1994).

A Tabela 1 exibe as principais características encontradas nos problemas do grupo 1. Nos problemas que possuem a granularidade como característica (*Active Chart Parsing e The Turing Ring*), o programador precisa encontrar o melhor ajuste entre comunicação e computação. Já em problemas onde a característica é a comunicação (*Image Thinning, Skyline Matrix Solver e The Turing Ring*), ocorre intensa troca de dados entre os processos. Dependendo da quantidade de dados, a tarefa de comunicação pode gerar custo extra para a aplicação. Nestes casos, cabe ao programador ajustar a comunicação, procurando eliminar este custo extra, a fim de obter melhor desempenho da aplicação. Isto é possível com a utilização de comunicações do tipo não-bloqueante (especificadas na Seção 2.2.2).

Problemas que operam sobre grande quantidade de dados, instigam o programador a realizar o balanceamento de carga (*Polygon Overlay and Display, Skyline Matrix Solver e The Turing Ring*), a fim de que os processos recebam tarefas/dados de forma balanceada, evitando assim a ociosidade dos processos e elevando a eficiência da aplicação. Por fim, a Busca em Árvore é uma característica encontrada em um único problema (*Kece*). Neste caso, o programador deve ser capaz de controlar o custo extra gerado pelas diferentes heurísticas de busca realizadas em estruturas do tipo árvore.

A linguagem para programação em sistemas distribuídos Orca, teve sua usabilidade testada utilizando os problemas do grupo 1 em Wilson e Bal (2007). Visando testar a comunicação em Orca, Bouman (1995) utilizou o problema *Skyline Matrix Solver*, que

necessita comunicação intensiva e ajuste da carga de trabalho entre os processos. Os *Cowichan Problems* também foram utilizados para testar a flexibilidade da linguagem de programação orientada a objetos X10 em Paudel e Amaral (2011).

Este trabalho irá utilizar esta suíte de problemas com o objetivo de avaliar a criação dinâmica de processos utilizando MPI em duas aplicações com características diferentes. Para isto, escolhemos um problema com complexidade baixa de programação (Jogo da Vida – Grupo 2) e um problema com complexidade alta de programação (*Skyline Matrix Solver* – Grupo 1), exigindo mais do programador. O Jogo da Vida permitirá avaliar o comportamento da criação dinâmica de processos em aplicações cujo foco é a distribuição dos dados entre os processos. Esta aplicação exige comunicação intensiva para troca de dados e o tempo de computação é similar entre os processos. Já o *Skyline Matrix Solver*, possibilitará avaliar a criação dinâmica de processos em aplicações que possuem carga irregular de trabalho, pois esta aplicação possui tempo de computação diferente entre os processos e possui comunicação intensa envolvendo grande quantidade de dados.

2.4 Trabalhos Relacionados

Este trabalho tem como objetivo realizar um estudo sobre o desenvolvimento de aplicações MPI com criação dinâmica. A criação dinâmica de processos tem sido utilizada nos modelos de Divisão e Conquista. Nele, para cada nova sub-tarefa, um novo processo pode ser criado. Por exemplo, em Pezzi et al. (2006), é realizado um estudo sobre como programar esse tipo de aplicação para se adaptar aos recursos, e como fazer um bom aproveitamento dos recursos disponíveis. Para isto, foram utilizadas duas aplicações que são resolvidas através da Divisão e Conquista: *Fibonacci* e *N-Queens*. Os resultados mostram que pode ser eficiente a utilização do MPI-2 para este tipo de aplicação, mais precisamente àquelas que necessitam muito poder computacional.

MPI-2 também tem sido utilizado em conjunto com o OpenMP em Leopold e Süß (2006). O trabalho tem o propósito de fornecer dinamicidade e adaptabilidade à aplicação *WaterGAP* modelada sobre o padrão Mestre/Trabalhador. Ela computa a disponibilidade atual e futura de água no mundo. Para isto, divide o continente em pequenas células (grades) de tamanho igual e utiliza dados de entrada (clima, vegetação) para simular o controle de água (precipitação) e a velocidade do fluxo da água em rios. Os cálculos realizados sobre cada célula são divididos em três níveis: grande, médio e fraco. Portanto, células podem ter tempos de computação diferentes. Com o gerenciamento de processos em tempo de execução, o processo Mestre, através de cálculos realizados por um gerenciador de carga, consegue adaptar a aplicação à arquitetura disponível conforme há aumento da carga de trabalho. Os resultados mostram que o uso do MPI-2 melhorou o desempenho da aplicação, principalmente por permitir melhor gerenciamento dos recursos

disponíveis.

Estudos têm sido realizado com o propósito de fornecer melhor aproveitamento dos recursos computacionais, porém através da melhoria de políticas de distribuição de novos processos nos recursos disponíveis da arquitetura alvo. Cera et al. (2006) propõe a implementação de um módulo extra para o escalonador de processos. Este módulo tem a finalidade de definir em tempo de execução em qual recurso computacional o novo processo será alocado. Ele interage com o processo que realizará a chamada para a primitiva `MPI_Comm_spawn()` e através de um grafo de tarefas presente no gerenciador de recursos do escalonador, indica qual recurso é mais adequado para alocar fisicamente um novo processo. De posse do local onde será criado o novo processo, ocorre então a criação do mesmo. Quando o processo criado for finalizado, ele informa ao gerenciador de recursos, o qual irá atualizar o grafo de tarefas. Este módulo foi implementado em dois modelos de escalonador: *Round-Robin* e *List Scheduling*. Os resultados mostram que para ambos os escalonadores, conseguiu-se obter equilíbrio na distribuição dos processos, melhorando assim, a distribuição dos processos criados em tempo de execução entre os recursos computacionais disponíveis.

A criação dinâmica de processos possibilita a implementação de aplicações MPI-2 flexíveis ou adaptativas. Portanto, uma aplicação com esta característica pode obter um melhor aproveitamento dos recursos computacionais disponíveis. Neste contexto, Cera et al. (2010) discute o uso de aplicações dinâmicas MPI para aumentar a taxa de utilização dos recursos de *clusters* de computadores. Neste trabalho, faz-se a utilização de trabalhos (*jobs*) maleáveis (ou seja, aqueles que podem se adaptar à recursos com disponibilidade dinâmica) através da dinamicidade oferecida pelo MPI-2 para avaliar o aproveitamento dos recursos. Os resultados mostram que a utilização do *cluster* pode ser aumentada em cerca de 25% com o uso de trabalhos maleáveis, quando comparado à uma abordagem não maleável.

Outra possibilidade de utilização do MPI-2 é oferecer meios de implementar tolerância a falhas à uma aplicação através da gerência de processos em tempo de execução. Por exemplo, o *midleware EasyGrid* MPI apresentado em Silva e Rebello (2011), implementa funções de criação dinâmica de processos para que aplicações MPI possam executar de maneira eficiente em ambientes onde as falhas de sistema, hardware e aplicação são recorrentes. Portanto, ao ocorrer uma falha na aplicação, o *midleware* desenvolvido contorna esta falha com o gerenciamento de processos em tempo execução, sem causar maiores danos a aplicação (i.e. finalização total da aplicação, causando desperdício do trabalho computado até o momento em que a falha ocorreu).

Entretanto, criar processos em tempo de execução traz custos adicionais à aplicação. Isto porque a criação dinâmica acontece de forma síncrona e leva ao estabelecimento de um modelo de comunicação hierárquico entre os processos criados e seus criadores

(vide Seção 2.2.3). Portanto, conhecer o custo da utilização desta característica pode auxiliar o desenvolvedor em momentos onde deve optar entre a criação de *threads* ou processos em tempo de execução. Este trabalho realiza um estudo sobre o desenvolvimento de aplicações com criação dinâmica de processos, comparando aplicações desenvolvidas com criação estática e dinâmica de processos, em que a principal diferença entre elas é a criação de processos em tempo de execução.

2.5 Conclusão do Capítulo

Este capítulo contextualizou a programação paralela em sistemas de memória distribuída utilizando MPI, apresentando as principais diferenças envolvendo as normas MPI-1 e MPI-2 no âmbito deste trabalho. Para isto, inicialmente discutiu-se sobre o paradigma de programar paralelo em sistemas de memória distribuída. Adicionalmente foram apresentados os modelos de programação que são eventualmente utilizados na paralelização de problemas utilizando MPI.

Inicialmente, na Seção 2.2.2, foram destacadas as funções básicas do MPI (inicialização, comunicação e finalização). Vimos também, que a comunicação entre os processos ocorre através de canais de comunicação chamados comunicadores, que podem ser intra e intercomunicadores.

A norma MPI-2 foi apresentada na Seção 2.2.3. Conforme mostrado, ela possibilita a criação dinâmica de processos, e a utilização desta característica resulta em dois modelos hierárquicos de comunicação (todos os processos em um único `MPI_Comm_spawn()`, ou um processo a cada chamada à primitiva). Vimos, que o processo filho irá encontrar-se em um intracomunicador diferente do pai, e que a comunicação entre eles ocorrerá através do uso de intercomunicadores retornados durante a criação do processo.

A Seção 2.3 descreve os *Cowichan Problems*, apresentando as suas principais características. Nela, é discutida a utilização e importância destes problemas na análise de sistemas de programação paralela. Também foram apresentados os dois problemas alvo que serão utilizados no desenvolvimento deste trabalho. Por fim, a Seção 2.4 apresentou os trabalhos relacionados dando ênfase na importância e na contribuição deste trabalho.

Conforme apresentado, a criação dinâmica de processos adiciona dinamicidade a aplicação. Entretanto, criar processos em tempo de execução eleva a complexidade do desenvolvimento de aplicações MPI, além de adicionar um custo extra ao tempo de execução da aplicação. Portanto, este trabalho discute o desenvolvimento de aplicações com criação dinâmica de processos, buscando encontrar o impacto e a complexidade envolvendo o uso desta característica fornecida pelo MPI. Desta forma, as Seções 3 e 4 apresentam os estudos de caso envolvendo os problemas Jogo da Vida e *Skyline Matrix Solver* respectivamente.

3 Estudo de Caso I: Jogo da Vida

Este capítulo discute um estudo de caso onde a criação dinâmica de processos é utilizada na implementação paralela do problema do Jogo da Vida. Este estudo objetiva evidenciar o custo adicional imposto pela criação dinâmica de processos e o impacto causado pelas comunicações que ocorrem em uma organização hierárquica de comunicadores em aplicações que possuem características similares à este problema. Adicionalmente, buscaremos explorar as potencialidades do MPI visando mostrar até que ponto pode-se atenuar o impacto da criação de processos em tempo de execução.

Para isto, inicialmente na Seção 3.1 é contextualizado o problema alvo. A seguir, a Seção 3.2 apresenta as principais características da implementação em ambientes de memória distribuída. Ainda na mesma seção, é apresentado um estudo preliminar com o objetivo de encontrar o melhor método de distribuição de dados entre os processos. De posse do melhor método encontrado para esta aplicação, a Seção 3.3 descreve as características das versões do problema alvo implementadas em MPI-1 e MPI-2. Já na Seção 3.4 são apresentados os dois cenários de testes criados para realização da análise e discussão dos resultados obtidos. Por fim, a Seção 3.5 conclui o capítulo.

3.1 Descrição do Problema

O problema do Jogo da Vida (*Game of Life* - (GARDNER, 1970)) consiste em simular a evolução da vida em uma sociedade. A sua principal característica é que a evolução da sociedade não ocorre com a influência de jogadores externos e sim através da utilização de regras determinísticas. A sociedade é representada por um conjunto de células, em que cada célula pode encontrar-se em dois estados: viva ou morta. Conforme Gardner (1970), a evolução destas células leva em consideração a aplicação das seguintes leis genéticas definidas por John H. Conway sobre as suas células vizinhas:

1. Uma célula morta somente se torna viva quando possuir exatamente três vizinhos vivos;
2. Células vivas morrem quando possuírem mais de três vizinhos vivos (Superpopulação);
3. Células vivas morrem quando possuírem menos de dois vizinhos vivos (Solidão);
4. Qualquer situação diferente das descritas acima, as células seguem inalteradas.

Figura 7 – Pseudocódigo sequencial do Jogo da Vida.

```
1.  int main(){
2.     Define vizinhos;
3.     for(etapa=1;etapa<=TotalEvolucao;etapa++){
4.        for(i=1;i<=N;i++){
5.           for(j=1;j<=N;j++){
6.              Aplicam-se as leis genéticas sobre a célula S_ij;
7.           }
8.        }
9.        Atualiza dados das bordas;
10.    }
11. }
```

Para melhor entender o funcionamento do jogo da vida, a Figura **??** apresenta seu pseudocódigo sequencial. Nele, partimos do pressuposto de que a sociedade inicial já está alocada em uma matriz S de tamanho $N \times N$. A Figura 8 ilustra a relação de vizinhança onde as setas indicam quais células são vizinhas das células das extremidades da matriz. Esta operação é necessária, pois os elementos situados nas extremidades da matriz tem seus vizinhos alocados na outra extremidade da matriz, como se a matriz representasse um torus (BLOK; BERGERSEN, 1997). Por exemplo, a coluna da extremidade esquerda da matriz passa a ser a borda da extremidade direita e assim sucessivamente.

O laço de repetição com início na linha 3 é o responsável por controlar a quantidade total de evoluções da sociedade. Já os laços aninhados das linhas 4 e 5 percorrem a matriz (sociedade) aplicando as leis genéticas. Para definir o estado de uma célula (viva ou morta) no próximo ciclo de vida da sociedade, as leis genéticas serão aplicadas à soma das células vivas presentes em sua vizinhança.

Um exemplo da evolução de um ciclo de vida é apresentado na Figura 9. A matriz da esquerda representa a sociedade inicial enquanto que a matriz da direita apresenta a sociedade após a evolução, em que as células vivas são representadas pela cor preta e as

Figura 8 – Relação de vizinhança entre as extremidades da matriz 6×6.

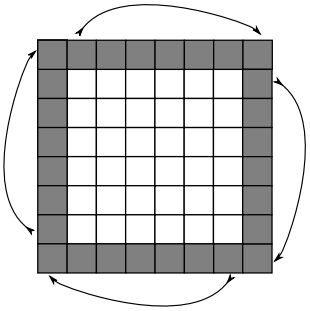
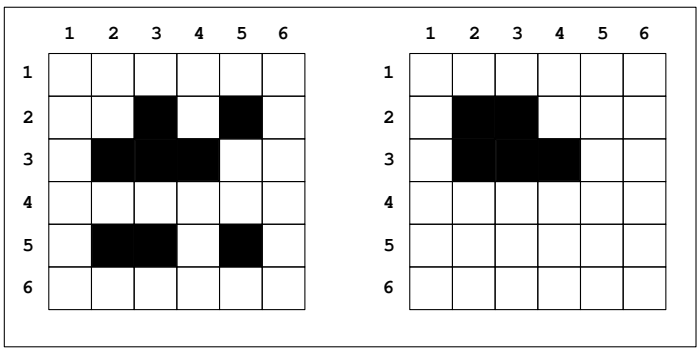


Figura 9 – Ciclo de evolução do Jogo da Vida.



Evolução da sociedade 6×6 através das leis genéticas de Conway.

células mortas pela cor branca. Nela, a posição S_{33} da matriz da esquerda aloca uma célula viva e seu estado futuro dependerá dos seus vizinhos. Para isto, realiza a soma das células vivas que são suas vizinhas. Neste caso, existem três células vivas, alocadas nas posições S_{23}, S_{32} e S_{34}. Desta forma, aplicando as leis genéticas, no próximo ciclo de evolução ela continuará sendo uma célula viva. Esta situação não ocorre com a célula da posição S_{55}, pois não possui nenhum vizinho, e assim esta célula morrerá por solidão.

Após finalizar um ciclo de evolução da sociedade, existe a necessidade de atualizar os dados das bordas, conforme mostrado na Figura 8. Esta operação ocorre na linha 9 do pseudocódigo da Figura 7. Ele será utilizado como base para as propostas de paralelização que serão apresentadas no decorrer deste capítulo. A próxima seção apresentará as características relacionadas a implementação paralela do problema alvo para sistemas de memória distribuída.

3.2 Paralelização do Jogo da Vida

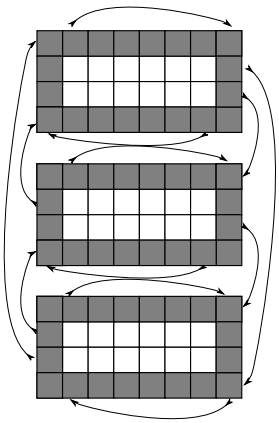
3.2.1 Organização Geral

O Jogo da Vida possui características bem definidas relacionadas à sua implementação paralela em sistemas de memória distribuída. Dentre elas, apresenta-se nesta seção as que são comuns às nossas propostas, tanto com MPI-1 quanto com MPI-2.

Analisando o pseudocódigo da Figura 7, identificamos que as iterações dos laços de repetição aninhados das linhas 4 e 5 podem ser realizados em paralelo. Desta forma, a matriz de entrada (sociedade) pode ser particionada entre os processos, em que cada processo ficará responsável por aplicar as leis genéticas sobre uma partição dos dados. Por este motivo, nós optamos por implementar a paralelização do Jogo da Vida através do particionamento dos dados.

Um exemplo de particionamento dos dados é apresentado na Figura 10. Nela, a matriz 6×6 foi dividida em 3 partições de 2 linhas. Conforme apresentado anteriormente,

Figura 10 – Relação de vizinhança entre as extremidades da matriz 6×6.



Particionamento da matriz em 3 partições de 2 linhas

o algoritmo do Jogo da Vida é baseado no estado da vizinhança, e isto requer um cuidado com as células das extremidades da matriz. Logo, há a necessidade de incorporar a borda no particionamento inicial dos dados, conforme ilustrado.

Conforme apresentado anteriormente, este trabalho objetiva estudar o desenvolvimento de aplicações com criação dinâmica de processos em MPI. Todos os modelos de programação apresentados na Seção 2.2.1 possibilitam o uso da criação dos processos em tempo de execução. Este estudo partirá da implementação utilizando o modelo Mestre/Trabalhador, o qual é o mais utilizado por quem está iniciando a desenvolver em paralelo. Desta forma, existirá um processo chamado de Mestre e os demais processos denominados Trabalhadores. O pseudocódigo ilustrado na Figura 11 descreve e mostra as funcionalidades de cada um deles.

Inicialmente, o **Mestre** receberá como parâmetros de entrada a matriz inicial e o número de processos Trabalhadores (necessário para o processamento da matriz). Na linha 1, ele realizará o particionamento dos dados da matriz, levando em conta o método de distribuição de dados. Após particionar a matriz, o Mestre definirá a relação das extremidades das partições (conforme ilustrado na Figura 10). Então, as partições de dados serão enviadas aos Trabalhadores, e o Mestre aguardará o resultado final da computação. Por fim, ele organizará os dados recebidos dos Trabalhadores na matriz

Figura 11 – Exemplo de implementação Mestre/Trabalhador do Jogo da Vida.

```
   Mestre
1.    Particiona matriz;
2.    Define vizinhos;
3.    Envia partições de dados;
4.    Recebe partições de dados;
5.    Finaliza;
```

```
   Trabalhador
1.    Recebe partições de dados;
2.    Computação sob seus dados;
3.    Atualização dos dados da borda;
4.    Envio das partições de dados
         finais ao Mestre;
5.    Finaliza;
```

final do jogo.

Ainda na Figura 11, temos as ações dos processos Trabalhadores, em que inicialmente (linha 1), cada um deles receberá a partição dos dados da matriz enviada pelo Mestre. De posse destes dados, ele realizará a computação (aplicação das leis genéticas) sobre estes dados. Após a aplicação das leis genéticas, realiza-se a atualização dos dados da borda. Esta computação e atualização dos dados da borda são repetidas até que o número total de evoluções da sociedade seja atingido. Quando a sociedade chegar ao fim de sua evolução, o Trabalhador retornará os dados atualizados para o Mestre, e assim, finalizará sua execução.

Conforme visto no pseudocódigo da Figura 11, a primeira função do Mestre é particionar a matriz de entrada entre os processos envolvidos na paralelização. A forma como isto é feito influencia diretamente na eficiência da aplicação. Assim, com a intenção de encontrar o método que melhor se adapta ao problema, foram implementados dois métodos de divisão de dados, os quais serão apresentados na próxima seção.

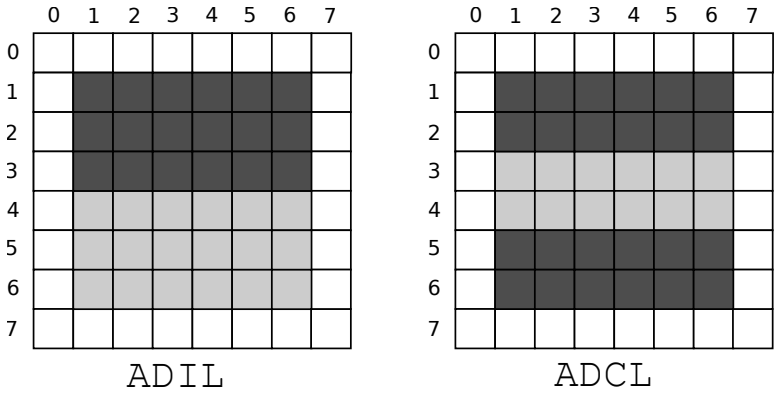
3.2.2 Métodos de Distribuição de Dados

A eficiência de programas paralelos para memória distribuída está diretamente relacionada à distribuição da carga de trabalho entre os nós. Neste sentido, foram desenvolvidos dois métodos de distribuição de carga: o Algoritmo de Divisão Igualitária de Linhas – ADIL, que distribui a carga de forma igualitária entre os processos no início da execução; e o Algoritmo de Divisão por *Chunk* de Linhas – ADCL, que distribui *chunk* de dados sob demanda durante a execução (LORENZON; CERA; ROSSI, 2012a). Os algoritmos foram desenvolvidos utilizando a linguagem de programação C e paralelizados com MPI-1.

ADIL: O Mestre particiona a matriz original de forma igualitária entre os processos Trabalhadores. Desta forma, cada Trabalhador receberá os dados de uma única vez no início da execução. A Figura 12 ilustra o particionamento de uma matriz 6×6 entre dois Trabalhadores (cinza escuro e cinza claro). Neste exemplo, as bordas da partição destinada ao Trabalhador representado em cinza escuro corresponde as linhas 0 e 4.

ADCL: O Mestre particiona os dados baseado em um *chunk* de linhas definido pelo usuário. As partições são atribuídas aos Trabalhadores em tempo de execução seguindo a política de escalonamento *Round Robin* (FOSTER, 1995). A Figura 12 mostra como é realizado o particionamento dos dados entre 2 Trabalhadores (cinza escuro e cinza claro) em uma matriz 6×6. O *chunk* considerado em questão é de duas linhas. A borda da partição destinada ao Trabalhador representado por cinza claro corresponde as linhas 2 e 5.

Figura 12 – Ilustração do particionamento dos dados.



Cabe ainda ressaltar as características que contemplam as duas versões:

1. O envio das partições dos dados iniciais pelo Mestre é realizado através de comunicações síncronas (`MPI_Send()` do lado do Mestre e `MPI_Recv()` do lado dos Trabalhadores). Optou-se por este modelo de comunicação, pois a computação dos Trabalhadores somente começará após o recebimento completo dos dados.

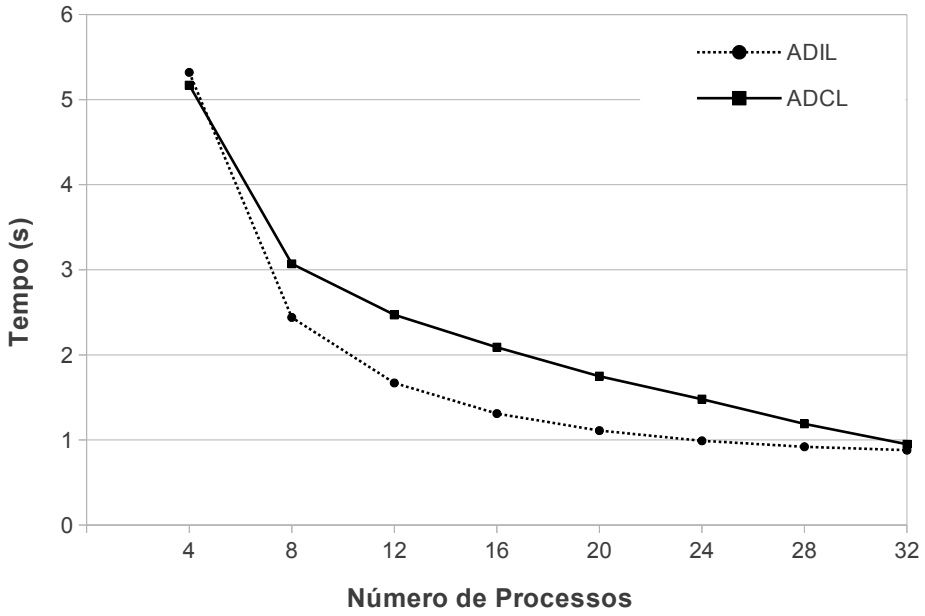
2. As comunicações de troca de borda entre os Trabalhadores são todas gerenciadas pelo Mestre com o uso de comunicações não-bloqueantes (`MPI_Send()` combinada com `MPI_Irecv()`). Desta forma, ao final de cada ciclo de evolução as bordas são enviadas para o Mestre, que redireciona estas bordas para os Trabalhadores correspondentes.

3.2.3 Resultados Experimentais: Análise da Distribuição de Dados

Os testes foram executados em um *cluster* do tipo *Beowulf* contendo 8 nós computacionais interligados por uma rede *Gigabit Ethernet*. Cada nó estava equipado com um processador Intel *Core i5*, com frequência de 3.20 Ghz. Em cada nó, haviam 4 *cores*, sendo 2 físicos e 2 lógicos (através da tecnologia *Hyper-Threading*) e 4 *GigaByte* de memória RAM. O Sistema Operacional em uso foi o Debian (*kernel* 2.6.32-5-amd64), e a versão da distribuição do MPI utilizada foi a OpenMPI 1.6. Para ambas as versões implementadas, o tempo de computação desconsidera o custo de operações de entrada e saída (leitura e escrita de arquivo).

Nosso conjunto de testes contempla execuções com diversas grandezas de matrizes com a finalidade de avaliar o comportamento do algoritmo conforme aumenta a quantidade de dados. Para isto, foram definidos os seguintes tamanhos: 1024×1024, 2048×2048, 4096×4096, 6144×6144, 8192×8192 e 16384×16384, este último indicando o tamanho máximo para a arquitetura em questão. Para aproveitar melhor os recursos disponíveis em

Figura 13 – Tempos de execução das versões ADIL e ADCL para execução com 4 até 32 processos em uma Matriz de tamanho 2048 × 2048



cada nó computacional, optamos pela utilização de 4 processos em cada nó, distribuídos através da política de escalonamento *Round-Robin*.

Para a execução da versão ADCL, foram definidos os valores do *chunk* de linhas com base na quantidade de blocos que cada processo irá receber. Isto implica na quantidade de comunicação necessária para atualizações de borda. Para fornecer uma comparação justa e avaliar o comportamento conforme aumenta o tamanho da matriz, o *chunk* aumenta de acordo com o tamanho da matriz. Desta forma, para cada matriz sempre existirá 64 blocos de linhas. Foram realizadas 30 execuções para cada configuração de quantidade de processos e tamanho da matriz, e analisaremos a média dos tempos de execução obtidos. O desvio padrão foi sempre inferior a 0,07 e 0,7 segundos nas execuções da versão ADIL e ADCL respectivamente.

Na Figura 13 é apresentado o gráfico dos tempos de execução com uma matriz de tamanho 2048 × 2048. Para este tamanho de matriz, o *chunk* definido para a versão ADCL foi de 32 linhas para cada bloco, totalizando assim, 64 blocos. Nela pode-se notar que a partir da execução com 4 processos, a versão ADIL sempre executou em menos tempo quando comparada à versão ADCL. Como pode ser visto na Figura 14, para uma matriz de 16384 × 16384, a versão ADIL continuou a mais eficiente ao aumentar-se a quantidade de dados. Os demais gráficos com os tamanhos de matrizes intermediários podem ser vistos no Apêndice B.

A Tabela 2 apresenta o percentual médio das diferenças dos tempos de execução para todo o nosso conjunto de teste. O percentual representa o quanto a versão ADIL foi

Figura 14 – Tempos de execução das versões ADIL e ADCL para execução com 4 até 32 processos em uma Matriz de tamanho 16384 × 16384

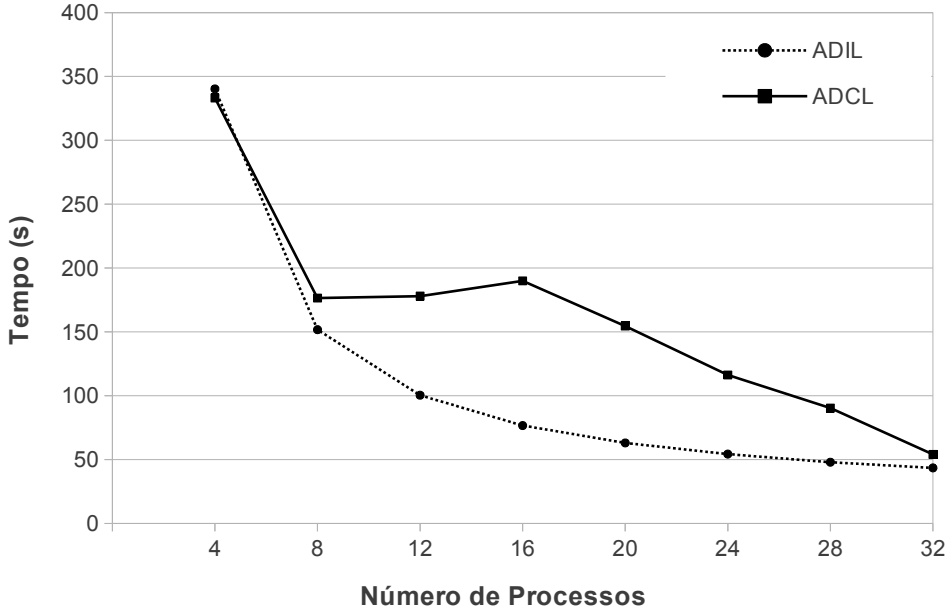


Tabela 2 – Percentual médio da diferença dos tempos de execução – ADIL X ADCL.

Ordem/Nº Processos	1024	2048	4096	6144	8196	16384
4	**-1%**	**-3%**	**-4%**	**-3%**	**-4%**	**-2%**
8	25%	26%	14%	14%	17%	16%
12	**47%**	48%	60%	70%	75%	77%
16	35%	**60%**	80%	86%	120%	**148%**
20	23%	58%	**85%**	**108%**	**133%**	145%
24	16%	49%	65%	96%	111%	114%
28	17%	29%	56%	61%	84%	89%
32	13%	8%	16%	19%	17%	24%

mais eficiente que a versão ADCL. Para auxiliar no entendimento dos dados, para cada grandeza de matriz (coluna), foi destacado em negrito o percentual mínimo e máximo de diferença entre as duas versões.

Pode-se observar na Tabela 2, que para todos os tamanhos de matrizes executados com 4 processos, a versão ADCL foi mais eficiente (sinal negativo informa que a versão ADIL obteve eficiência menor). Este cenário acontece somente com esta quantidade de processos, pois os 4 processos estão alocados em 4 *cores* de um mesmo nó. Desta forma, não se tem o ônus da comunicação, já que o MPI trata localmente a comunicação quando utilizado um ambiente de memória compartilhada. Podemos notar que a partir da utilização de 8 processos, conforme o tamanho da matriz é aumentado, maior é a diferença. Isto ocorre, pois a versão ADCL requer mais comunicações.

Ainda na Tabela 2, observa-se que a diferença dos tempos de execução entre as versões tornou-se considerável no intervalo entre 12 e 28 processos. Isto ocorre, pois o ADCL leva ao desbalanceamento de carga entre os processos. Por exemplo, para 20 processos (19 trabalhadores), enquanto que 12 Trabalhadores realizarão o trabalho sobre 3 blocos cada, 7 Trabalhadores realizarão o mesmo trabalho sobre 4 blocos. Assim, quando estes processos executarem sobre o quarto bloco, os demais trabalhadores estarão ociosos aguardando a finalização de um ciclo da evolução.

Outro fato que contribui para a versão ADIL ter sido a mais eficiente está relacionado a carga de trabalho destinado a cada processo Trabalhador. Com 4 processos, cada processo Trabalhador receberá uma carga de trabalho consideravelmente grande, visto que a matriz será particionada entre 3 Trabalhadores. Entretanto, conforme ocorre o aumento do número de processos (de 8 até 32 processos), a carga de trabalho destinada à cada Trabalhador diminui. Assim, os processos da versão ADCL passarão muito mais tempo comunicando, do que computando.

Com base nestes resultados, conclui-se que o método mais eficiente é o que particiona os dados da matriz igualitariamente entre os processos – ADIL. Assim, este método será utilizado no desenvolvimento das implementações MPI-1 e MPI-2.

3.3 Questões de Implementação

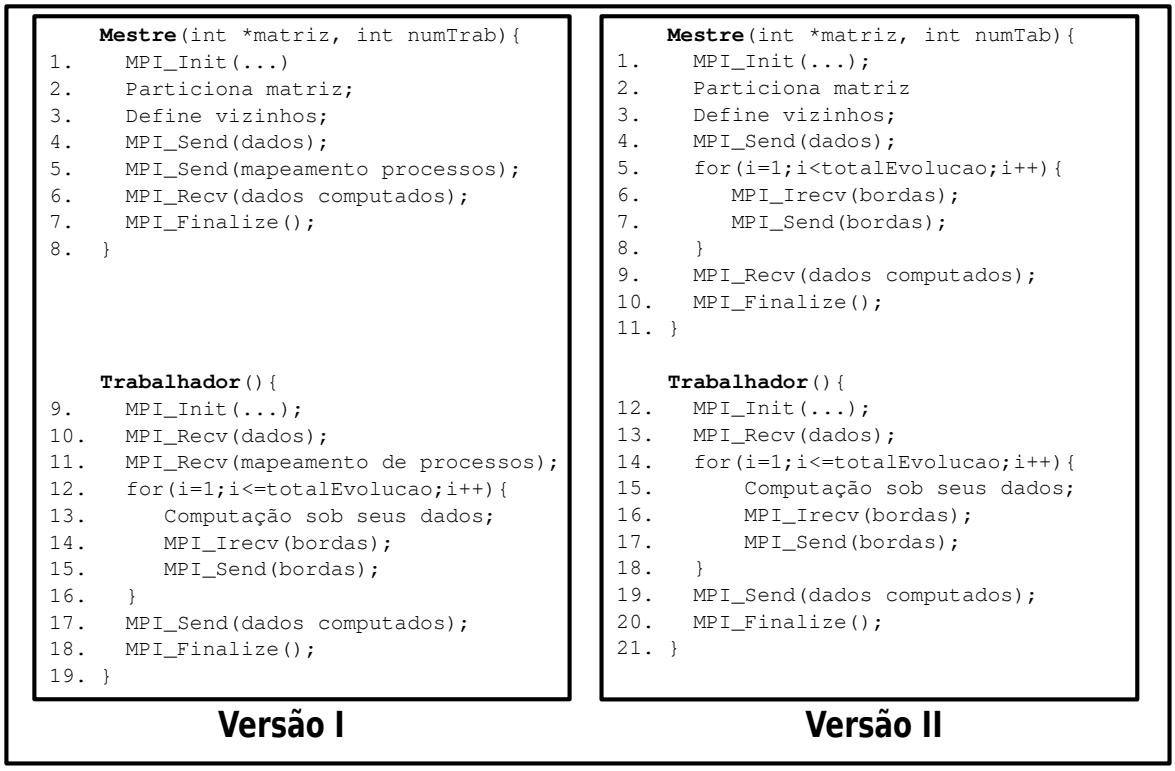
Esta seção apresenta as questões técnicas das implementações paralelas do Jogo da Vida. Assim, a Seção 3.3.1 mostra os detalhes das versões com MPI-1, enquanto que a Seção 3.3.2 mostra os detalhes com MPI-2.

3.3.1 Usando o MPI-1

Com o propósito de avaliar o comportamento decorrente da alteração do modelo hierárquico, do uso da criação dinâmica de processos, para explorar o universo de comparações MPI-1 e MPI-2, foram desenvolvidas duas versões da aplicação com MPI-1. Nos pseudocódigos apresentados na Figura 15 e na ilustração da Figura 16, pode-se acompanhar como as duas versões estão implementadas, em que ambas foram desenvolvidas em um contexto SPMD. Importante salientar que as funções apresentadas nos pseudocódigos das Figuras 15 e 17 não contém todos os parâmetros por questões de legibilidade do pseudocódigo. As especificações dos parâmetros de cada função são encontradas no Apêndice A.

A Versão I, descrita à esquerda da Figura 15 e ilustrada na Figura 16 apresenta a maneira convencional de desenvolver aplicações MPI-1 com o modelo de programação Mestre/Trabalhador. Nela, após o Mestre inicializar o ambiente MPI e realizar ajustes para a paralelização, ele envia os dados de entrada de cada Trabalhador através da co-

Figura 15 – Pseudocódigo das duas versões do Jogo da Vida com MPI-1



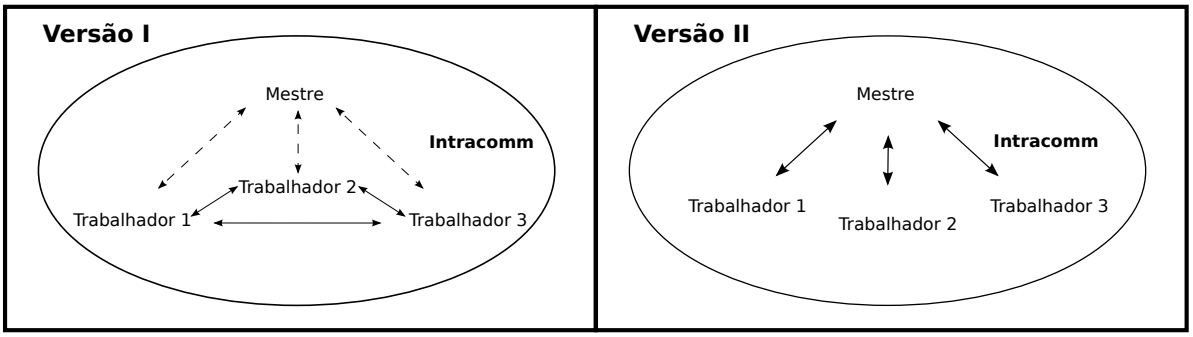
municação síncrona `MPI_Send()` (linha 4). Por sua vez, o Trabalhador, após inicializar o ambiente MPI, recebe através do `MPI_Recv()` os dados que irá computar (linha 10).

Em um segundo momento, o Mestre envia o mapeamento de processos aos Trabalhadores, para que eles possam atualizar os dados de suas bordas sem a necessidade de intermediação do Mestre. Esta comunicação também utiliza comunicação síncrona com as diretivas `MPI_Send()` do lado do Mestre (linha 5) e `MPI_Recv()` do lado dos Trabalhadores (linha 11). Após este envio, o processo Mestre aguardará na linha 6, o recebimento dos dados finais computados pelos Trabalhadores.

De posse dos dados necessários, os Trabalhadores aplicarão as leis genéticas sob os dados até a sociedade atingir seu nível final de evolução (linha 12 e 13). Após cada ciclo de evolução, ocorre a troca dos dados das bordas entre os processos Trabalhadores. Ela se dá através de comunicação assíncrona com o uso das diretivas combinadas `MPI_Irecv()` e `MPI_Send()` (linha 14 e 15). Desta forma, cada processo ficará aguardando através de uma comunicação não-bloqueante o recebimento dos dados atualizados das bordas para que possa continuar a computação.

Ao finalizar a evolução total da sociedade, cada Trabalhador retornará os dados computados para o Mestre. Este retorno ocorre através da comunicação síncrona `MPI_Send()` (linha 17). Por fim, após enviar os dados computados, o processo Trabalhador finaliza seu ambiente de execução (linha 18). Já o processo Mestre, aguarda o recebimento dos dados computados por todos os Trabalhadores para compor a sociedade

Figura 16 – Ilustração das duas versões do Jogo da Vida com MPI-1.



Versão I - Trabalhadores comunicam-se entre si e *Versão II* - Comunicações passam pelo Mestre.

final do jogo, para só assim finalizar o ambiente de execução (linha 7).

A Versão II, tem seu pseudocódigo descrito à direita da Figura 15 e é ilustrada na Figura 16. A diferença para a Versão I está nas linhas 5, 6 e 7, onde as comunicações para atualização de borda passam a ser intermediadas pelo Mestre. Desta forma, o mapeamento de atualização de bordas deixa de ser transmitido aos Trabalhadores. Porém, todas as trocas de bordas que na Versão I ocorriam de forma direta entre os Trabalhadores, passam a ser gerenciadas pelo Mestre.

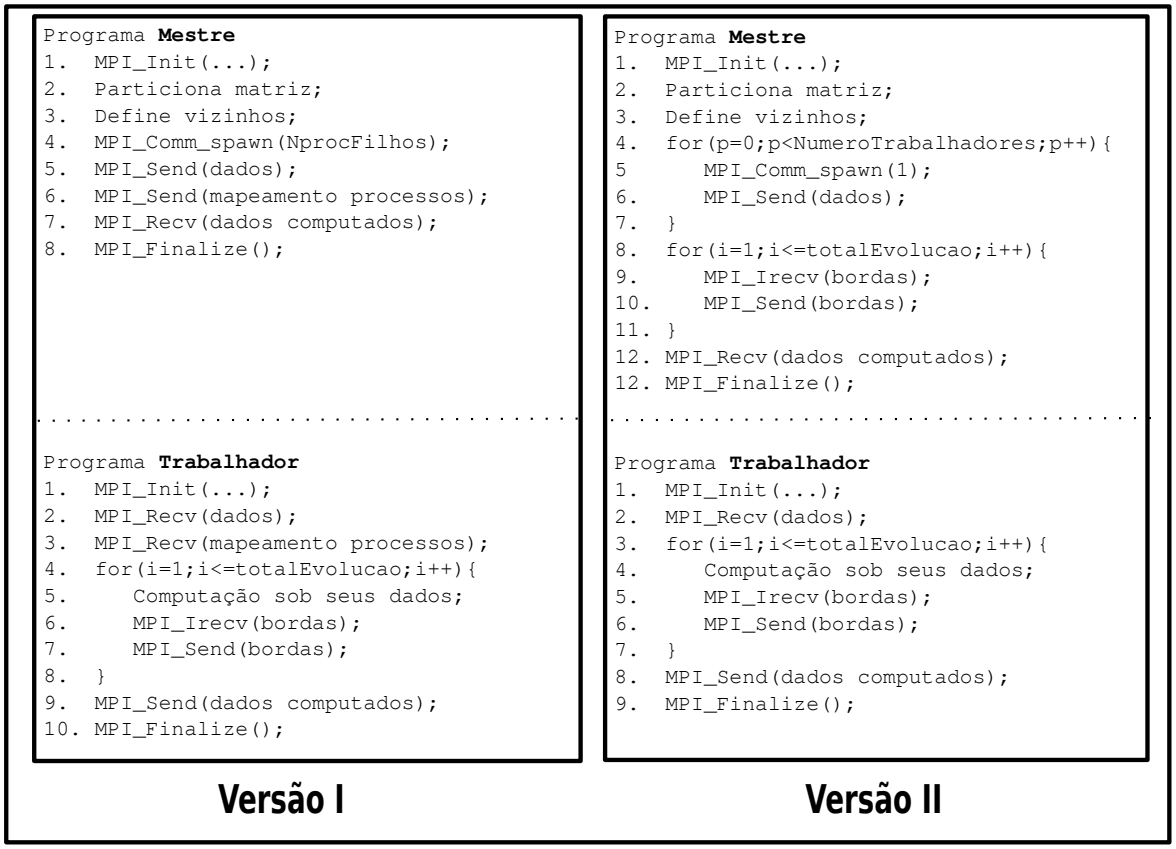
A Figura 16 ilustra a diferença técnica entre as duas versões implementadas com MPI-1: na Versão I, todos os processos fazem parte do intracomunicador `MPI_COMM_WORLD` (representado pela elipse), e os Trabalhadores comunicam-se com o Mestre (setas tracejadas) e entre si para atualização de borda (setas com linha cheia). Já na Versão II, os processos pertencem também ao mesmo intracomunicador `MPI_COMM_WORLD`, porém, as comunicações para atualização de borda são todas intermediadas pelo Mestre (setas com linha cheia).

3.3.2 Usando o MPI-2

A implementação em MPI-2 deu-se no contexto MPMD, em que o processo Mestre executa um programa e os Trabalhadores (filhos) executam um programa diferente. A primeira versão MPI-2 foi implementada de forma similar à versão I da implementação em MPI-1. A diferença foi a inserção da criação dinâmica de processos. As primitivas de comunicações utilizadas seguem as mesmas e o pseudocódigo da Versão I da Figura 17 apresenta como a aplicação está organizada.

A aplicação inicia pelo lançamento do processo Mestre (via linha de comando com `mpiexec`), o qual após realizar os ajustes iniciais (particionamento da matriz e definição dos vizinhos – linhas 1 e 2) cria todos os Trabalhadores (filhos) com uma única chamada `MPI_Comm_spawn()` (Linha 4, em que `NprocFilhos` representa o número total de

Figura 17 – Pseudocódigo das duas versões do Jogo da Vida com MPI-2.

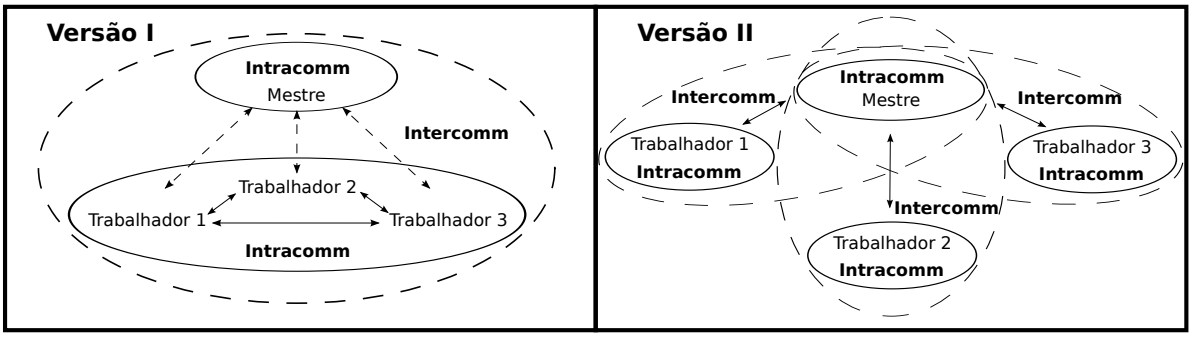


filhos/Trabalhadores). Isto faz com que todos os Trabalhadores estejam inseridos dentro do mesmo intracomunicador, um `MPI_COMM_WORLD()`, que não inclui o Mestre, conforme pode ser acompanhado na Figura 18. Assim, é possível manter as mesmas funcionalidades da versão padrão MPI-1. O Mestre comunica-se com os Trabalhadores através do intercomunicador retornado pelo `MPI_Comm_spawn()` e os Trabalhadores interagem entre si para atualizar as bordas através do intracomunicador `MPI_COMM_WORLD()` de seu grupo.

Na Versão II desenvolvida em MPI-2, descrita no pseudocódigo da Figura 17, o Mestre criará apenas um processo Trabalhador à cada `MPI_Comm_spawn()`. Isto permite criar Trabalhadores e destinar-lhes uma carga de trabalho sem obrigar a espera pela criação de todos os demais trabalhadores (Linhas 5 e 6). Desta forma cada processo pertencerá a um grupo diferente conforme ilustrado na Figura 18 e a as atualizações de borda entre os Trabalhadores serão intermediadas pelo Mestre, que é o único processo que tem acesso a todos os processos da aplicação. Logo, esta versão obriga um redirecionamento das mensagens, as quais passarão pelo Mestre conforme o modelo de comunicação hierárquico resultante da criação dinâmica de processos. Com esta versão, será possível verificar o impacto trazido pela hierarquia de comunicadores e o consequente redirecionamento das mensagens (LORENZON; CERA; ROSSI, 2012b).

A Figura 18 ilustra a diferença entre as duas versões do Jogo da Vida com MPI-2.

Figura 18 – Ilustração das duas versões do Jogo da Vida com MPI-2.



Versão I - Trabalhadores comunicam-se entre si e *Versão II* - Comunicações passam pelo Mestre.

Na Versão I existe um intracomunicador `MPI_COMM_WORLD` (elipse) permitindo que os trabalhadores atualizem suas bordas. É importante notar que as comunicações com o Mestre acontecem através do intercomunicador (elipse vazada) que liga o intracomunicador do Mestre aos Trabalhadores. Já na Versão II, as comunicações de atualização precisam passar pelo Mestre, visto que apenas este tem acesso (via intercomunicadores - elipses vazadas) aos demais Trabalhadores.

3.4 Resultados Experimentais: Comparação MPI-1 e MPI-2

Esta seção apresenta os resultados experimentais da comparação realizada entre MPI-1 e MPI-2. Para isso, definiu-se o ambiente e cenário de teste, os quais são apresentados na Seção 3.4.1. Após, a Seção 3.4.2 mostra os resultados obtidos.

3.4.1 Ambiente de Teste

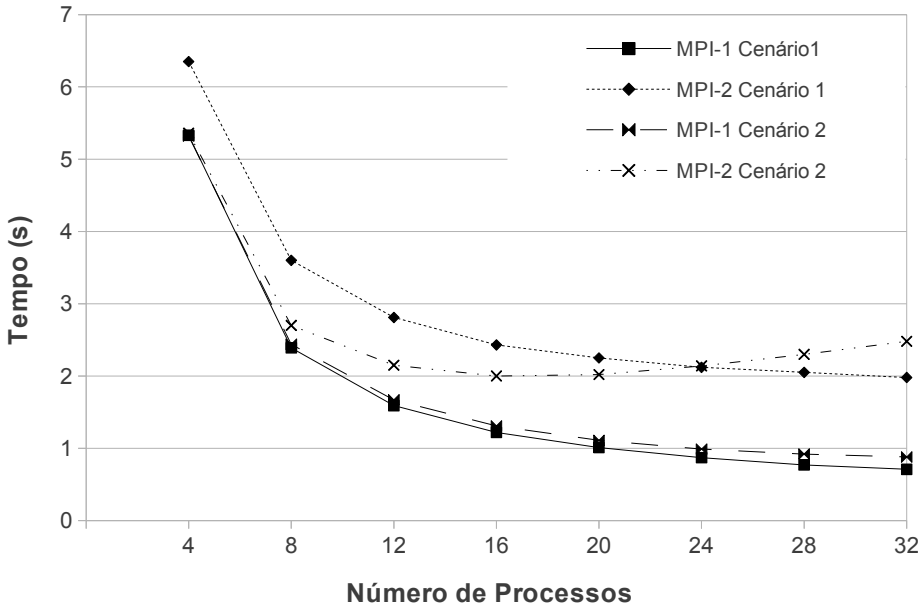
O ambiente e as características do conjunto de teste de execução seguem os mesmos que foram utilizados no estudo preliminar (ver Seção 3.2.3). Para cada configuração de tamanho de matriz, cenário e número de processos, foram realizadas 30 execuções onde analisaremos a média dos tempos de execução. O desvio padrão foi sempre inferior a $0,1$ segundos nas aplicações implementadas em MPI-1 e $0,17$ segundos nas aplicações MPI-2.

Para facilitar a compreensão dos testes, são analisados 2 cenários diferentes:

Cenário 1: Trabalhadores comunicam-se entre si para atualização das bordas – versões 1 tanto para o MPI-1 (Figura 16) quanto para o MPI-2 (Figura 18);

Cenário 2: Trabalhadores comunicam-se com o Mestre para atualização de bordas – versões 2 tanto para o MPI-1 (Figura 16) quanto para o MPI-2 (Figura 18).

Figura 19 – Tempos de execução de 4 a 32 processos para ambos os cenários em uma matriz 2048 × 2048



Reiterando, os objetivos desta comparação entre as implementações com MPI-1 e MPI-2 são: *(i)* identificar o impacto causado pela criação dinâmica de processos no desempenho da aplicação alvo; *(ii)* identificar o impacto causado pelas comunicações que ocorrem em uma organização hierárquica de comunicadores; e *(iii)* comparar o tempo de processamento considerando as diferenças entre os cenários testados (comunicações independentes ou gerenciadas pelo Mestre).

3.4.2 Análise dos Resultados MPI-1 e MPI-2

A Figura 19 apresenta os tempos de execução (em segundos) para uma matriz de entrada de tamanho 2048 × 2048 para os Cenários 1 e 2. Nela pode-se observar que o Cenário 1 com MPI-1 foi mais eficiente, ou seja, quando os Trabalhadores comunicam-se entre si para atualizar as bordas. O segundo Cenário mais eficiente foi o Cenário 2 com MPI-1. Analisando somente os cenários implementados com MPI-2, no intervalo de execução entre 4 e 20 processos o Cenário 2 foi mais eficiente. Após isto, a implementação do Cenário 1 foi a mais eficiente. Este comportamento se repete para todos os demais tamanhos de entrada (Figura 20 e demais gráficos de tamanhos de matrizes apresentados no Apêndice B). Nesta última, os tempos obtidos para os cenários e versões MPI-1 e MPI-2 são muito próximos. Isto deve-se ao custo extra ocasionado pela transferência de volumes de dados maiores nas trocas de mensagens.

Analisando o percentual de diferença entre as versões MPI-1 e MPI-2, temos que no pior dos casos, com uma matriz 2048 × 2048, o impacto da criação dinâmica de processos

foi de $1,9\%$ e 178% para 4 e 32 processos. Este valor é alto, pois para uma matriz pequena não há carga de trabalho suficientemente grande (tempo de computação), visto que o tempo total da aplicação com 32 processos em MPI-1 é menor do que 1 segundo. Entretanto, com uma matriz 16384×16384, que possui carga de trabalho superior, o impacto causado pela criação dinâmica de processos foi de apenas $0,2\%$ e $3,5\%$ para 4 e 32 processos respectivamente.

Para facilitar a compreensão da análise de desempenho das diferentes configurações de teste, as Figuras 21 e 22 apresentam a média da diferença dos tempos de execução (em segundos) para o intervalo entre 4 a 32 processos, considerando todos os tamanhos de matrizes. Nelas, as barras de erro representam a variação entre as médias para cada número de processos testados.

Observando o gráfico da Figura 21a, é possível notar que, independente da quantidade de processos em execução, a média ficou próxima a $1,25$ segundos. Esta média representa o quanto a versão MPI-1 foi mais rápida que a versão MPI-2 no Cenário 1. Com estes dados, é possível identificar que esta diferença refere-se ao impacto da criação dinâmica de todos os processos em um único `MPI_Comm_spawn` na nossa aplicação, uma vez que foi mantido o mesmo modelo de comunicação para ambas (Trabalhadores trocando bordas entre si). Lembrando que o Mestre permanece bloqueado até que todos os Trabalhadores tenham sido criados, já que a criação processos é uma operação síncrona. Só depois disso é que os Trabalhadores receberão os dados à serem computados.

É importante notar que o tempo destacado acima é inteiramente dependente da

Figura 20 – Tempos de execução de 4 a 32 processos para ambos os cenários em uma matriz 16384×16384

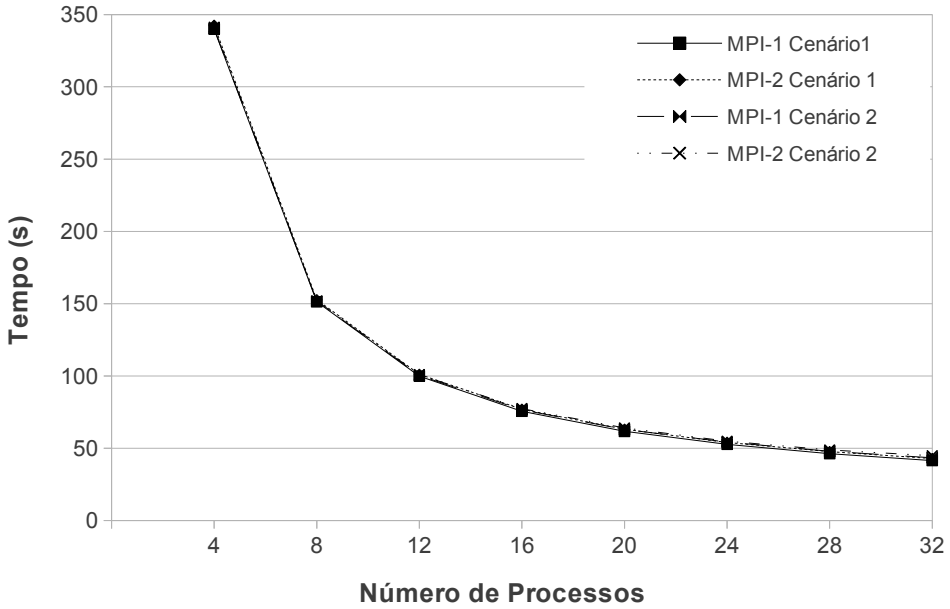
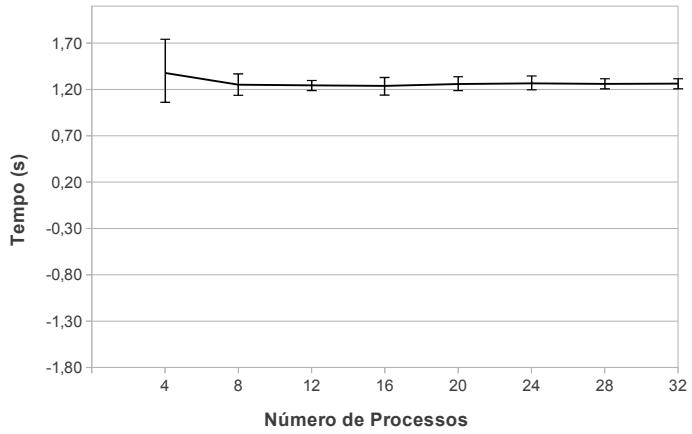
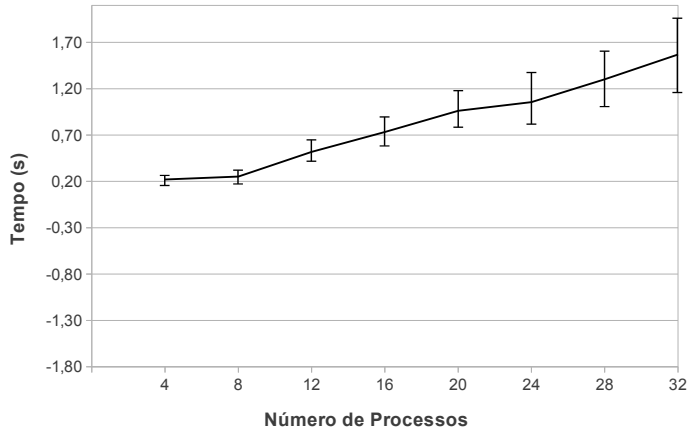


Figura 21 – Média da diferença dos tempos de execução para cada quantidade de processos testados



(a) Cenário 1 – MPI-1 × MPI-2

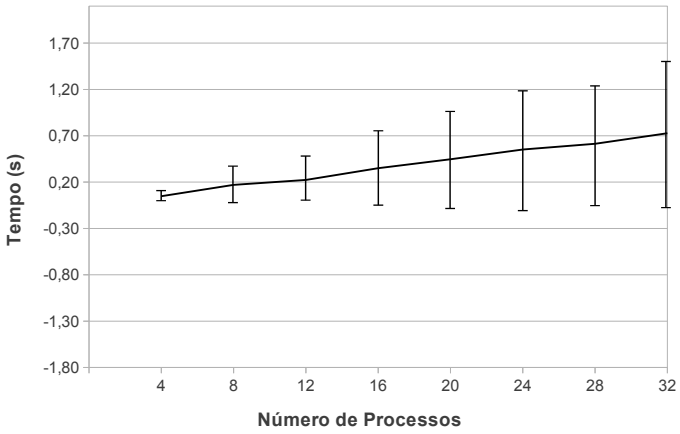


(b) Cenário 2 – MPI-1 × MPI-2

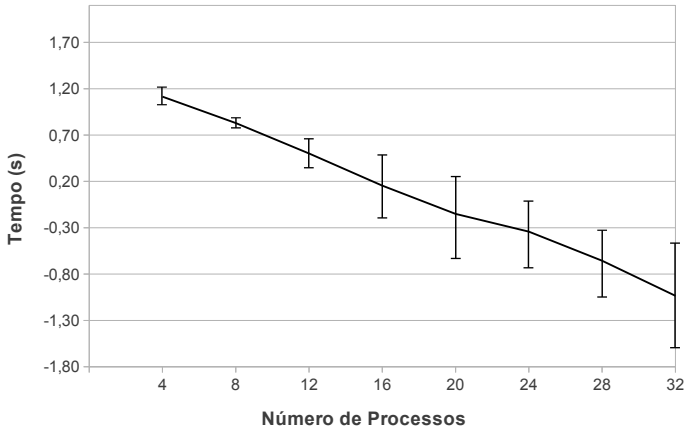
característica de nosso ambiente de teste (arquitetura computacional e interface de rede). Assim, por exemplo, este tempo poderia ser inferior ao utilizar-se uma interface de rede mais eficiente que a nossa *Gigabit Ethernet*. O que queremos destacar aqui, não é o valor absoluto, mas o reflexo da criação síncrona de todos os processos em um único `MPI_Comm_spawn`.

Já a Figura 21b apresenta as diferenças de tempo de execução encontradas no Cenário 2, ou seja, quando todas as comunicações passam pelo Mestre. Novamente, a diferença representa o quanto a versão MPI-1 foi mais rápida que a versão MPI-2. Nesta situação, o custo extra da criação dinâmica de processos foi atenuado pela criação de um processo a cada `MPI_Comm_spawn`, o qual é inferior ao da criação síncrona de processos em um único `MPI_Comm_spawn`. Nas condições expostas no Cenário 2, cada Trabalhador recebe sua fatia de trabalho tão logo seja inicializado. Logo, alguns Trabalhadores estarão realizando computação útil enquanto os demais ainda estão sendo criados. Isto inclusive

Figura 22 – Média da diferença dos tempos de execução para cada quantidade de processos testados



(a) MPI-1: Cenário 1 × Cenário 2



(b) MPI-2: Cenário 1 × Cenário 2

consegue atenuar o custo adicional do redirecionamento das mensagens passando pelo Mestre. Entretanto, conforme aumenta o número de processos à serem criados (em consequência, aumenta o número de chamadas ao `MPI_Comm_spawn`), este custo extra cresce de forma linear.

Através da Figura 22a, apresentamos a diferença entre as versões MPI-1, onde tempos o quanto a versão do Cenário 1 foi mais rápida que a do Cenário 2. Nesta circunstância, a diferença representa o quanto a versão em que as comunicações de borda ocorrem de forma independente entre os Trabalhadores (Versão 1 MPI-1) foi mais rápida que a versão em que as comunicações são intermediadas pelo Mestre (Versão 2 MPI-1 - ver Figura 16). Podemos notar que com o aumento no número de processos, ocorre também o aumento na diferença de tempo entre as versões. Isto porque o Mestre torna-se um gargalo, visto que conforme aumenta o número de Trabalhadores, também aumenta o número de redirecionamentos a serem realizados por ele. Isto não ocorre para o Cenário

1, a troca de dados ocorre de forma independente entre os processos.

Por fim, a média das diferenças de tempo entre as versões MPI-2 do Jogo da Vida são ilustradas na Figura 22b. Ela representa a diferença entre os tempos da versão com um único `MPI_Comm_spawn()` e comunicações de borda independentes entre os Trabalhadores (Versão 1 MPI-2) e a versão com múltiplos `MPI_Comm_spawn()` e comunicações intermediadas pelo Mestre (Versão 2 MPI-2 - ver Figura 18). Nestas circunstâncias, a criação de um único processo a cada `MPI_Comm_spawn` teve seu custo atenuado gradativamente até 16 processos. Isto representa que para até esta quantidade, o sincronismo da versão MPI-2 do Cenário 1 foi mais custoso do que o redirecionamento de mensagens realizado pelo Mestre no Cenário 2. Já a partir de 20 processos, a situação se inverte e o sincronismo da criação de múltiplos processos (o qual possui um valor estável em torno de $1,25$ segundos - ver Figura 21a) passa a ser menos custoso do que a quantidade de comunicações necessárias para o redirecionamento das mensagens pelo Mestre.

Pode-se notar também nas Figuras 21b, 22a e 22b, que quanto maior o número de processos, maior aumenta o desvio padrão. Este comportamento é explicado pelo fato da análise levar em consideração os tempos obtidos com todos os tamanhos de matrizes testados para cada quantidade de processos, ou seja, para a execução em uma matriz de tamanho 2048×2048 a diferença entre as versões fica em torno de 1 segundo, enquanto que para matrizes de tamanhos maiores, esta diferença aumenta proporcionalmente.

3.5 Conclusão do Capítulo

Este capítulo apresentou os resultados obtidos na execução de versões similares em MPI-1 e MPI-2 do Jogo da Vida. O objetivo desta análise foi mostrar o impacto da utilização da criação dinâmica de processos e do consequente modelo de comunicação hierárquico em nossa aplicação alvo. Para isso, avaliamos inicialmente dois métodos de distribuição de dados, buscando encontrar o mais eficiente para nossa aplicação alvo. Foram analisadas duas versões (Algoritmo de Divisão Igualitária de Linhas – ADIL e Algoritmo de Divisão por *Chunk* de Linhas – ADCL), em que a mais eficiente foi a ADIL, ou seja, a versão que distribui os dados (linhas da matriz) de forma igualitária entre os processos. Após encontrar a melhor distribuição, foi implementado uma versão Mestre/Trabalhadores em MPI-1 e foi adicionada a criação dinâmica de processos mantendo sempre as mesmas características do Jogo da Vida. Neste contexto surgiram duas versões MPI-1 e duas MPI-2.

Nas comparações envolvendo a criação de todos os processos em um único `MPI_Comm_spawn`, identificou-se um acréscimo constante de 1,25 segundos para a versão MPI-2 quando comparada com a versão MPI-1. Embora esta quantização reflita, em parte, a velocidade da rede de interconexão da plataforma de testes, ela também se deve ao

bloqueio do Mestre aguardando a criação de todos os processos. Percebemos que conforme aumenta-se o tamanho da entrada, reduz-se o percentual do impacto, uma vez que outros custos passam a ter maior influência. Um destes custos é o de comunicação, o qual aumenta naturalmente com o aumento da entrada, conforme mostrado na comparação entre as duas versões MPI-1.

As comparações quando implementa-se um `MPI_Comm_spawn()` para cada processo Trabalhador, mostraram que é possível atenuar o custo adicional proveniente da criação dinâmica. Assim, tão logo um Trabalhador seja criado ele já recebe sua parcela de dados à computar. Desta maneira, alguns Trabalhadores iniciam a computação enquanto os demais ainda estão sendo criados. Entretanto, a partir da criação de 20 processos, o custo proveniente do redirecionamento das mensagens realizado pelo Mestre passa a influenciar negativamente na eficiência da aplicação.

Observou-se também que no melhor dos casos, a criação dinâmica de processos impactou em apenas $0,2\%$ e $0,67\%$ para a criação de 4 e 32 processos respectivamente no tempo da aplicação quando comparada as versões MPI-1. Isto nos mostra um excelente resultado, visto que as aplicações MPI-2 foram implementadas ao modo de se programar MPI-1. Nosso próximo passo agora, é analisar o custo da criação dinâmica de processos na aplicação alvo *Skyline Matrix Solver*, a qual será apresentada na próxima seção.

4 Estudo de Caso II: *Skyline Matrix Solver*

Este capítulo objetiva estudar a criação dinâmica de processos na implementação paralela do problema *Skyline Matrix Solver*. Este estudo buscará identificar o impacto da criação síncrona de todos os processos em um único `MPI_Comm_spawn` para aplicações que possuem carga irregular de trabalho similares à esta. Adicionalmente, foi investigado o impacto da criação de um único processo por chamada ao `MPI_Comm_spawn`, investigando o envio assíncrono dos dados para os processos recém criados.

Para isto, inicialmente a Seção 4.1 contextualiza o problema alvo e suas principais características. Também é apresentado a implementação sequencial do Método de *Doolittle*, utilizado para computar o *Skyline Matrix Solver*. Na Seção 4.2 são apresentados os desafios encontrados durante a paralelização do problema alvo em sistemas de memória distribuída. Nesta seção, também é apresentada a organização geral da paralelização envolvendo MPI. As questões particulares das implementações utilizando MPI-1 e MPI-2 estão descritas na Seção 4.3, enquanto que a Seção 4.4 apresenta os resultados obtidos. Por fim, a Seção 4.5 conclui este capítulo.

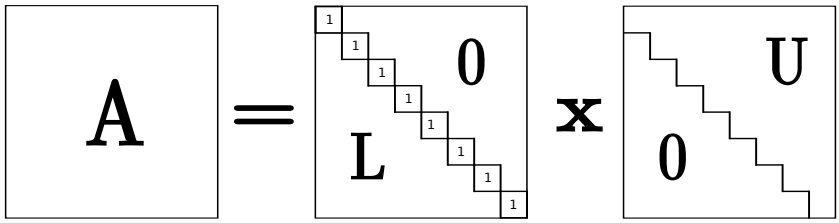
4.1 Descrição do Problema

Dada uma matriz A do tipo *skyline*, o problema *Skyline Matrix Solver* consiste em resolver a decomposição-LU (*Lower-Upper*) desta matriz. Uma matriz *skyline* é caracterizada por possuir uma quantidade significante de elementos nulos (zeros) distribuídos nas bordas da extremidade superior e à esquerda da matriz, sendo que a diagonal principal não possui elementos nulos (PRESS et al., 2002). A Figura 23 apresenta uma matriz

Figura 23 – Exemplo de uma Matriz *Skyline*.

r_i \ c_j	1	1	3	3	2	4	5	4
1	9	3	0	0	0	0	0	0
1	1	4	0	0	8	0	0	0
2	0	11	3	17	2	0	0	0
3	0	0	4	7	9	6	0	13
3	0	0	8	16	12	1	5	19
4	0	0	0	8	9	5	3	1
3	0	0	4	14	2	8	2	7
5	0	0	0	0	9	1	4	3

Figura 24 – Ilustração da operação $A = L * U$



skyline de tamanho 8×8, em que nela, os elementos nulos estão destacados com fundo em tom de cinza e os elementos da diagonal possuem borda mais espessa.

Ainda na Figura 23, c_j e r_i são denominadas as constantes que possuem a posição do primeiro elemento diferente de zero, em que c contém as posições das linhas e r das colunas. Por exemplo, vejamos a terceira coluna da matriz, em que $j = 3$, a posição do primeiro elemento não nulo é 3. Assim, o primeiro elemento diferente de zero da coluna 3 estará na linha 3. Estas constantes são utilizadas para o acesso direto aos elementos não nulos e possuem grande importância no cálculo da decomposição-LU (DICHTER; MAHMOOD; SHOLL, 1999).

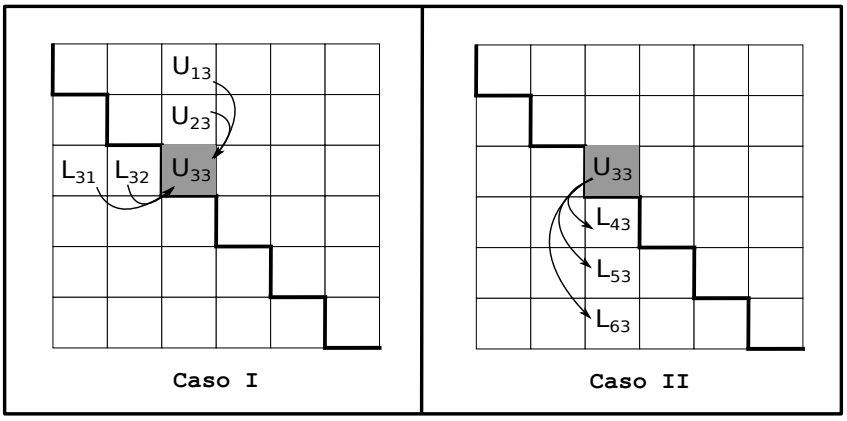
A decomposição-LU consiste em decompor a matriz de coeficientes A em duas matrizes L e U, sendo que o produto destas duas matrizes resulta na matriz A novamente. A Figura 24 apresenta a operação $A = L \times U$. Nela, a matriz L é representada pelos elementos situados abaixo da diagonal principal, e preenchida com números um na diagonal principal, enquanto que o restante da matriz é zerada. Já a matriz U corresponde aos elementos situados acima da diagonal principal, inclusive ela, enquanto que o restante da matriz é preenchida com zeros.

A decomposição da matriz A nas matrizes triangulares L e U consiste de um processo de sucessivos cálculos sobre os dados da matriz A. Ela segue operações específicas, as quais necessitam ser realizadas uma após a outra a fim de manter a integridade do resultado final. Desta forma, na seção seguinte, apresentamos as operações necessárias para decompor a matriz A e a dependência de dados existente. A seguir, apresentamos a implementação sequencial da decomposição-LU utilizando o Método de *Doolittle* na Seção 4.1.2.

4.1.1 Dependência de Dados

Para melhor entender a dependência de dados, ela é separada em três tipos: Dependência Diagonal, Dependência de Linha e Dependência de Coluna. O primeiro tipo, Dependência Diagonal, que envolve os elementos da diagonal, está ilustrada na Figura 25. Nela, o Caso I apresenta os elementos necessários para o cálculo do elemento da posição U_{33}, em que é necessário que os elementos da matriz *lower* L_{31} e L_{32} e *upper* U_{13} e U_{23}

Figura 25 – Dependência Diagonal



tenham sido previamente calculados. Já o Caso II contém as posições em que o elemento U_{33} será necessário. Assim, a computação dos elementos L_{43}, L_{53} e L_{63} só poderá ser realizada após a computação do elemento U_{33}.

Na Figura 26, é apresentado o segundo tipo de dependência, classificado como Dependência de Linha. Ela envolve as operações dos elementos situados acima da diagonal principal. Nela, o Caso I representa o cálculo do elemento da posição U_{34}. Para calcular este elemento, já devem ter sido calculados os elementos das posições L_{31}, L_{32}, U_{14} e U_{24} da matriz *lower* e *upper* respectivamente. De outro modo, o Caso II apresenta quais elementos dependem da posição U_{34} para calcula-los. Neste caso, U_{34} será utilizado no cálculo das posições U_{44}, L_{54} e L_{64}.

Por fim, o terceiro e último tipo é definido como Dependência de Coluna. Ela representa as operações envolvendo os elementos situados abaixo da diagonal principal. Nela, o Caso I compreende o cálculo do elemento da posição L_{53}. O cálculo deste elemento será possível somente após ter sido efetuado o cálculo das posições L_{51} e L_{52} da diagonal *lower* e os elementos das posições U_{13}, U_{23} e U_{33} da matriz *upper*. O Caso II representa quais posições necessitarão do elemento L_{53} para calcular, no caso, L_{54}, U_{55} e U_{56}.

Figura 26 – Dependência de Linha.

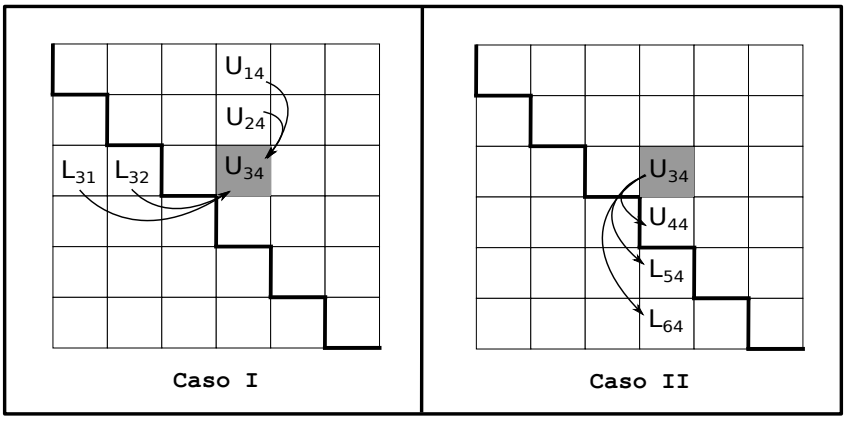
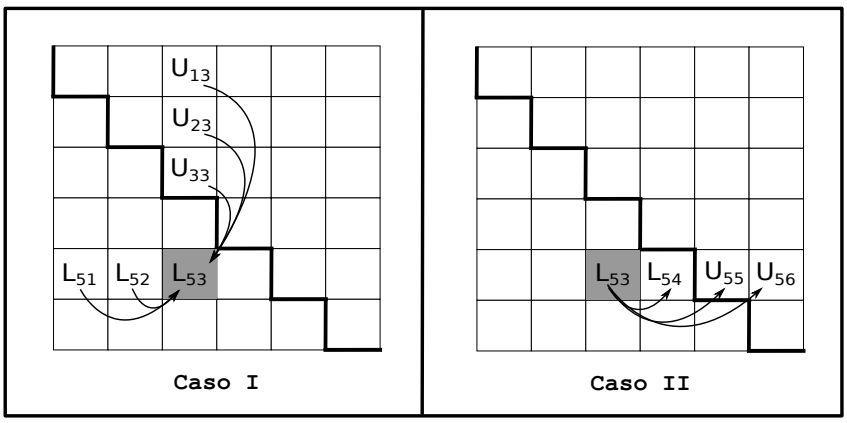


Figura 27 – Dependência de Coluna



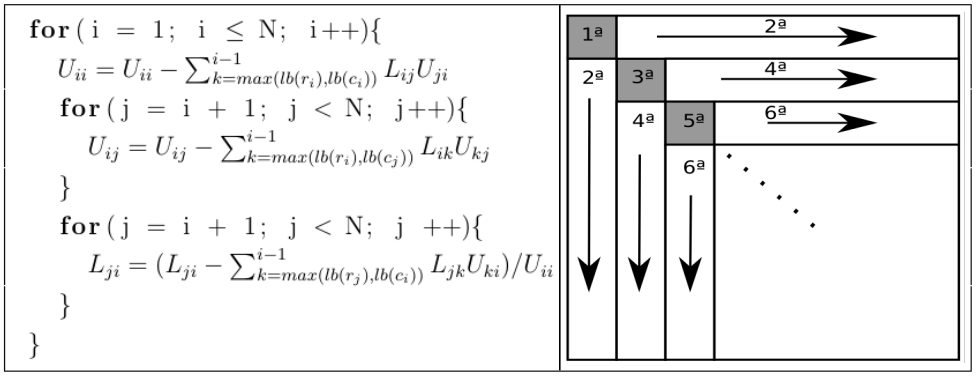
4.1.2 Método de *Doolittle*

O Método de *Doolittle*, descrito em Press et al. (2002) é um algoritmo muito utilizado para a decomposição-LU em matrizes do tipo *skyline*. Isto porque suas operações utilizam as constantes c_j e r_i para desconsiderar as posições com elementos nulos em seus cálculos, já que para estas posições o resultado será sempre zero.

O pseudocódigo do Método de *Doolittle* em sua versão sequencial é apresentado à esquerda da Figura 28. Nele, o laço de repetição (*for*) externo, com início na Linha 1 e finalização na Linha 9 percorre a diagonal principal da matriz da posição 1 até N (tamanho da matriz). A Linha 2 corresponde à equação que realiza o cálculo do elemento situado na diagonal principal da matriz (U_{ii}). Esta operação é ilustrada na Figura 25. A função $max(lb(r_i), lb(c_i))$ retorna a posição do primeiro elemento não-nulo (diferente de zero) da coluna ou linha necessária para realizar o cálculo, conforme armazenado nas constantes r_i e c_i. Esta função garante que o cálculo irá iniciar na primeira posição diferente de zero, não realizando o cálculo de elementos nulos.

Ainda no mesmo pseudocódigo, após realizar o cálculo do elemento da diagonal principal, o laço de repetição da Linha 3 percorrerá as posições correspondentes a linha

Figura 28 – Pseudocódigo do Método de *Doolittle* e Etapas da Execução



U_{ii+1} até a posição U_{iN-1}. Neste laço será realizado o cálculo dos elementos situados acima da diagonal principal, conforme ilustrado no Caso I da Figura 26. Novamente utiliza-se a função $max(lb(r_i), lb(c_i))$ para garantir que a computação ocorrerá somente sobre os elementos não nulos. Já o laço de repetição da Linha 6 percorre os elementos situados abaixo da diagonal principal entre as posições L_{i+1i} e L_{N-1i}. Como nos cálculos anteriores, garante que os elementos nulos não serão computados. Uma ilustração do cálculo de uma posição é ilustrado no Caso I da Figura 27.

Através do pseudocódigo, podemos perceber que existem duas etapas diferentes. Uma que calcula o elemento da diagonal principal e a outra que calcula os demais elementos (linha e coluna). Para auxiliar no entendimento da realização destas etapas, a matriz à direita da Figura 28 apresenta a sequencia das operações realizadas pelo método de *Doolittle*. Nela, a 1ª etapa realiza o cálculo do elemento da diagonal principal (equação da Linha 2 do pseudocódigo). O valor obtido será necessário para realizar o cálculo da 2ª etapa (equações das Linhas 4 e 6), logo, ela só poderá iniciar quando a etapa anterior ter sido concluída. O mesmo ocorre para as próximas etapas.

Analisando o pseudocódigo sequencial do Método de *Doolittle*, identifica-se oportunidades de paraleliza-lo. Uma destas oportunidades corresponde a execução concorrente das etapas pares, ou seja, etapa 2, 4, 6 e assim sucessivamente. Por exemplo, o cálculo da segunda etapa pode ser dividido de forma que se compute os elementos da linha e coluna simultaneamente. Sendo assim, a próxima seção apresenta a paralelização do *Skyline Matrix Solver* destacando os principais desafios a serem vencidos durante a paralelização e a organização geral da nossa proposta de paralelização.

4.2 Paralelização do *Skyline Matrix Solver*

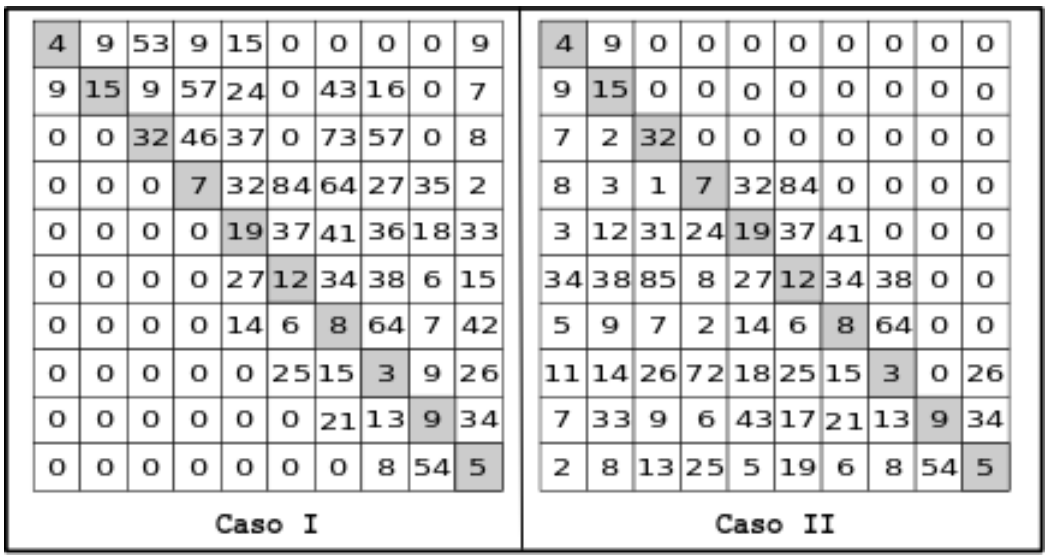
4.2.1 Desafios para a Paralelização

O *Skyline Matrix Solver* é considerado uma aplicação que possui carga de trabalho irregular devido às características da matriz *skyline* e do Método de *Doolittle*. Desta forma, existem dois desafios principais à serem vencidos durante a paralelização do problema em sistemas de memória distribuída: Prover Balanceamento da Carga de Trabalho e Comunicação Eficiente. As subseções seguintes apresentam estes dois desafios.

4.2.1.1 Prover Balanceamento da Carga de Trabalho

Conforme apresentado na Seção 4.1, uma matriz *skyline* é caracterizada por possuir elementos nulos (zeros) situados nas bordas da extremidade superior e à esquerda da matriz. Portanto, o método de *Doolittle* tende a ser eficiente na decomposição da matriz A, pois não realiza o cálculo das posições que contêm elementos nulos (BOUMAN, 1995).

Figura 29 – Exemplo de matriz *skyline* desbalanceada de tamanho 10 × 10



4	9	53	9	15	0	0	0	0	9
9	15	9	57	24	0	43	16	0	7
0	0	32	46	37	0	73	57	0	8
0	0	0	7	32	84	64	27	35	2
0	0	0	0	19	37	41	36	18	33
0	0	0	0	27	12	34	38	6	15
0	0	0	0	14	6	8	64	7	42
0	0	0	0	0	25	15	3	9	26
0	0	0	0	0	0	21	13	9	34
0	0	0	0	0	0	0	8	54	5

Caso I

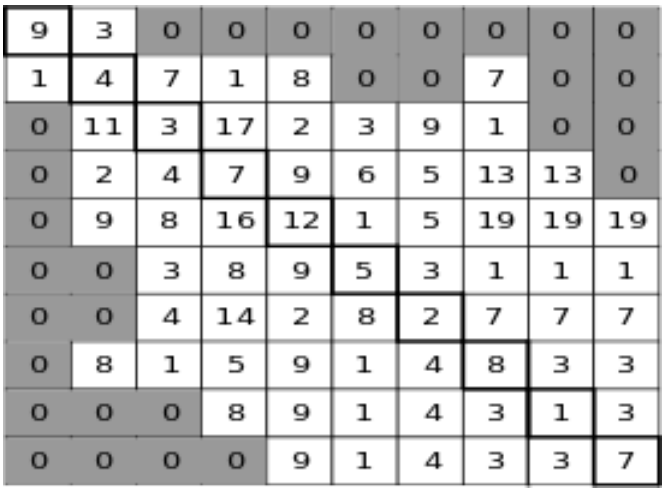
4	9	0	0	0	0	0	0	0	0
9	15	0	0	0	0	0	0	0	0
7	2	32	0	0	0	0	0	0	0
8	3	1	7	32	84	0	0	0	0
3	12	31	24	19	37	41	0	0	0
34	38	85	8	27	12	34	38	0	0
5	9	7	2	14	6	8	64	0	0
11	14	26	72	18	25	15	3	0	26
7	33	9	6	43	17	21	13	9	34
2	8	13	25	5	19	6	8	54	5

Caso II

Para melhor entender o balanceamento da carga de trabalho, a Figura 29 apresenta duas matrizes *skyline* de tamanho 10 × 10. No Caso I, a maioria dos elementos nulos estão situados abaixo da diagonal principal (destacada com a cor cinza). Logo, o cálculo envolvendo a computação dos elementos da matriz L, será realizado rapidamente. Por outro lado, a computação dos elementos da matriz U, levará um tempo consideravelmente maior. Assim, neste caso, uma das opções possíveis seria focar na distribuição dos elementos da matriz U, já que ela possuirá maior tempo de computação. Ainda na Figura 29, o Caso II apresenta exatamente o contrário, em que a maioria dos elementos nulos situam-se acima da diagonal principal. Logo, a computação dos elementos da matriz L levaria um tempo consideravelmente maior do que a computação da matriz U.

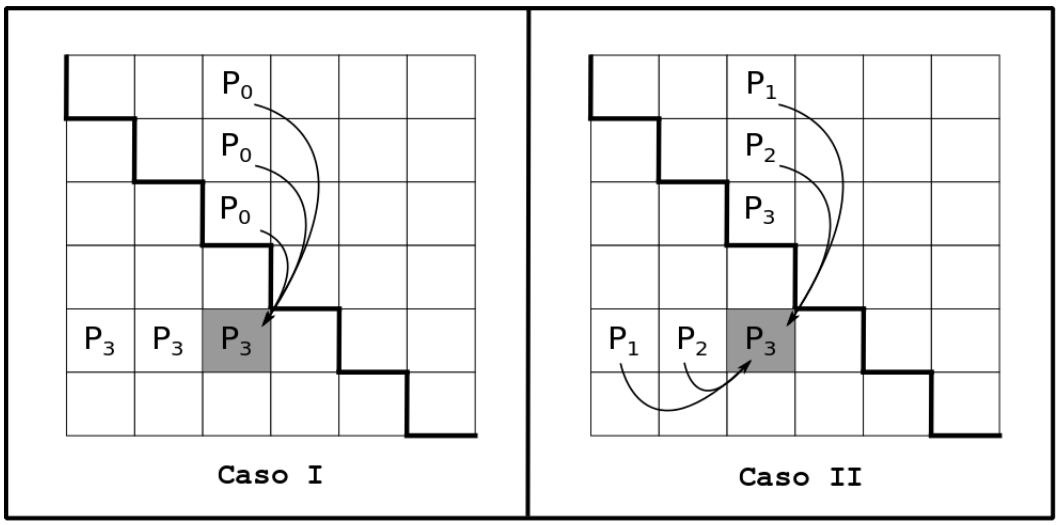
Este trabalho realizará todos os testes em uma matriz que possui os elementos distribuídos uniformemente. Um exemplo desta matriz é encontrado na Figura 30. Nela,

Figura 30 – Exemplo da matriz *skyline* uniforme



9	3	0	0	0	0	0	0	0	0
1	4	7	1	8	0	0	7	0	0
0	11	3	17	2	3	9	1	0	0
0	2	4	7	9	6	5	13	13	0
0	9	8	16	12	1	5	19	19	19
0	0	3	8	9	5	3	1	1	1
0	0	4	14	2	8	2	7	7	7
0	8	1	5	9	1	4	8	3	3
0	0	0	8	9	1	4	3	1	3
0	0	0	0	9	1	4	3	3	7

Figura 31 – Exemplo de comunicação de dados



a matriz de tamanho 10×10 possui 30% de densidade, ou seja, dos 100 elementos totais da matriz, 30 destes são nulos e estão distribuídos uniformemente entre as posições acima e abaixo da diagonal principal.

4.2.1.2 Prover Comunicação Eficiente

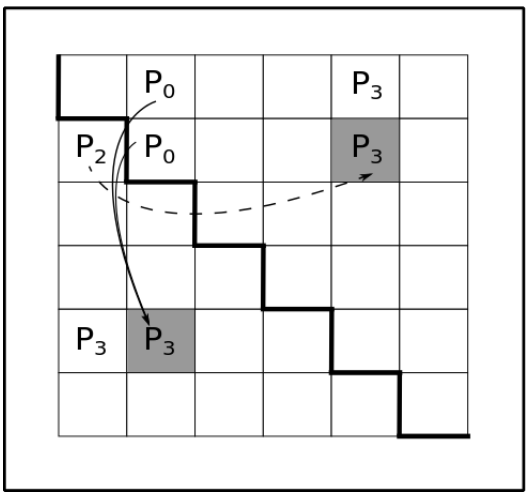
A paralelização do *Skyline Matrix Solver* em sistemas de memória distribuída faz com que cada processo mantenha uma cópia local atualizada dos dados utilizados para sua computação. Assim, entre cada etapa à ser realizada em paralelo, deve existir a troca de dados entre os processos, e a consequente atualização da matriz local de cada processo. Esta atualização é necessária, pois como apresentado na Seção 4.1.1, exceto para a primeira linha e coluna, a computação de qualquer elemento depende dos dados computados em iterações anteriores.

A comunicação eficiente dos dados do problema *Skyline Matrix Solver* depende de vários fatores, entre eles:

- **Particionamento dos Dados da Matriz:** Conforme Rauber e Rünger (2010), a matriz de entrada pode ser particionada em vetores de uma dimensão (vetor de linhas ou colunas), e em vetores de duas dimensões (blocos de tamanho $B \times N$) que contém os elementos de linha e coluna. De acordo com o modelo de particionamento utilizado, a quantidade de comunicações difere, pois em vetores unidimensionais haverão mais comunicações com menos dados, enquanto que na divisão bidimensional haverão menos comunicações, porém com maior quantidade de dados.

- **Distribuição dos Dados:** A grande dependência de dados existente na decomposição-LU (ver Seção 4.1.1) e a forma com que os dados são distribuídos entre os processos possuem grande influência na eficiência da paralelização do *Skyline Ma-*

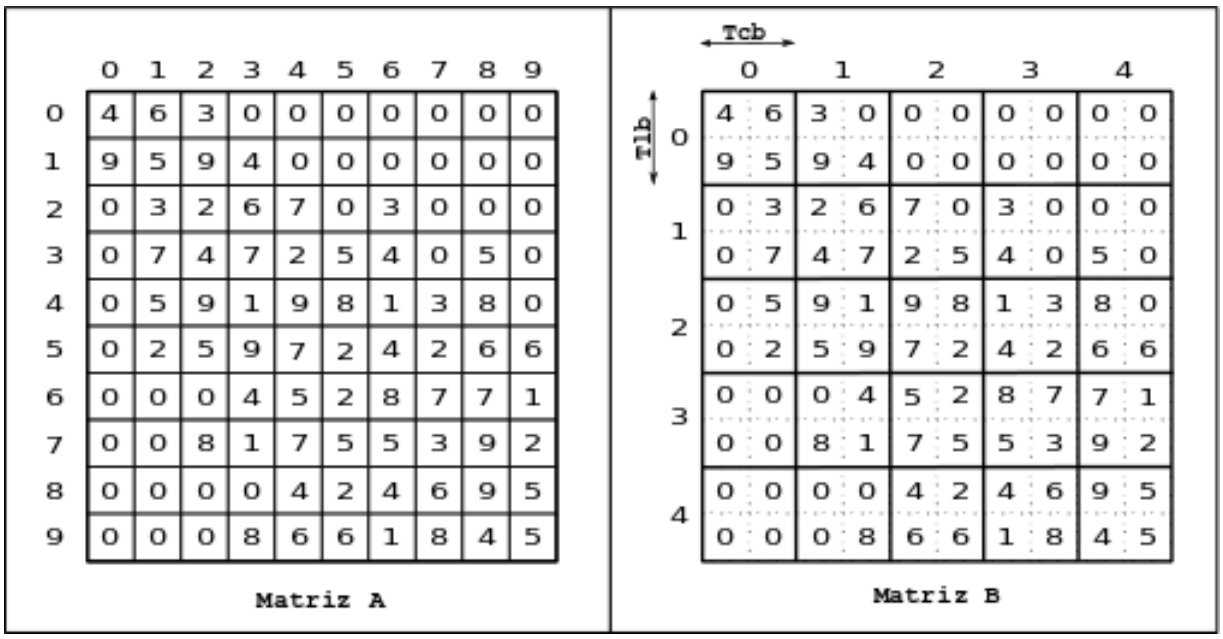
Figura 32 – Exemplo de utilização da comunicação assíncrona



trix Solver. Isto porque, uma distribuição boa dos dados implica em um número menor de comunicações. A Figura 31 ajuda a entender este raciocínio, onde o caso I representa uma boa divisão. Nele, para o processo P_3 computar seu elemento preenchido pela cor cinza, necessitará de dados provenientes das colunas computadas pelo processo P_0 (setas representam as comunicações necessárias). Os elementos anteriores da linha computados pelo processo P_3 já estão armazenados em sua memória local, portanto não necessita comunicação para estes dados. Neste caso, o cálculo envolveu dados provenientes de um único processo. Já o Caso II apresenta as comunicações necessárias para o cálculo da mesma posição pelo processo P_3, porém os dados necessários estão alocados na memória local dos processos P_1, P_2 e inclusive P_3. Assim, será necessário comunicar-se com os processos P_1 e P_2 para atualizar seus dados, exigindo maior número de comunicações.

- **Modelo de Comunicação:** Conforme a dependência de dados apresentada na Seção 4.1.1, a paralelização do Método de *Doolittle* envolve grande quantidade de comunicação entre os processos. O sobre-custo causado por estas comunicações pode ser compensado através do envio e recebimento assíncrono dos dados. Assim, enquanto o processo está recebendo ou enviando os dados, ele pode realizar computações. A Figura 32 apresenta um exemplo desta situação. Nela, o processo P_3 irá computar sobre dois elementos diferentes, preenchidos pela cor cinza. As setas indicam o envio dos elementos pelos processos P_0 e P_2 para P_3. Tão logo P_3 receba os dados vindos do processo P_0, ele já pode computar sobre estes dados enquanto recebe de forma assíncrona (seta pontilhada) o elemento de P_2. Assim, quando terminar de computar sobre o primeiro elemento, P_3 já possuirá em sua memória local os elementos para computar o segundo bloco.

Figura 33 – Ilustração do mapeamento da matriz A para uma matriz em blocos



4.2.2 Organização Geral da Paralelização

Esta seção apresenta as características comuns à paralelização com MPI-1 e MPI-2. Sendo assim, apresentaremos como a matriz de entrada A será particionada (linhas e colunas ou em blocos), a forma com que os dados serão atribuídos aos processos e o consequente modelo de comunicação utilizado.

4.2.2.1 Particionamento da Matriz

O particionamento da matriz de entrada A se deu utilizando o modelo de divisão de dados em blocos. Desta forma, a matriz A será mapeada para uma matriz de blocos. Optamos por este modelo, pois, conforme apresentado na Seção anterior, ele requer menor quantidade de comunicações.

O mapeamento de uma matriz A de tamanho $N \times N$ para uma matriz B de tamanho $M \times M$ é ilustrado na Figura 33. Nela, uma matriz 10×10 foi mapeada para uma matriz de blocos 5×5 em que cada bloco possui tamanho 2×2. As variáveis `Tcb` e `Tlb` representam o total de colunas e linhas de cada bloco respectivamente, em que $Tcb = Tlb$.

A forma com que os dados são computados na matriz B é similar a computação da matriz A. O Caso I da Figura 34 apresenta a computação do primeiro bloco da diagonal, denominado B_{00}. A dependência de dados apresentada na Seção 4.1.1 continua sendo válida para este tipo de divisão de matriz. Assim, a computação do bloco B_{00} inicia pela posição da diagonal principal A_{00}. A seguir ocorre o cálculo dos demais elementos da

linha e coluna, neste caso A_{01} e A_{10}, e por fim, ocorre a computação do último elemento, A_{11}.

Ainda na Figura 34, o Caso II apresenta o cálculo do bloco B_{14}, que é um bloco situado acima da diagonal principal. Para computá-lo, será necessário os blocos B_{10}, B_{04} e B_{11}. Assim, para computar a posição A_{38}, será necessário ter acesso aos elementos A_{30}, A_{31}, A_{32}, A_{08} e A_{18} dos blocos B_{10} e B_{04} respectivamente. O Cálculo dos elementos dos blocos situados abaixo e acima da diagonal principal segue a mesma sequencia de computação apresentada no Caso I.

Após definir a forma com que a matriz de entrada será particionada, a seguir será apresentado como é realizada a distribuição destes blocos de dados entre os processos.

4.2.2.2 Divisão da Carga de Trabalho entre os Processos

A divisão da carga de trabalho entre os processos está relacionada diretamente às características da matriz e do Método de *Doolittle*. Conforme apresentado, a aplicação possui carga de trabalho irregular, o que pode ser conferido nas matrizes da Figura 34. Portanto, a divisão dos blocos entre os processos deve levar em consideração esta característica, com a finalidade de encontrar o melhor balanceamento de carga de trabalho. Importante lembrar que a matriz utilizada em nossos casos de teste segue uma distribuição uniforme dos elementos nulos conforme a Figura 30.

O modelo de programação utilizado na paralelização deste problema foi o Mestre/Trabalhador, em que o processo Mestre realizará a computação dos elementos dos blocos diagonais, enquanto que os Trabalhadores computarão sobre os demais blocos. Sendo assim, a Figura 35 apresenta como é realizada a distribuição da carga de trabalho entre os processos. Nela, a matriz de blocos de tamanho 5×5 foi mapeada para 5 processos, em que P_0 representa o Mestre, enquanto que os demais (P_1, P_2, P_3 e P_4) são os

Figura 34 – Ilustração da computação sobre um bloco

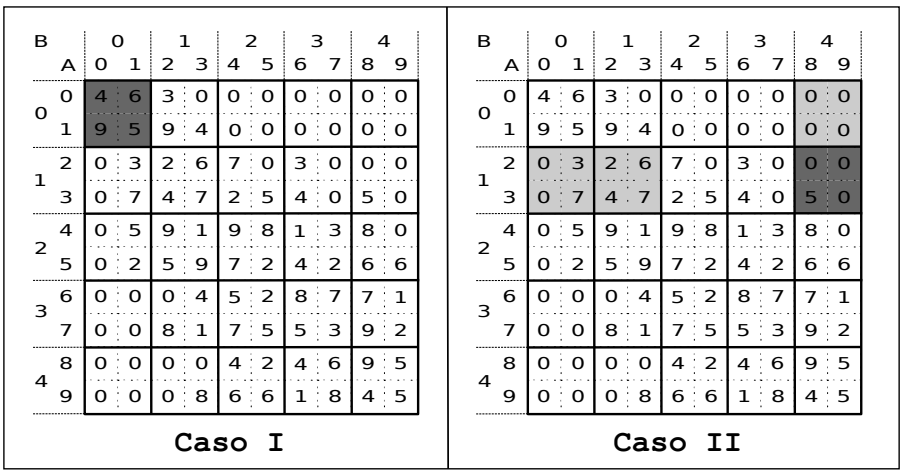
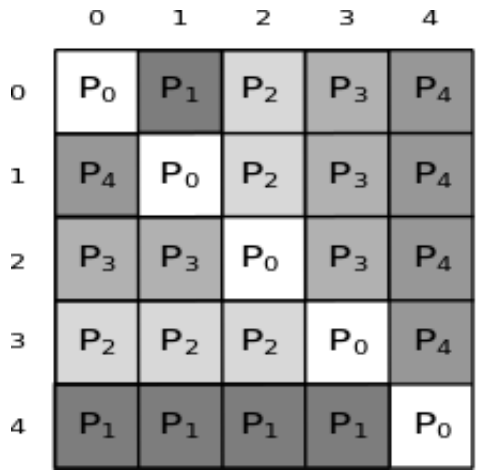


Figura 35 – Ilustração da divisão da carga de trabalho



Trabalhadores.

Este modelo contempla a divisão balanceada dos dados, em que cada processo será responsável por calcular a mesma quantidade de blocos. Podemos notar na Figura 35 que cada processo computará 5 blocos, os quais são atribuídos de acordo com a característica da matriz. Por exemplo: o bloco B_{01} mapeado para P_1 possui pouca carga de computação, enquanto que a posição B_{43} atribuída para o mesmo processo possui grande carga de computação. O mesmo ocorre para os demais processos Trabalhadores, em que o processo que receber uma carga de trabalho menor abaixo da diagonal principal, receberá uma carga de trabalho maior acima da diagonal principal.

4.2.2.3 Estrutura da Aplicação Paralela

Através do modelo proposto de distribuição da carga de trabalho entre os processos, identificou-se duas diferentes funções à serem executadas pelos processos: a função **calculaDiagonal**, que é computada pelo Mestre e contém as operações relacionadas ao cálculo dos blocos diagonais da matriz; e a função **calculaLU**, que é computada pelos Trabalhadores e contém as operações relacionadas ao cálculo dos blocos situados acima e abaixo dos blocos da diagonal principal.

O pseudocódigo da Figura 36 apresenta as operações realizadas na função **calculaDiagonal**. Nela, inicialmente ocorre o carregamento da matriz e o seu mapeamento para uma matriz de blocos. A quantidade de blocos por linha é relacionada a quantidade de Trabalhadores. No exemplo da Figura 35, a matriz A 10×10 computada por 5 Trabalhadores, foi mapeada para uma matriz B 5×5, em que cada bloco possui tamanho 2×2. Em seguida, após computar o bloco diagonal, o Mestre enviará o resultado mais os blocos necessários para os processos Trabalhadores computarem através da diretiva `MPI_Send`. Logo após finalizar o envio para cada Trabalhador, ele executa duas vezes a diretiva `MPI_Irecv` para aguardar os blocos linha e coluna atualizados (linha 5).

Na linha 6, o Mestre aguardará através da diretiva `MPI_Waitany` os dois blocos necessários para calcular o próximo bloco da diagonal. Após calcula-lo, o Mestre aguardará o recebimento do restante dos blocos da linha computados pelos Trabalhadores que executam a função **calculaLU** (correspondente ao `MPI_Irecv` executado na linha 5). Para cada recebimento de linha, o Mestre verifica se existem dados à serem enviados para aquele Trabalhador. Sempre que existirem dados, o Mestre enviará assincronamente. Estas operações são repetidas enquanto houverem blocos linha à serem recebidos pelo Mestre (laço *for* com início na linha 10 e fim na linha 16).

Já o recebimento do restante dos blocos da coluna computados ocorre na linha 18 (correspondente ao `MPI_Irecv` executado na linha 5). As mesmas operações realizadas para o recebimento dos dados das linhas serão repetidos para o recebimento dos dados das colunas. As operações descritas entre as linhas 7 e 24 são repetidas enquanto houverem etapas à serem computadas.

A função **calculaLU** tem seu pseudocódigo descrito na Figura 37. Nela, o recebimento dos primeiros blocos vindos da função **calculaDiagonal** ocorre forma síncrona. Utilizou-se este tipo de comunicação, pois neste caso, o processo só possuirá carga de trabalho para computar quando finalizar o recebimento de todos os blocos. De posse de todos os dados, o processo computará sobre os blocos que lhe foram atribuídos (linha 3). Assim que finalizar a computação do primeiro bloco, realiza o envio dele para o processo Mestre que já executou um `MPI_Irecv` para esperá-lo. Enquanto realiza o envio deste primeiro bloco através de comunicação assíncrona, o Trabalhador computa sobre o segundo bloco que lhe foi atribuído. Enquanto houver computação à ser realizada, ou seja, blocos à serem computados, o Trabalhador receberá de forma assíncrona estes dados do

Figura 36 – Pseudocódigo da função calculaDiagonal

```
void CalculaDiagonal(int nProcSum, int nProcLU){
1.  Carrega matriz e mapeia para blocos;
2.  Computa blocoDiagonal;
3.  for(p=0;p<nProcLU;p++){
4.    Envios de blocos para calculaLU;
5.    Recebimento assíncrono dos blocos de calculaLU;
6.  }
7.  do{
8.    Aguarda blocos para computar a diagonal;
9.    Computa blocoDiagonal;
10.   for(i=0;i<qtdRecvLinha;i++){
11.         Aguarda blocos linha de calculaLU;
12.       if(existe computacao){
13.           Envios de blocos para calculaLU;
14.           Recebimento assíncrono de calculaLU;
15.       }
16.   }
17.   for(i=0;i<qtdRecvColuna;i++){
18.       Aguarda blocos coluna de calculaLU;
19.       if(existe computacao){
20.           Envios de blocos para calculaLU;
21.           Recebimento assíncrono de calculaLU;
22.       }
23.   }
24. }while(houver computacao);
25.}
```

processo Mestre. Neste ponto da aplicação, optou-se por este tipo de comunicação pois o Trabalhador poderá receber rapidamente um bloco atualizado e ir computando enquanto aguarda o recebimento do restante dos blocos.

Com a finalidade de encontrar o tempo aproximado da computação de cada bloco e propor alternativas na paralelização, realizamos testes preliminares da aplicação (com as duas funções **calculaDiagonal** e **calculaLU**). Estes testes compreenderam a execução com 4 processos computando sobre uma matriz de teste 5000×5000, o qual permitiu analisar a escalabilidade da aplicação. Desta forma, seguindo o modelo de particionamento da matriz proposto anteriormente, cada bloco possui tamanho de 1250×1250. A aplicação foi executada 10 vezes sobre a mesma arquitetura utilizada nos testes da Seção 3.2.3.

Percebe-se, com os resultados obtidos, que o tempo de computação dos blocos de cada etapa é diferente. Na Figura 38 estes tempos são apresentados, em que a computação de cada bloco da 1ª e 2ª etapa possuem tempo aproximado de 10 segundos. Este tempo aumenta consideravelmente para a computação de cada bloco das demais etapas, chegando ao tempo aproximado de 100 segundos para computar o bloco da última etapa (cálculo do último bloco da diagonal principal). Este comportamento é explicado pelo fato de que quanto mais próximo da última etapa, maior será a quantidade de operações realizadas. Por exemplo, na Figura 38, para computar o bloco B_{11}, será realizada a seguinte operação:

$$B_{11} = B_{11} - (B_{10} * B_{01})$$

enquanto que a computação do bloco B_{33} envolverá a seguinte operação:

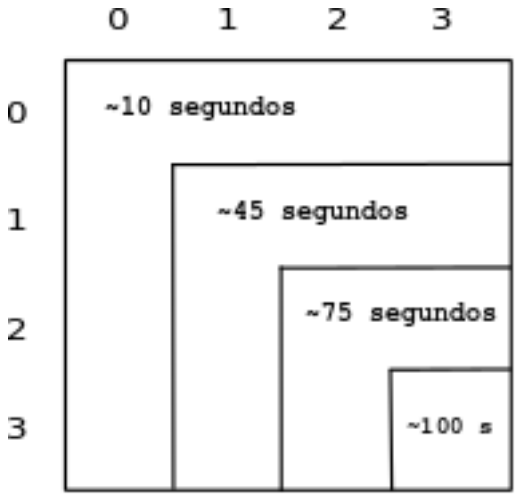
$$B_{33} = B_{33} - [(B_{30} * B_{03}) + (B_{31} * B_{13}) + (B_{32} * B_{23})]$$

Assim, com a intenção de reduzir o tempo de computação do bloco da diagonal, foi implementada uma nova função chamada **calculaSomatorioLU**, que objetiva realizar o somatório preliminar dos dados utilizados para computar o bloco diagonal. Portanto, a computação dos elementos do bloco B_{33} apresentada acima, com a inclusão desta nova função, ficará como segue:

Figura 37 – Pseudocódigo da função calculaLU

```
void calculaLU(){
1. Recebimento síncrono dos blocos de calculaDiagonal;
2. do{
3.     Computa blocoLinha e blocoColuna;
4.     Envios dos blocos linha e coluna atualizados para calculaDiagonal;
5.     if(houver computacao)
6.        Recebimento assíncrono dos blocos de calculaDiagonal;
7.  }while(houver computacao);
8.}
```

Figura 38 – Tempos de execução para computação por bloco em cada etapa da aplicação



$$B_{33} = B_{33} - [calculaSomatorioLU(B_{33}) + (B_{32} * B_{23})]$$

em que **calculaSomatorioLU(B_{33})** corresponde à:

$$calculaSomatorioLU(B_{33}) = (B_{30} * B_{03}) + (B_{31} * B_{13})$$

Com a inclusão desta nova função na aplicação, a função **calculaDiagonal** foi alterada e passa agora a enviar os blocos para a função **calculaSomatorioLU** auxiliar

Figura 39 – Pseudocódigo da função calculaDiagonal com a inclusão da função calculaSomatorioLU

```
void CalculaDiagonal(int nProcSum, int nProcLU){
1.    Carrega matriz e mapeia para blocos;
2.    Computa blocoDiagonal;
3.    for(p=0;p<nProcLU;p++){
4.      Envios de blocos para calculaLU;
5.      Recebimento assíncrono dos blocos de calculaLU;
6.    }
7.    do{
8.      Aguarda blocos para computar a diagonal;
9.      if(existe dados do somatorio)
10.          Recebe o somatório de calculaSomatorioLU;
11.     Computa blocoDiagonal;
12.     for(i=0;i<qtdRecvLinha;i++){
13.          Aguarda blocos linha de calculaLU;
14.          Envio do bloco linha para calculaSomatorioLU;
15.          if(existe computacao){
16.             Envios de blocos para calculaLU;
17.             Recebimento assíncrono de calculaLU;
18.          }
19.      }
20.      for(i=0;i<qtdRecvColuna;i++){
21.          Aguarda blocos coluna de calculaLU;
22.          Envio de bloco coluna para calculaSomatorioLU;
23.          if(existe computacao){
24.             Envios de blocos para calculaLU;
25.             Recebimento assíncrono de calculaLU;
26.          }
27.      }
28.   }while(houver computacao);
29.}
```

Figura 40 – Pseudocódigo da função calculaSomatorioLU

```
void calculaSomatorioLU(){
1.  do{
2.      Recebimento assíncrono dos blocos de calculaDiagonal;
3.      Computa somatório dos blocos;
4.  }while(houver computacao);
5.  Envio do somatório dos blocos para calculaDiagonal;
6.}
```

na computação do bloco da diagonal. Assim, a função **calculaDiagonal** alterada tem seu pseudocódigo descrito na Figura 39. Nele, na linha 9, antes do processo Mestre computar o bloco diagonal, ele verifica se existe somatório pré-calculado pela função **calculaSomatorioLU**, e caso exista, recebe-o através do `MPI_Recv`. O envio dos blocos para o processo da função **calculaSomatorioLU** (linha 14 e 22) ocorre assim que o Mestre os receba da função **calculaLU**. Este envio se dá através do uso de comunicações assíncronas, para que o processo Mestre continue executando suas operações enquanto envia os dados para este Trabalhador.

O pseudocódigo da função **calculaSomatorioLU** é apresentado na Figura 40. As operações dos Trabalhadores que utilizarem esta função compreendem em receber os blocos vindos da função **calculaDiagonal** através de comunicações assíncronas (`MPI_Irecv`) e computar o somatório do produto dos elementos dos blocos. Esta operação de receber e computar é repetido enquanto houverem dados para serem computados. Quando não houverem mais blocos para serem calculados, o Trabalhador retorna para a função **calculaDiagonal** o somatório dos blocos computados.

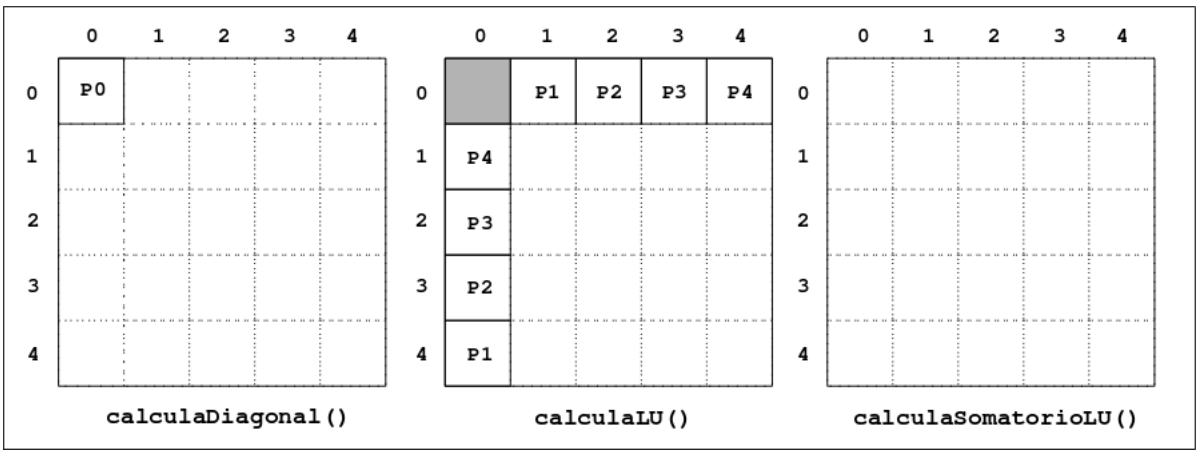
A partir deste momento, os processos serão classificados em: **Processos principais**, que correspondem aos processos que computarão as operações das funções **calculaDiagonal** e **calculaLU**. Destes, 1 sempre representará o Mestre e os demais os Trabalhadores; e **Processos auxiliares**, que correspondem aos processos Trabalhadores que executarão a função **calculaSomatorioLU**.

Para auxiliar no entendimento de como a aplicação está estruturada, a Seção seguinte contém um exemplo didático.

4.2.2.4 Exemplo da Paralelização

Como um exemplo didático, a seguir são apresentadas as etapas realizadas durante a execução sobre uma matriz A de tamanho 20×20 com 7 processos. Destes, 5 processos serão os principais, e 2, os processos auxiliares. Assim, mapeando a matriz de entrada para uma matriz de blocos B, ela possuirá tamanho 5×5. As Figuras 41, 42, 43, 44 e 45 apresentam a evolução das etapas. Nelas, os blocos que estiverem com preenchimento da cor cinza já foram computados. Os blocos que estiverem com a identificação do processo

Figura 41 – Ilustração do cálculo da decomposição-LU – Etapa 1



(por exemplo: Px), significa que está ocorrendo computação sobre estes blocos, e por fim os blocos destacados com borda pontilhada indicam que não está ocorrendo qualquer tipo de computação sobre eles.

Na Figura 41 ilustramos a primeira etapa da aplicação. Nela, inicialmente o processo $P0$ (executando a função **calculaDiagonal**) calculará o bloco B_{00}. Em seguida, enviará os blocos para os processos da função **calculaLU** através das diretivas `MPI_Send()` combinada com o `MPI_Recv()`. Ao final de cada envio, o processo $P0$ executa uma diretiva `MPI_Irecv()` para cada bloco de linha/coluna enviado. Quando os processos Trabalhadores da função **calculaLU** finalizarem o recebimento dos blocos, computarão sobre seus blocos. Por exemplo, o processo $P4$ realizará a computação dos blocos B_{04} e B_{10}. Para isto, de acordo com a dependência de dados apresentada na Seção 4.1.1, o cálculo destes blocos depende dos blocos anteriores à ele, B_{00}, B_{04} e B_{10}. Assim, será necessário enviar os blocos B_{00}, B_{04} e B_{10} à este processo. Nesta etapa, os processos da função **calculaSomatorioLU** não computarão, pois ainda não existe carga de trabalho atribuída à eles.

Após finalizar o envio de todos os blocos, o processo $P0$ aguarda o retorno dos

Figura 42 – Ilustração do cálculo da decomposição-LU – Etapa 2

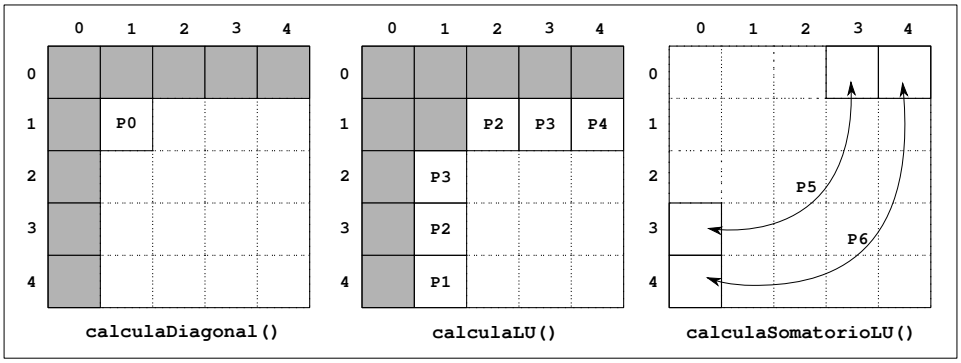
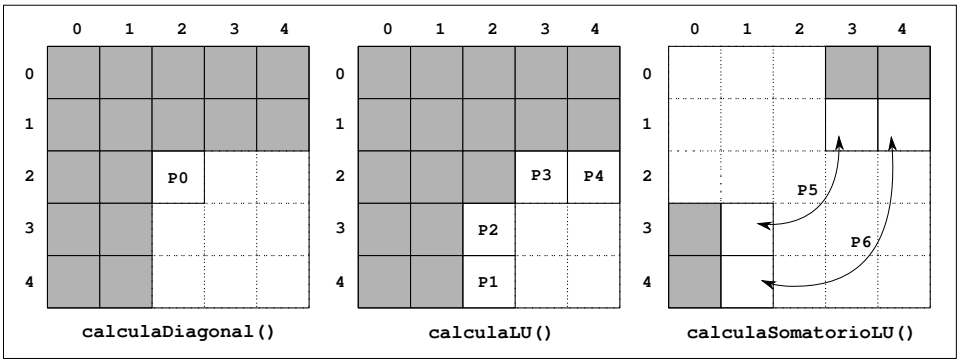


Figura 43 – Ilustração do cálculo da decomposição-LU – Etapa 3



blocos através da diretiva `MPI_Waitany()`. A aplicação prioriza o recebimento dos blocos necessários para calcular o bloco da diagonal principal. Já os demais blocos, serão recebidos enquanto o processo $P0$ computa o bloco da diagonal. Desta forma, consegue-se computar enquanto ocorre o recebimento assíncrono dos dados.

Na etapa 2, ilustrada na Figura 42, tão logo o processo $P0$ receba os dados das posições B_{10} e B_{01}, ele calcula o bloco diagonal B_{11}. A seguir, aguarda o recebimento dos demais blocos atualizados (B_{02}, B_{03}, B_{04}, B_{20}, B_{30} e B_{40}) da função **calculaLU**. Assim que receber qualquer destes blocos vindo de qualquer processo, $P0$ verifica se este bloco será utilizado para calcular algum bloco da diagonal futuramente. Se isto ocorrer e houverem processos mapeados para auxiliar no cálculo desta diagonal, ocorre o envio deste bloco para o processo da função **calculaSomatorioLU**. Por exemplo, $P0$ recebe o bloco B_{03}. Este bloco será necessário para o cálculo do bloco diagonal B_{33}. Assim, $P0$ verifica em uma estrutura auxiliar se existe processo definido para auxiliar no cálculo desta posição. Neste caso, o processo $P5$ será o responsável por auxiliar neste cálculo. Desta forma, $P0$ envia o bloco B_{03} e B_{30} tão logo os tenha recebido. Outra situação que pode ocorrer, é $P0$ receber blocos que não possuam nenhum processo mapeado, que é o caso dos blocos B_{02} e B_{20}. Assim, o cálculo do bloco diagonal B_{22} será realizada somente por $P0$. Esta operação é realizada enquanto houverem processos aguardando por blocos de dados na função **calculaSomatorioLU**.

Quando $P0$ envia os blocos para os processos da função **calculaSomatorioLU**, ele também envia os blocos necessários para os processos da função **calculaLU**. Por exemplo, o processo $P3$ retornou os blocos B_{03} e B_{30}, logo, $P0$ verifica se existem processos que necessitam destes blocos para calcular suas posições, e caso exista, redireciona estes blocos aos respectivos processos. De outro modo, quando $P0$ receber os blocos necessários para $P3$ continuar a computação, os blocos são redirecionados para $P3$. Neste caso, $P3$ receberá os blocos B_{10}, B_{11} e B_{13} para computar o bloco B_{13} (não recebe B_{03}, pois o mesmo já se encontra em sua memória local), e os blocos B_{01} e B_{21} para computar o bloco B_{21} (o bloco diagonal B_{11} só é recebido uma única vez). O mesmo ocorre para os demais processos da

função **calculaLU**. Após a computação, o retorno dos dados ocorre da mesma forma que a etapa anterior.

Ao final da etapa 3, o processo $P5$ terá concluído sua função de auxiliar na computação do bloco diagonal. Desta forma, o resultado obtido do somatório dos blocos B_{30}, B_{03}, B_{31} e B_{13} será retornado para o processo $P0$ utilizar na computação do bloco B_{33} na etapa 4.

Na etapa 4, ilustrada na Figura 44, inicialmente $P0$ recebe os dados dos blocos B_{23} e B_{32} mais o bloco do somatório auxiliar realizado por $P5$ na função **calculaSomatorioLU**. Dessa forma, ao invés de realizar o cálculo padrão para a posição B_{33}, que corresponde à expressão

$$B_{33} = B_{33} - [(B_{03} * B_{30}) + (B_{13} * B_{31}) + (B_{23} * B_{32})]$$

realizará somente o seguinte cálculo

$$B_{33} = B_{33} - [calculaSomatorioLU(B_{33}) + (B_{23} * B_{32})].$$

Ao finalizar o cálculo da posição B_{33} e receber os demais blocos (B_{24} e B_{42}), $P0$ envia os blocos necessários para os processos correspondentes. Nesta etapa, os processos

Figura 44 – Ilustração do cálculo da decomposição-LU – Etapa 4

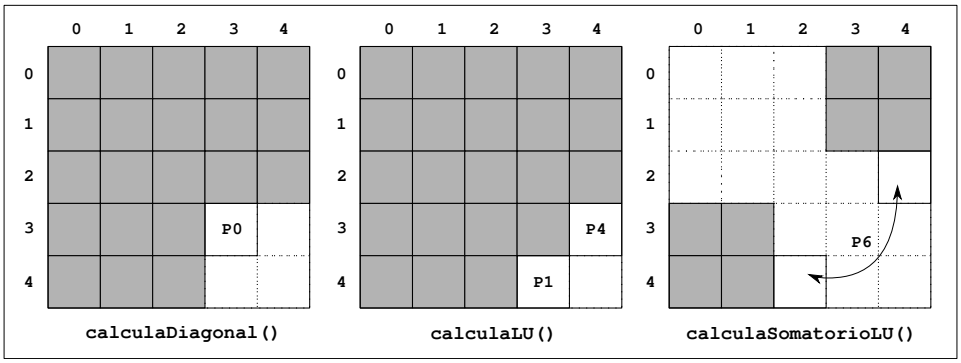
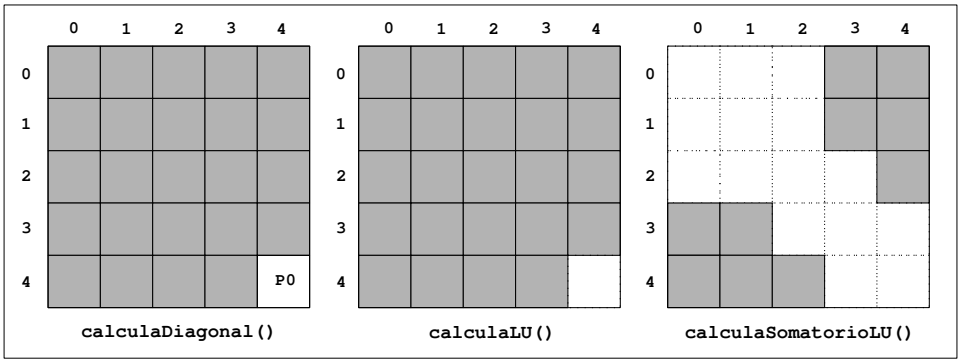


Figura 45 – Ilustração do cálculo da decomposição-LU – Etapa 5



$P2$, $P3$ e $P5$ já concluíram sua computação e finalizaram seu ambiente de execução. Por outro lado, $P1$ e $P4$ computarão sobre os blocos B_{43} e B_{34} respectivamente, enquanto que $P6$ continua auxiliando no cálculo do bloco B_{55}.

Ao finalizar a etapa 4, o processo $P6$ retorna para $P0$ o somatório das posições que lhe foi atribuído. Assim, conforme ilustrado na Figura 45, restará somente o cálculo do bloco diagonal B_{44}. Portanto, $P0$ recebe os blocos restantes para finalizar o cálculo deste bloco. Esta é a última etapa da aplicação, e após ela, $P0$ finaliza seu ambiente de execução.

4.3 Questões de Implementação

As questões técnicas da implementação usando MPI-1 e MPI-2 serão descritas nesta seção. Para isto, inicialmente na Seção 4.3.1 serão apresentadas as questões específicas da implementação com MPI-1. A seguir, na Seção 4.3.2 são apresentadas as questões da implementação utilizando o MPI-2.

4.3.1 Usando MPI-1

A implementação usando MPI-1 deu-se num contexto SPMD. Sendo assim, a aplicação tem todos os processos (principais e auxiliares) criados na inicialização (lançados via linha de comando pelo comando `mpiexec`). Inicialmente, todos os processos irão executar a função **decomposicaoLU**, que irá direcionar cada processo à sua função.

A Figura 46 apresenta o pseudocódigo da função **decomposicaoLU**. Nela, o fluxo de execução que cada cada processo irá seguir é definido pelo seu identificador (*rank*) dentro do comunicador `MPI_COMM_WORLD`. Sendo assim, o processo com identificador 0 representará o Mestre e executará a função **calculaDiagonal**. Os demais processos principais que executarão a função **calculaLU**, serão os Trabalhadores que possuem identificador maior que zero e menor que o identificador do último processo definido para executar esta função. Por fim, caso existam processos auxiliares, estes computarão a função **calculaSomatorioLU**.

O restante da aplicação MPI-1 possui as mesmas características da implementação apresentada na Seção 4.2.2.3 e exemplificada na Seção 4.2.2.4. Sendo assim, a próxima seção apresenta as questões técnicas da implementação usando o MPI-2.

4.3.2 Usando MPI-2

Para aumentar o universo de comparações MPI, foram desenvolvidas quatro versões MPI-2. Todas elas seguem o contexto MPMD e estão implementadas de forma similar a implementação MPI-1, porém com diferenças relacionadas à criação dinâmica

de processos. A seguir, são apresentadas as características específicas de cada versão implementada.

Versão I

A versão I contempla a criação dos processos auxiliares em tempo de execução. Sendo assim, a aplicação inicia sua execução com todos os processos principais criados estaticamente (lançados via de comando através do comando `mpiexec`). Importante lembrar que os processos principais correspondem aos que executam as funções **calculaDiagonal** e **calculaLU**, enquanto que os processos auxiliares computam a função **calculaSomatorioLU**.

Os processos auxiliares são criados pelo processo Mestre, que executa a função **calculaDiagonal**. O pseudocódigo desta função está descrito na Figura 47. Nele, a operação de criação dos processos auxiliares ocorre uma única vez na aplicação. A diretiva `MPI_Comm_spawn` inserida na linha 14 apresenta esta operação, em que serão criados 1 processo à cada chamada. Desta forma, cada processo pertencerá a um intercomunicador diferente e só possuirá comunicação com o processo da função **calculaDiagonal**. O restante da aplicação está implementada de forma similar à apresentada na Seção 4.2.2.3. Entretanto, as comunicações entre os processos das funções **calculaDiagonal** e **calculaSomatorioLU** passarão a ser realizadas através de intercomunicadores.

Versão II

Partindo da implementação da Versão I, a Versão II adiciona a criação dos processos que executam **calculaLU** em tempo de execução. Desta forma, a aplicação inicia somente com o processo Mestre (lançado via linha de comando pelo `mpiexec`), que irá executar diretamente a função **calculaDiagonal**, descrita no pseudocódigo da Figura 48.

Nesta versão, todos os processos Trabalhadores (principais) que computão a função **calculaLU** são criados em uma única execução da diretiva `MPI_Comm_spawn`. Utilizou-se a criação dinâmica desta forma para que possa ser avaliado o impacto de criar todos os

Figura 46 – Pseudocódigo da função **decomposicaoLU** em MPI-1

```
DecomposicaoLU(float *A, int calcLU){
1.  MPI_Init(...);
2.  if(rank==0){
3.     calculaDiagonal(qtdProcLU, qtdProcSum);
4.  }else if(rank > 0 && rank <= calcLU){
5.     calculaLU();
6.  }else if(rank pertence a processos auxiliares){
7.     calculaSomatorioLU();
8.  }
9.  MPI_Finalize();
10.}
```

processos em um único `MPI_Comm_spawn`. Por consequência, estes processos Trabalhadores criados pertencerão a um mesmo intracomunicador, diferente do processo Mestre. Assim, as comunicações entre os processos das funções **calculaDiagonal** e **calculaLU** passam à ser realizadas por meio de intercomunicadores. O restante da aplicação não sofreu alterações.

Versão III

Para evitar o sincronismo da criação de todos os processos num único `MPI_Comm_spawn`, a Versão III implementa a criação de um único processo à cada `MPI_Comm_spawn`. O pseudocódigo desta versão está descrito na Figura 49. Nela, a criação dos processos ocorre internamente ao laço de repetição que realiza a distribuição dos primeiros blocos aos processos da **calculaLU** (linha 4). Assim, tão logo o processo Mestre finalizar a criação do processo, já envia a carga de trabalho à ele. Esta mudança implica também no relacionamento hierárquico dos comunicadores. Isto porque na Versão II todos os processos filhos criados pertencem ao mesmo intracomunicador, enquanto que nesta versão, cada processo criado pertencerá à um intracomunicador diferente.

Versão IV

A Versão IV está implementada de forma similar à Versão III, em que a diferença está no modelo de comunicação utilizado para enviar os blocos após a criação dos processos **calculaLU**. Portanto, nesta versão, utilizaremos a diretiva `MPI_Isend` para realizar o envio dos blocos de forma assíncrona. Desta forma, o processo Mestre não é obrigado

Figura 47 – Pseudocódigo da função **calculaDiagonal** em MPI-2 – Versão I

```
void CalculaDiagonal(int nProcSum, int nProcLU){
1.    Carrega matriz e mapeia para blocos;
2.    Computa blocoDiagonal;
3.    for(p=0;p<nProcLU;p++){
4.      Envios de blocos para calculaLU;
5.      Recebimento assíncrono dos blocos de calculaLU;
6.    }
7.    do{
8.      Aguarda blocos para computar a diagonal;
9.      if(existe dados do somatorio)
10.         Recebe o somatório de calculaSomatorioLU;
11.      Computa blocoDiagonal;
12.      if(primeira computação)
13.         for(p=0;p<nProcSum;p++)
14.            MPI_Comm_spawn(1) p/ função calculaSomatorioLU;
15.      for(i=0;i<qtdRecvLinha;i++){
16.         Realiza operações sobre os blocos linha;
17       }
18.      for(i=0;i<qtdRecvColuna;i++){
19.         Realiza operações sobre os blocos coluna;
20.      }
21.   }while(houver computacao);
22.}
```

a esperar que o envio dos blocos sejam concluídos para poder criar um novo processo. Assim, na Figura 49, na linha 5, ocorrerá o envio assíncrono dos dados.

4.4 Resultados Experimentais: Comparação MPI-1 e MPI-2

Esta seção apresenta os resultados experimentais da comparação realizada entre MPI-1 e MPI-2. Para isso, a configuração dos testes são apresentadas na Seção 4.4.1. Após, a Seção 4.4.2 contextualiza sobre os resultados obtidos, além de analisá-los.

4.4.1 Configuração dos Testes

As características do ambiente de teste (arquitetura) continuam as mesmas utilizadas para as execuções das aplicações do problema do Jogo da Vida, descrito na Seção 3.2.3. O tamanho da matriz de entrada foi definido em 5000×5000, o que proporcionou analisar a escalabilidade das aplicações. Para cada configuração de teste, foram realizadas 50 execuções em que analisaremos a média dos tempos de execução. Destes, foram retirados os 10 melhores e os 10 piores tempos, a fim de minimizar o valor do desvio padrão, que foi inferior à $0,51$ segundos com o MPI-1 e $1,52$ segundos com o MPI-2, valor este, obtido com a execução de 4 processos principais. O desvio padrão dos resultados restantes foi inferior a $0,25$ e $0,42$ segundos em MPI-1 e MPI-2 respectivamente.

Para facilitar a compreensão dos testes, nossa análise está separada em dois cenários:

Figura 48 – Pseudocódigo da função **calculaDiagonal** em MPI-2 – Versão II

```
void CalculaDiagonal(int nProcSum, int nProcLU){
1.   Carrega matriz e mapeia para blocos;
2.   Computa blocoDiagonal;
3.   MPI_Comm_spawn(nProcLU) p/ função calculaLU;
4.   for(p=0;p<nProcLU;p++){
5.     Envios de blocos para calculaLU;
6.     Recebimento assíncrono dos blocos de calculaLU;
7.   }
8    do{
9.     Aguarda blocos para computar a diagonal;
10.     if(existe dados do somatorio)
11.        Recebe o somatório de calculaSomatorioLU;
12.     Computa blocoDiagonal;
13.     if(primeira computação)
14.        for(p=0;p<nProcSum;p++)
15.           MPI_Comm_spawn(1) p/ função calculaSomatorioLU;
16.     for(i=0;i<qtdRecvLinha;i++){
17.        Realiza operações sobre os blocos linha;
18      }
19.     for(i=0;i<qtdRecvColuna;i++){
20.        Realiza operações sobre os blocos coluna;
21.     }
22.  }while(houver computacao);
23.}
```

Figura 49 – Pseudocódigo da função **calculaDiagonal** em MPI-2 – Versão III

```
void CalculaDiagonal(int nProcSum, int nProcLU){
1.    Carrega matriz e mapeia para blocos;
2.    Computa blocoDiagonal;
3.    for(p=0;p<nProcLU;p++){
4.      MPI_Comm_spawn(1) p/ função calculaLU;
5.      Envios de blocos para calculaLU;
6.      Recebimento assíncrono dos blocos de calculaLU;
7.    }
8      do{
9.      Aguarda blocos para computar a diagonal;
10.      if(existe dados do somatorio)
11.         Recebe o somatório de calculaSomatorioLU;
12.      Computa blocoDiagonal;
13.      if(primeira computação)
14.         for(p=0;p<nProcSum;p++)
15.            MPI_Comm_spawn(1) p/ função calculaSomatorioLU;
16.      for(i=0;i<qtdRecvLinha;i++){
17.         Realiza operações sobre os blocos linha;
18       }
19.      for(i=0;i<qtdRecvColuna;i++){
20.         Realiza operações sobre os blocos coluna;
21.      }
22.   }while(houver computacao);
23.}
```

Cenário 1: Comparar a implementação paralela MPI-1 e as três versões MPI-2 que possuem criação em tempo de execução dos processos **calculaLU** (Versões II, III e IV).

Cenário 2: Verificar o impacto da criação dinâmica dos processos auxiliares **calculaSomatorioLU**. Para isto, serão comparadas a implementação MPI-1 e a Versão I implementada em MPI-2;

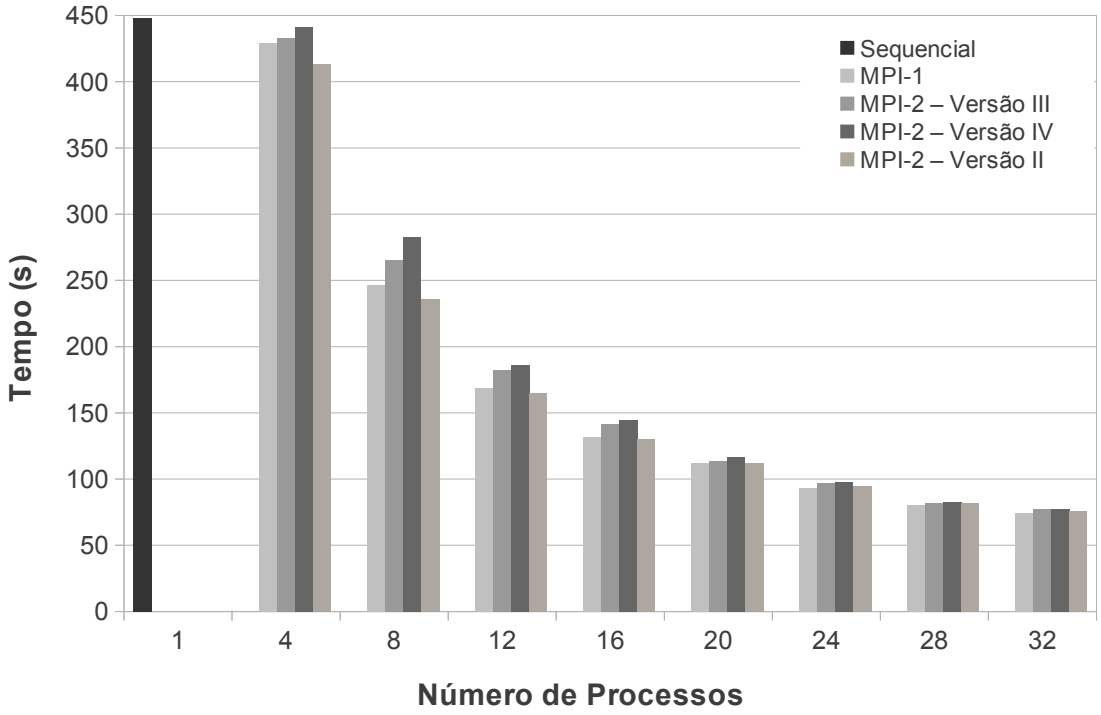
4.4.2 Análise dos Resultados MPI-1 e MPI-2

Cenário 1

O objetivo desta análise é identificar qual das três versões implementadas em MPI-2 (Versões II, III e IV) possui menor impacto na criação dinâmica dos processos **calculaLU** na aplicação. Para isto, definimos alguns objetivos específicos: *(i)* Identificar o impacto da criação síncrona de todos os processos em um único `MPI_Comm_spawn` (Versão II); *(ii)* Identificar o impacto da criação de um processo por chamada ao `MPI_Comm_spawn` (Versão III); e *(iii)* Verificar o comportamento da aplicação com a utilização de envio assíncrono após a criação de cada processo (Versão IV).

Os tempos obtidos (em segundos) com a execução da implementação MPI-1 e das três Versões MPI-2 (II, III e IV) para o intervalo de 1 a 32 processos são apresentados no gráfico de barras da Figura 50. Nele, podemos notar que todas as versões paralelas desenvolvidas permitiram ganho em relação a versão sequencial, que obteve tempo de $448,39$ segundos. Este ganho chegou a cerca de 83% com a execução de 32 processos.

Figura 50 – Tempos de execução de 1 a 32 processos principais



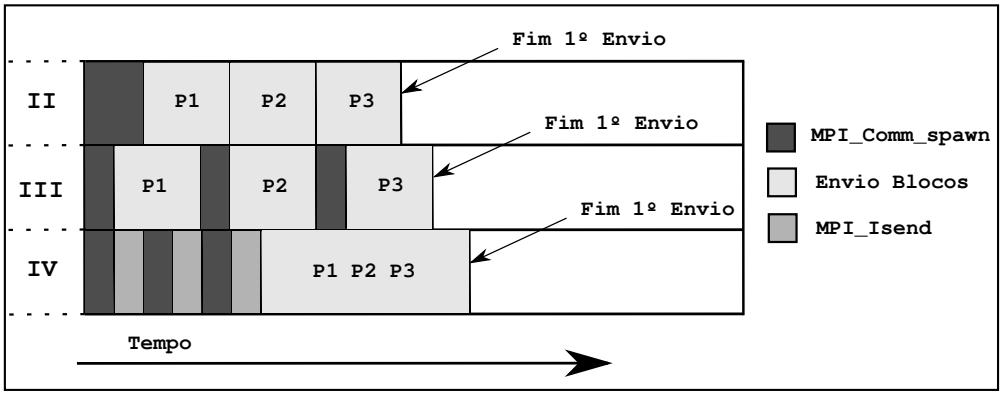
Observa-se que para a execução de 4, 8, 12 e 16 processos a Versão II – MPI-2 foi a melhor, enquanto que a partir da execução com 20 processos, a mais rápida foi a implementação em MPI-1.

Realizando uma comparação entre as três versões desenvolvidas em MPI-2, verificou-se que para todas as quantidades de processo, a Versão II manteve-se a mais eficiente. Isto mostra que para este tipo de aplicação, a criação de todos os processos em um único `MPI_Comm_spawn` é a mais eficiente. Embora tenha-se o custo do sincronismo da operação (o Mestre só conseguirá computar quando finalizar a criação de todos os processos), ele é inferior ao da criação de um único processo a consequente espera pelo envio dos dados à ele, contido nas Versões III e IV.

A Figura 51 auxilia a entender a diferença no tempo de execução entre as três versões MPI-2. Nela, é apresentado o comportamento de cada uma destas versões durante a criação de 3 processos Trabalhadores e consequente envio dos dados. Podemos ver que a Versão II possui um custo único da criação de todos os processos em uma única chamada, e após esta operação, o processo Mestre só possui a tarefa de enviar os dados aos processos (P1, P2 e P3) recém criados.

Ainda na Figura 51, na Versão III, após a criação de cada processo Trabalhador, o Mestre redireciona os dados para que o processo recém criado possa computar. Observa-se que o processo P1 conclui o recebimento dos dados antes que na Versão II. Entretanto, conforme o tempo decorrido da aplicação, o envio dos dados tão logo o processo seja criado

Figura 51 – Comportamento das Versões II, III e IV com relação ao tempo no Cenário 1



acaba se tornando ineficiente. Isto porque o tempo que o Mestre terá que esperar para criar um novo processo é dependente do tempo de comunicação dos dados do processo anterior. Portanto, para uma quantidade grande de dados comunicando, que é o caso que acontece até a utilização de 20 processos, este tempo impacta negativamente na eficiência da aplicação.

O comportamento da Versão IV também é apresentado na Figura 51. Nela, tão logo o Mestre finaliza a criação de um processo, ele realiza uma chamada à diretiva de envio assíncrono `MPI_Isend`. Desta forma, ele consegue criar processos enquanto está ocorrendo o envio dos dados. Entretanto, quanto antes os Trabalhadores receberem os dados, antes começaram a computar. Portanto, percebe-se que o envio assíncrono dos dados para este tipo de aplicação causa impacto negativo na eficiência da aplicação. Isto porque, conforme características do Método de *Doolittle* apresentado na Seção 4.1.1, a primeira computação é dependente do recebimento completo dos dados.

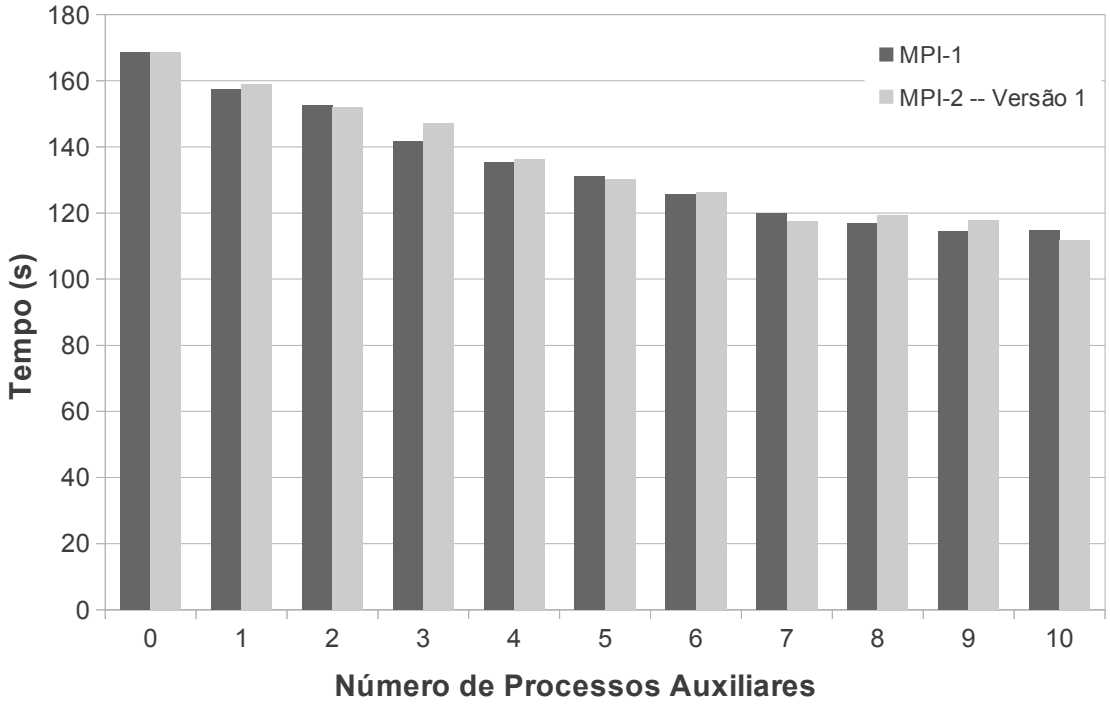
Assim, pode-se concluir que para as três versões paralelas (II, III e IV) implementadas em MPI-2, a mais eficiente e adequada para esta aplicação é a Versão II. No melhor dos casos, com a execução de 8 processos (1 Mestre e 7 Trabalhadores), a Versão II foi 12% e 17% mais rápida que as duas versões MPI-2 respectivamente. Embora este valor seja sensivelmente reduzido com o aumento do número de processos e consequente diminuição do tempo de comunicação, a Versão II continua sendo a mais eficiente para todas as quantidades de processo envolvendo as versões MPI-2.

Cenário 2

O objetivo desta análise é identificar o impacto da criação dos processos auxiliares em tempo de execução. Para isto, serão comparadas a implementação MPI-1 e a Versão I implementada em MPI-2.

Na Figura 52, são apresentados os tempos de execução (em segundos) com a utilização de até 10 processos auxiliares. Neste caso de teste, foram utilizados 12 processos

Figura 52 – Tempos de execução de até 10 processos auxiliares

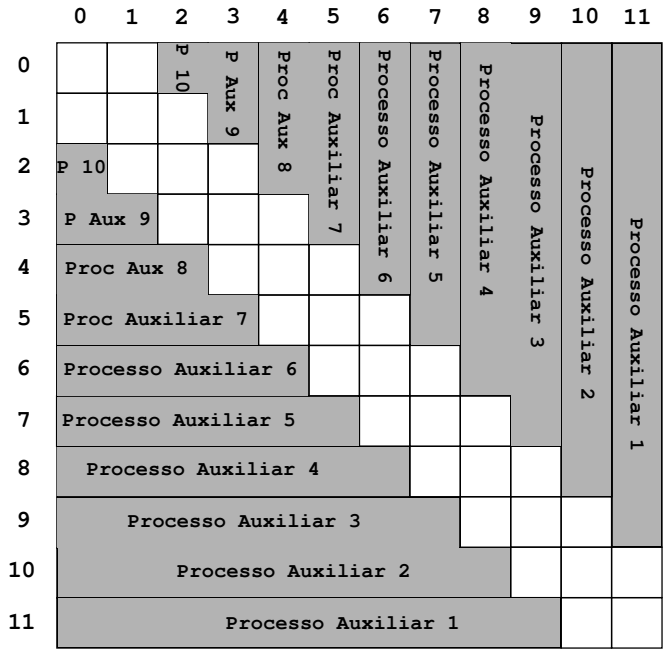


principais para calcular as funções **calculaDiagonal** e **calculaLU**. Assim, o total de processos utilizados pela aplicação é soma dos processos principais e auxiliares, neste caso, 22 processos. Observa-se que para todas as quantidades de processos auxiliares, esta nova função desenvolvida possibilitou ganhos em relação a versão sem utilização de processos auxiliares (0 processos auxiliares).

Ainda na Figura 52, nota-se que a criação dos processos auxiliares em tempo de execução não possibilitou ganho expressivo. Da criação de até 10 processos auxiliares, a criação dinâmica foi mais eficiente em 4 configurações, com a criação de 2, 5, 7 e 10 processos auxiliares. Nos demais casos, a criação destes processos em tempo de execução causou impacto negativo na eficiência da aplicação.

Para auxiliar no entendimento do porque a criação dinâmica de processos foi mais eficiente somente nestes 4 casos, ilustramos a divisão da carga de trabalho entre os processos auxiliares na Figura 53. Ela apresenta a forma com que os 10 processos auxiliares são distribuídos em uma matriz de blocos 12×12. A distribuição começa pela última linha/coluna, em que o cálculo do bloco B_{1111} será auxiliado pelo processo auxiliar 1. Já o bloco B_{1010} será previamente calculado pelo processo auxiliar 2. Esta distribuição segue esta ordem até o número de processos auxiliares definidos ou suportado pela aplicação. Neste caso, o máximo de processos auxiliares que podem ser utilizados são 10. Por exemplo, para a utilização de 5 processos auxiliares, os blocos da diagonal compreendidos entre as posições B_{77} e B_{1111} serão previamente calculados por estes processos.

Figura 53 – Distribuição de 10 processos auxiliares



Conforme apresentado na Figura 52, a utilização de 2, 5, 7 e 10 processos auxiliares se mostrou mais eficiente com a criação dinâmica. Isto quer dizer que para estas quantidades de processos auxiliares, a carga de trabalho disponibilizada à estes processos compensou o custo do sincronismo envolvido na criação dinâmica. Isto não ocorre nos demais casos, pois a carga de trabalho atribuída não é suficientemente grande para com-

Figura 54 – Tempos de execução de até 14 processos auxiliares.

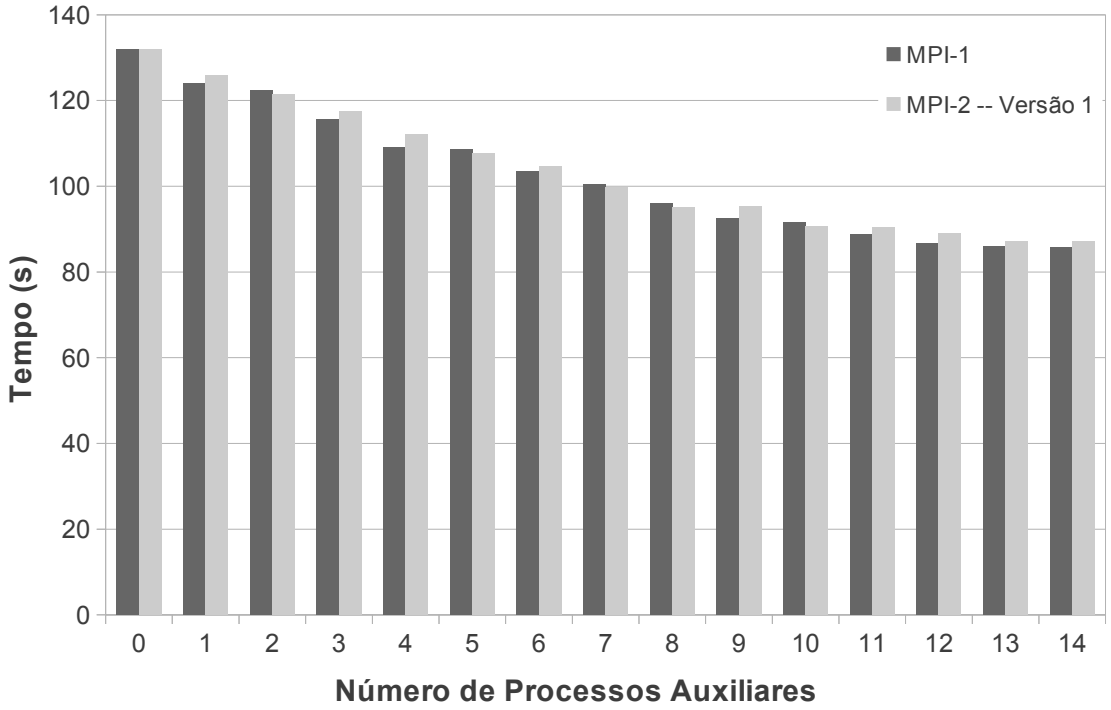
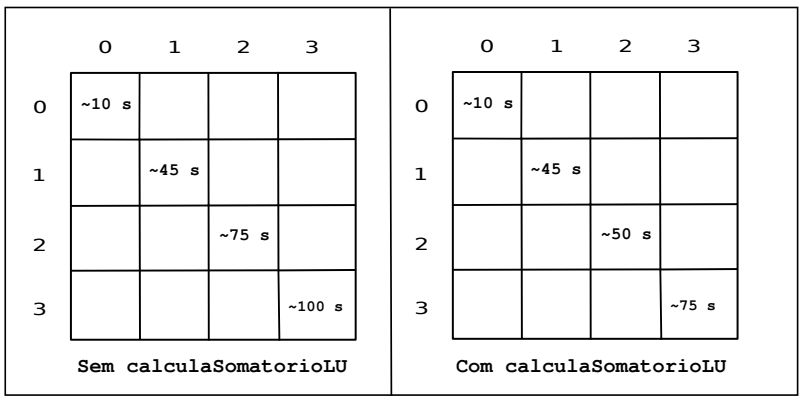


Figura 55 – Comportamento da aplicação com a função **calculaSomatorioLU**.



pensar o sobre-custo da criação dos processos em tempo de execução. Este resultado é influenciado diretamente pela quantidade de elementos em cada bloco, ou seja, os casos em que o MPI-1 foi mais eficiente, significa que estes blocos possuíam quantia considerável de elementos nulos.

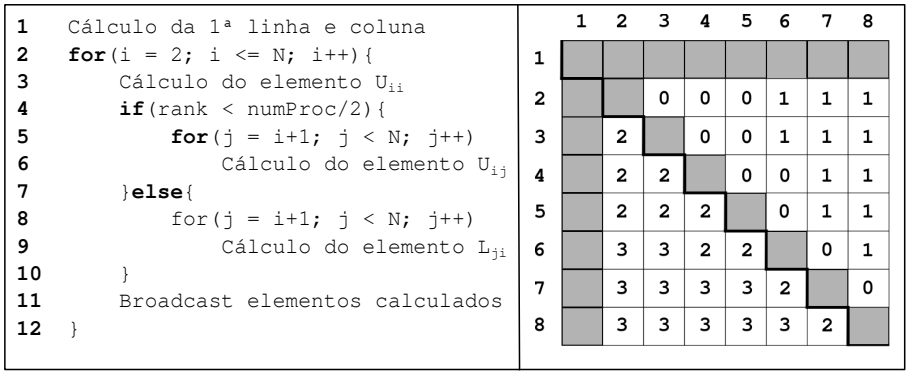
A Figura 54 apresenta os tempos obtidos para a execução de 16 processos principais e 14 processos auxiliares. Nela, a criação dinâmica de processos foi mais eficiente em 5 casos de teste, com 2, 5, 7, 8 e 10 processos auxiliares. No restante dos casos de teste, novamente a implementação em MPI-1 foi a mais eficiente.

Estas duas configurações apresentadas (criação de 10 e 14 processos auxiliares) foram as mais equilibradas, em que a criação dinâmica de processos possibilitou algum ganho. Nos demais casos de teste, a implementação MPI-1 foi a mais eficiente na maioria das configurações. Os tempos de execução para estes outros casos de teste são apresentados no Apêndice B.

A função **calculaSomatorioLU** possibilitou ganhos, pois o processo que a executa, realiza previamente o cálculo do somatório do produto dos blocos utilizados para calcular o bloco principal. Desta forma, o tempo de computação do bloco da diagonal é reduzido consideravelmente, e assim, o cálculo do bloco diagonal deixa de ser o gargalo principal da aplicação, visto que todos os processos têm que aguardar o cálculo do bloco diagonal para computarem seus elementos dos blocos linha e coluna.

A Figura 55 apresenta o comportamento da aplicação quando utilizada a função **calculaSomatorioLU**. Ela apresenta os tempos obtidos para a execução de 4 processos principais e 2 processos auxiliares. Na matriz à direita (com calculaSomatorioLU), podemos notar que o tempo de computação dos blocos B_{22} e B_{33} reduziu consideravelmente com a utilização dos processos auxiliares para calcular previamente o somatório destes blocos.

Figura 56 – Pseudocódigo paralelo e ilustração da distribuição das tarefas.



```
1   Cálculo da 1ª linha e coluna
2   for(i = 2; i <= N; i++){
3       Cálculo do elemento U_ii
4       if(rank < numProc/2){
5           for(j = i+1; j < N; j++)
6               Cálculo do elemento U_ij
7       }else{
8           for(j = i+1; j < N; j++)
9               Cálculo do elemento L_ji
10      }
11      Broadcast elementos calculados
12  }
```

	1	2	3	4	5	6	7	8
1								
2			0	0	0	1	1	1
3		2		0	0	1	1	1
4		2	2		0	0	1	1
5		2	2	2		0	1	1
6		3	3	2	2		0	1
7		3	3	3	3	2		0
8		3	3	3	3	3	2	

4.4.3 Discussão e Possíveis Melhorias

Neste trabalho, foi apresentada a implementação paralela utilizando a distribuição dos dados da matriz de entrada em blocos quadrados e a utilização do modelo de programação Mestre/Trabalhador. A implementação utilizando este modelo nos permitiu comparar o impacto da criação dinâmica de processos em dois momentos diferentes da aplicação. Durante o desenvolvimento e análise das implementações paralelas do *Skyline Matrix Solver*, identificamos oportunidades de realizar melhorias na aplicação, que podem influenciar positivamente na eficiência da aplicação paralela:

Envio dos dados: Com a distribuição dos elementos da matriz baseado em blocos quadrados e comunicações ponto-a-ponto, notou-se que a comunicação influenciou na eficiência da aplicação. Isto ocorreu pois a quantidade de dados envolvida em cada comunicação de envio/recebimento de blocos é muito grande. Portanto, pretende-se testar a implementação com divisão dos dados em linhas e colunas e a utilização de comunicações coletivas, com a finalidade de eliminar o redirecionamento dos blocos por parte do Mestre.

Modelo de Programa: Identificou-se que o cálculo do bloco da diagonal pode ser realizado pelos processos Trabalhadores, sem a necessidade de aguardar a computação pelo Mestre. Desta forma, consegue-se eliminar a comunicação para troca dos blocos da diagonal com o Mestre. Para tanto, uma possibilidade seria utilizar outro modelo de programa, como por exemplo, o de Fases Paralelas.

Através das oportunidades identificadas, foi realizada a implementação em MPI-1 utilizando o modelo Fases Paralelas e particionamento dos dados em vetores unidimensionais (linha e coluna). A Figura 56 ilustra como a aplicação está organizada. Nela, o pseudocódigo (situado à esquerda) descreve o algoritmo em alto nível de abstração e a matriz (situada à direita) apresenta a estratégia utilizada na distribuição das tarefas entre 4 processos para uma matriz 8×8.

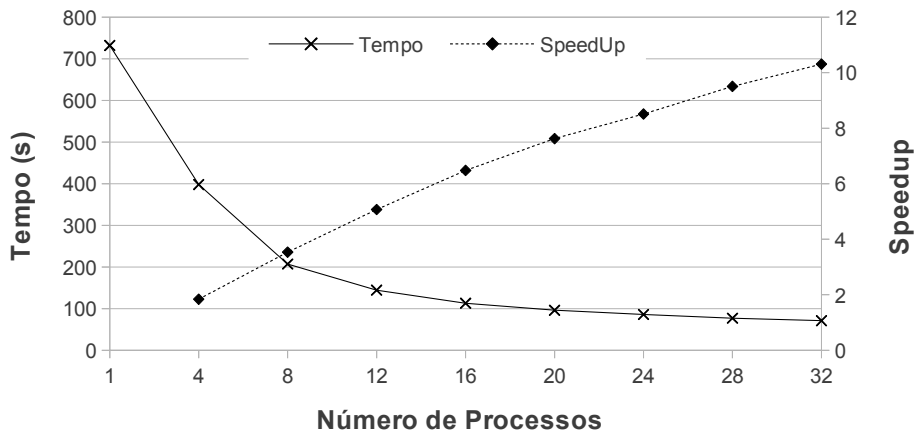
Para melhor compreender o pseudocódigo apresentado na Figura 56, dividimos ele em quatro fases: *(i)* Inicialmente, todos os processos (neste caso, 4) carregam a matriz de entrada e calculam a primeira linha e coluna da matriz (linha 1 do pseudocódigo); *(ii)* Todos os processos calculam o elemento da diagonal principal (linha 3); *(iii)* Se o *rank* (identificador) do processo for menor for menor do que a metade do número total de processos, caberá ao processo computar os elementos correspondentes a matriz U (linha 4 a 6). Caso contrário, o processo calculará seus elementos correspondentes à matriz L (linha 8 e 9). Por exemplo, na segunda linha da matriz, o processo 0 calculará os elementos U_{23}, U_{24} e U_{25} e o processo 1 os outros três elementos da linha. No segundo caso, o processo 2 calculará os elementos L_{32}, L_{42} e L_{52} e o processo 3 os outros três elementos da coluna 2; *(iv)* Após o cálculo de seus elementos, cada processo envia seus dados computados aos demais processos através da primitiva de comunicação coletiva `MPI_Bcast` (operação síncrona, onde um processo envia dados para um grupo de processos). As fases *(ii)*, *(iii)* e *(iv)* são repetidas até que todos os elementos da matriz tenham sido computados.

Esta versão desenvolvida em MPI-1 proporcionou uma redução de até 90% do tempo da execução sequencial para a execução com 32 processos em uma matriz de tamanho 6000×6000, conforme apresenta o gráfico da Figura 57 (LORENZON; CERA; ROSSI, 2013). Assim, um dos trabalhos futuros será desenvolver uma versão com modelo Fases Paralelas e comunicação coletiva em MPI-2.

4.5 Conclusão do Capítulo

Este capítulo apresentou os resultados obtidos na execução de aplicações MPI-1 e MPI-2 do *Skyline Matrix Solver*. O objetivo desta análise foi identificar o impacto da criação dinâmica de processos em dois momentos diferentes de uma aplicação com carga

Figura 57 – Tempo de execução em segundos (eixo y à esquerda) e *speedup* (eixo y à direita) para 4 a 32 processos.



irregular de trabalho. Para isso, foi implementada uma versão paralela em MPI-1 e outras 4 versões em MPI-2.

As comparações envolvendo a implementação MPI-1 e as versões II, III e IV em MPI-2 (Cenário 1) possibilitou identificar qual destas três versões possui menor impacto na criação dinâmica dos processos **calculaLU**. Verificou-se que a criação de todos os processos em um único `MPI_Comm_spawn` (Versão II) se tornou mais eficiente que a criação de um processo a cada `MPI_Comm_Spawn` e o redirecionamento imediato dos blocos para o processo recém criado. Adicionalmente, exploramos a utilização do envio assíncrono dos blocos tão logo o processo tenha sido criado. Verificou-se que para este tipo de aplicação, que possui grande dependência de dados, em que a computação só ocorre quando o processo tiver todos os dados necessários, o envio assíncrono causa impacto negativo na eficiência da aplicação.

Ainda nas comparações do Cenário 1, a criação de todos os processos em um único `MPI_Comm_spawn` foi mais eficiente que a criação estática de todos os processos no intervalo de 4 a 16 processos. Entretanto, conforme aumenta o número de processos à serem criados, também aumenta o tempo decorrente do sincronismo envolvendo a criação deles. Portanto, a partir de 20 processos, a criação dos processos ao iniciar a execução, passou a ser mais eficiente.

As comparações do Cenário 2 identificaram o impacto da criação dos processos auxiliares em tempo de execução. Para isto, comparou-se a implementação MPI-1, que cria os processos auxiliares no início da aplicação e a Versão I implementada em MPI-2, que possui a criação destes processos em tempo de execução. Verificou-se que para os casos onde há carga de trabalho suficientemente grande para ser destinada à estes processos auxiliares, a criação dinâmica deles foi mais eficiente. Este comportamento é altamente dependente da densidade de valores nulos (zeros) na matriz *skyline*. Isto porque para uma matriz com baixa densidade de zeros, a carga de trabalho será maior, e consequentemente a eficiência da aplicação MPI-2 será maior. Entretanto, para matrizes com alta densidade de zeros, a carga de trabalho é menor e o mais eficiente é manter os processos auxiliares alocados desde o início da aplicação.

Por fim, com as melhorias propostas, foi realizada a implementação paralela em MPI-1 utilizando o modelo fases paralelas e divisão dos dados da matriz em linhas e colunas. Os resultados mostraram que as melhorias possibilitaram um ganho maior em relação à versão com divisão dos dados em blocos quadrados. Portanto, pretendemos, futuramente inserir a criação dinâmica de processos na implementação do *Skyline Matrix Solver* com modelo de programação Fases Paralelas e comunicação coletiva.

5 Conclusão

Este trabalho realizou um estudo sobre o desenvolvimento de aplicações com criação dinâmica de processos. Desta forma, optamos por estudar a implementação desta característica do MPI-2 em dois problemas da suíte *Cowichan Problems* que possuem características diferentes: O Jogo da Vida, que possui carga de trabalho regular, em que todos os processos possuem tempo de computação similar, e o *Skyline Matrix Solver*, que possui carga de trabalho irregular, em que todos os processos possuem tempos diferentes de computação e o problema é altamente dependente.

A análise realizada sobre o Jogo da Vida procurou identificar o custo extra causado pela criação de processos em tempo de execução. Para isto, foram desenvolvidas duas versões MPI-1 e MPI-2, em que a diferença entre elas era unicamente a criação dinâmica dos processos e o modelo hierárquico consequente. Nas comparações envolvendo a criação de todos os processos em um único `MPI_Comm_spawn` identificou-se um acréscimo constante de 1,25 segundos para todas as quantidades de processos para a versão MPI-2 no nosso ambiente de testes. Parte deste valor se deve ao bloqueio do Mestre aguardando a criação de todos os processos. Quando inserimos a criação de um único processo à cada `MPI_Comm_spawn`, verificou-se que é possível atenuar o custo adicional proveniente da criação dinâmica. Assim, tão logo um Trabalhador seja criado, ele já recebe sua parcela de dados à computar.

O estudo envolvendo o *Skyline Matrix Solver* possibilitou analisar o impacto de criar processos dinamicamente e até que ponto as características avançadas do MPI podem ser utilizadas para atenuar este custo extra. Assim, implementou-se uma versão MPI-1, com a criação de todos os processos estaticamente, e quatro versões MPI-2. Verificou-se, que a criação dinâmica de processos em aplicações com carga irregular de trabalho pode ser compensada através da sobreposição de computação com comunicação. Conforme apresentado, a criação dinâmica de todos os processos em um único `MPI_Comm_spawn` foi mais rápida que a criação de todos os processos criados estáticamente em quatro configurações de teste: com a execução para 4, 8 12 e 16 processos. Isto ocorreu, pois o tempo útil de computação destinado à estes processos é suficientemente grande para compensar o custo da criação de todos os processos em um único `MPI_Comm_spawn`, fixado em aproximadamente 1,25 segundos conforme análise realizada sobre o Jogo da Vida.

Diferente dos resultados obtidos sobre o estudo do Jogo da Vida, o envio dos dados tão logo o processo tenha sido criado, não compensou a espera pela criação síncrona de todos os processos em um único `MPI_Comm_spawn`. Conforme os resultados, para uma

carga grande de dados, o processo Mestre ficará bloqueado até o final do envio para cada processo, para só após, poder criar um novo processo. Entretanto, com o aumento do número de processos e a diminuição da quantidade de dados, o custo da criação dos processos em tempo de execução é sensivelmente atenuado. Isto porque o tempo que o Mestre ficará bloqueado realizando o envio dos dados é menor. Com a utilização do envio de dados assíncrono, tão logo o processo tenha sido criado, verificou-se que para este tipo de aplicação, em que os dados são altamente dependentes, o envio assíncrono é ineficiente. Isto porque os dados não serão obrigatoriamente enviados no momento da chamada a primitiva, e sim em algum momento do tempo da aplicação. Portanto, não é indicado para aplicações em que os processos dependam de dados provenientes de outros processos para poder computar.

O *Skyline Matrix Solver* permitiu analisar ainda, o impacto da criação de processos para computar sobre diferentes dados que possuem tempos de computação diferentes. Assim, verificou-se que a criação dinâmica só é compensada quando é oferecido tempo útil de computação suficientemente grande.

Conforme apresentado no decorrer deste trabalho, a característica de criação dinâmica de processos oferecida pelo MPI-2 pode ser utilizada com sucesso em aplicações que possuem carga de trabalho irregular. Entretanto, a implementação de aplicações com criação dinâmica de processos envolve uma complexidade maior do que a criação de *threads*, por exemplo. Portanto, as comparações realizadas neste trabalho permitirão planejar o uso desta característica do MPI-2 em situações onde a flexibilidade proporcionada seja o fator determinante da aplicação/arquitetura.

Ao fim deste trabalho, identificamos oportunidades de realizar trabalhos futuros com a finalidade de explorar ainda mais a criação dinâmica de processos em MPI. Desta forma, pretendemos implementar o *Skyline Matrix Solver* utilizando criação de processos em tempo de execução para o modelo de programa Fases Paralelas. Também pretendemos analisar a utilização de Fases Paralelas no problema do Jogo da Vida. Adicionalmente, buscaremos explorar o consumo de energia e memória das aplicações desenvolvidas com MPI-1 e MPI-2. Esta análise futura será útil para a paralelização de aplicações em sistemas embarcados, visto que possuem quantidade reduzida de memória.

Referências

ALMASI, G.; GOTTLIEB, A. (Ed.). *Highly Parallel Computing*. [S.l.]: Benjamin Cummins, 1994. Citado na página 25.

ANVIK, J. et al. Asserting the utility of CO_2P_3S using the cowichan problem set. *J. Parallel Distrib. Comput*, 2005. p. 1542, 2005. Citado na página 22.

ASPRAY, W. *John von Neumann and the Origins of Modern Computing*. [S.l.]: MIT Press, 1990. 394 p. Citado na página 25.

BLOK, H. J.; BERGERSEN, B. Effect of boundary conditions on scaling in the game of life. *Physical. Review. E*, 1997. v. 55, n. 5, p. 6249–6252, 1997. Citado na página 40.

BOUMAN, D. S. *Parallelizing a Skyline Matrix Solver using Orca*. [S.l.], set. 1995. Citado 2 vezes nas páginas 34 e 58.

BROWN, T.; XIONG, R. A parallel quicksort algorithm. *J. Parallel Distrib. Comput.*, 1993. Academic Press, Inc., Orlando, FL, USA, p. 83–89, 1993. ISSN 0743-7315. Citado na página 29.

CERA, M. C. et al. Supporting malleability in parallel architectures with dynamic cpusets mapping and dynamic mpi. In: *Proceedings of the 11th international conference on Distributed computing and networking*. Berlin, Heidelberg: Springer-Verlag, 2010. (ICDCN'10), p. 242–257. ISBN 3-642-11321-4, 978-3-642-11321-5. Citado 2 vezes nas páginas 21 e 36.

CERA, M. C. et al. Improving the dynamic creation of processes in mpi-2. In: *Proceedings of the 13th European PVM/MPI User's Group conference on Recent advances in parallel virtual machine and message passing interface*. Berlin, Heidelberg: Springer-Verlag, 2006. (EuroPVM/MPI'06), p. 247–255. ISBN 3-540-39110-X, 978-3-540-39110-4. Citado na página 36.

DICHTER, J.; MAHMOOD, A.; SHOLL, H. A. Optimum data distributions for parallel partitioned LU decomposition. In: LEE, R. Y. (Ed.). Cancun, Mexico: ISCA, 1999. p. 289–293. Citado na página 56.

FLYNN, M. Some computer organizations and their effectiveness. *IEEETC:JOURNAL*, 1972. p. 948–960, 1972. Citado na página 25.

FOSTER, I. *Designing and building parallel programs:Concepts and tools for parallel software*. [S.l.]: Addison-Wesley, 1995. Citado 2 vezes nas páginas 28 e 43.

GALANTE, G.; BONA, L. de. Nebulous: A framework for scientific applications execution on cloud environments. In: *Simpósio em Sistemas Computacionais (WSCAD-SSC)*. Vitória, ES: IEEE, 2011. p. 1–8. Citado na página 21.

GARDNER, M. Mathematical games - the fantastic combinations of john conway's new solitaire game, life. *Scientific American*, 1970. USA, n. 223, p. 120–123, out. 1970. Citado na página 39.

GEIST, A. et al. *PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing*. [S.l.]: MIT Pres, 1994. (Scientific and Engineering Computation). Citado na página 28.

GEIST, G. A.; KOHLA, J. A.; PAPADOPOULOS, P. M. PVM and MPI: A comparison of features. *Calculateurs Paralleles*, 1996. v. 8, p. 137, 1996. Citado na página 28.

GROPP, W. et al. (Ed.). *MPI – The Complete Reference*. Cambrdge, MA: MIT Press, 1998. Citado 2 vezes nas páginas 21 e 28.

GROPP, W. D. et al. Using MPI-2: Advanced features of the message passing interface. In: *CLUSTER*. [S.l.]: IEEE Computer Society, 2003. ISBN 0-7695-2066-9. Citado 2 vezes nas páginas 21 e 31.

JEON, M.; KIM, D. Parallel merge sort with load balancing. *Int. J. Parallel Program.*, 2003. Kluwer Academic Publishers, Norwell, MA, USA, p. 21–33, 2003. ISSN 0885-7458. Citado na página 29.

LEOPOLD, C.; SüSS, M. Observations on mpi-2 support for hybrid master/slave applications in dynamic and heterogeneous environments. In: *Proceedings of the 13th European PVM/MPI User's Group conference on Recent advances in parallel virtual machine and message passing interface*. Berlin, Heidelberg: Springer-Verlag, 2006. (EuroPVM/MPI'06), p. 285–292. ISBN 3-540-39110-X, 978-3-540-39110-4. Citado na página 35.

LORENZON, A. F.; CERA, M. C.; ROSSI, F. D. Análise da distribuição de carga de trabalho em mpi utilizando o jogo da vida. In: *12th Escola Regional de Alto Desempenho - ERAD*. Erechim, RS: SBC, 2012. p. 129–132. ISSN 2177-0085. Citado na página 42.

LORENZON, A. F.; CERA, M. C.; ROSSI, F. D. Analysing the impact of mpi-2 dynamic process creation to the game of life problem. In: *Computer Systems (WSCAD-SSC), 2012 13th Symposium on*. Petrópolis - RJ: IEEE, 2012. p. 133 –140. Citado na página 48.

LORENZON, A. F.; CERA, M. C.; ROSSI, F. D. Proposta de paralelização do skyline matrix solver utilizando mpi. In: *13 th Escola Regional de Alto Desempenho*. Porto Alegre - RS: SBC, 2013. Citado na página 76.

MAILLARD, N.; CERA, M. C. *Message-Passing Interface Avançado*. [S.l.]: 10th Escola Regional de Alto Desempenho - SBC, 2010. Citado na página 30.

PAUDEL, J.; AMARAL, J. N. Using the cowichan problems to investigate the programmability of x10 programming system. In: *Proceedings of the 2011 ACM SIGPLAN X10 Workshop*. New York, USA: ACM, 2011. (X10 11), p. 4:1–4:10. Citado 2 vezes nas páginas 22 e 35.

PEZZI, G. P. et al. Escalonamento dinâmico de programas mpi-2 utilizando divisão e conquista. In: *Simpósio em Sistemas Computacionais (WSCAD-SSC)*. Ouro Preto - MG: IEEE, 2006. p. 1–8. Citado na página 35.

PRESS, W. H. et al. *Numerical Recipes in C++ - The Art of Scientific Computing - Second Edition*. 2002. Citado 2 vezes nas páginas 55 e 57.

RAUBER, T.; RÜNGER, G. *Parallel Programming - for Multicore and Cluster Systems*. [S.l.]: Springer, 2010. Citado 3 vezes nas páginas 21, 25 e 59.

SCHEPKE, C. *Ambientes de Programação Paralela*. Porto Alegre, RS, Brasil, 2009. Citado na página 29.

SILVA, J. A. da; REBELLO, V. E. F. A hybrid fault tolerance scheme for easygrid mpi applications. In: *Proceedings of the 9th International Workshop on Middleware for Grids, Clouds and e-Science*. New York, NY, USA: ACM, 2011. (MGC '11), p. 4:1–4:6. ISBN 978-1-4503-1068-0. Citado na página 36.

SNIR, M. et al. (Ed.). *MPI: The Complete Reference. Volume 1, The MPI-1 Core*. pub-MIT:adr: MIT Press, 1998. Citado 2 vezes nas páginas 28 e 29.

TANENBAUM, A. S. *Structured Computer Organization*. Englewood Cliffs, NJ: Prentice-Hall, 1976. Citado na página 26.

WALKER, D. MPI: A standard message passing interface for distributed memory environments. In: *Intel Supercomputer User Group 1993 Conference (ISUG)*. St. Louis: Intel Scientific Computers, 1993. p. 67–75. Citado na página 29.

WILKINSON, B.; ALLEN, M. (Ed.). *Parallel programming - techniques and applications using networked workstations and parallel computers (2. ed.)*. [S.l.]: Pearson Education, 2005. Citado na página 28.

WILSON, G. V. Assessing the usability of parallel programming systems: the cowichan problems. *Programming Environments for Massively Parallel Distributed Systems Working Conference of the IFIP WG 103*, 1994. Birkhauser, p. 183–193, 1994. Citado na página 34.

WILSON, G. V.; BAL, H. E. Using the Cowichan problems to assess the usability of Orca. *IEEE parallel and distributed technology: systems and applications*, 1996. p. 36–44, 1996. Citado na página 34.

WILSON, G. V.; BAL, H. E. *The Cowichan Experience*. 2007. Citado 2 vezes nas páginas 22 e 34.

Apêndices

APÊNDICE A – Funções MPI

Este Apêndice objetiva apresentar as funções MPI utilizadas na implementação dos problemas Jogo da Vida e *Skyline Matrix Solver*.

A.1 Inicialização/Finalização

- **MPI_Init**

  int `MPI_Init( int *argc, char **argv );`

  Inicializa o ambiente de execução MPI, em que os argumentos *argc* e *argv* são ponteiros para o número de argumentos e para o vetor de argumentos respectivamente. Todo processo MPI deve executar esta rotina ao iniciar a execução.

- **MPI_Finalize**

  int `MPI_Finalize( void );`

  Finaliza o ambiente de execução MPI. Todos os processos MPI devem executar esta rotina antes de finalizar sua execução.

A.2 Envio/Recebimento

- **MPI_Send**

  int `MPI_Send( void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm );`

  Função utilizada para realizar o envio bloqueante de dados, ou seja, a função ficará bloqueada até que o último dado tenha sido enviado ou copiado para o *buffer* de memória. Nela, serão enviados *count* dados presentes em *buf* com o tipo de dados definido em *datatype* para o processo *dest*. A mensagem é identificada por uma *tag* e o canal de comunicação utilizado é representado pelo parâmetro *comm*.

  Parâmetros de entrada

  - **buf**: Endereço inicial do *buffer* de envio;
  - **count**: Número (não-negativo) de elementos contidos no *buffer* de envio;

- **datatype**: Tipo de dados de cada elemento contido no *buffer* de envio;
- **dest**: *Rank* do processo de destino;
- **tag**: Identificador da mensagem;
- **comm**: Canal de comunicação utilizado para realizar a transferência de dados.

- **MPI_Isend** int `MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request);`

  Esta função, representa o envio de dados não-bloqueante, ou seja, difere da anterior (`MPI_Send()`) **Parâmetros de Entrada**

  - **buf**: Endereço inicial do *buffer* de envio;
  - **count**: Número (não-negativo) de elementos contidos no *buffer* de envio ;
  - **datatype**: Tipo de dados de cada elemento contido no *buffer* de envio;
  - **dest**: *Rank* do processo de destino;
  - **tag**: Identificador da mensagem;
  - **comm**: Comunicador utilizado para realizar a transferência de dados.

  Parâmetro de Saída

  - **request**: Requisição da comunicação.

- **MPI_Recv**

  int `MPI_Recv( void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status );`

  Função utilizada para o recebimento bloqueante de dados. Nela, os *count* dados enviados pelo processo *source* são recebidos e armazenados em *buf* com tipo de dados *datatype*. A mensagem possui uma *tag* de identificação e é transferida através do canal de comunicação *comm*. Informações relacionadas à operação podem ser recuperadas em *status*.

  Parâmetros de Entrada

  - **count**: Número (não-negativo) de elementos contidos no *buffer* de recebimento;
  - **datatype**: Tipo de dados de cada elemento contido no *buffer* de recebimento;
  - **source**: *Rank* do processo de origem da mensagem;
  - **tag**: Identificador da mensagem;
  - **comm**: Canal de comunicação utilizado para realizar a transferência de dados.

  Parâmetro de Saída

- **buf**: Endereço inicial do *buffer* de recebimento;
- **status**: Objeto status, utilizado para recuperar informações sobre o procedimento.

- **MPI_Irecv**

int `MPI_Irecv( void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request );`

Função utilizada para o recebimento não-bloqueante de dados. Esta é uma operação não síncrona de recebimento, em que o processo que está recebendo os dados executa esta diretiva e segue sua computação. O recebimento dos dados pode ser aguardado pela função `MPI_Waitany`, que será abordada a seguir. Possui parâmetros similares aos da função `MPI_Recv`, porém difere do último parâmetro que será a posição da operação no vetor de respostas. Este vetor será utilizado para aguardar a finalização da operação pelas diretivas de espera, por exemplo, `MPI_Waitany`. Seus parâmetros são apresentados a seguir:

Parâmetros de Entrada

- **count**: Número (não-negativo) de elementos contidos no *buffer* de recebimento;
- **datatype**: Tipo de dados de cada elemento contido no *buffer* de recebimento;
- **source**: *Rank* do processo de origem da mensagem;
- **tag**: Identificador da mensagem;
- **comm**: Comunicador utilizado para realizar a transferência de dados.

Parâmetro de Saída

- **buf**: Endereço inicial do *buffer* de recebimento;
- **request**: Requisição da comunicação.

- **MPI_Bcast**

int `MPI_Bcast( void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm );`

Função utilizada para o envio de dados de um processo para todos os outros processos contidos no mesmo grupo. Nela, o processo com *rank root* envia os *count* dados que estarão armazenados em *buf* com tipo de dados *datatype*. A comunicação abrangerá os processos do comunicador *comm*. Seus parâmetros estão organizados como segue:

Parâmetros de Entrada

- **buf**: Endereço inicial do *buffer* de recebimento. Considerado parâmetro de saída para o processo que envia e parâmetro de entrada para processos que recebem a mensagem;
- **count**: Número (não-negativo) de elementos contidos no *buffer* de recebimento;
- **datatype**: Tipo de dados de cada elemento contido no *buffer* de recebimento;
- **root**: *Rank* do processo de origem da mensagem;
- **comm**: Comunicador utilizado para realizar a transferência de dados.

- **MPI_Waitany**

  int `MPI_Waitany( int count, MPI_Request *vetor_de_respostas, int *index, MPI_Status *status );`

  Função utilizada para aguardar a finalização de uma ou mais operação assíncrona (por exemplo: `MPI_Irecv()`). Esta finalização ocorre através da requisição de resposta (identificada por *index*) contida no *vetor_de_respostas* (definido nas funções assíncronas). A quantidade de respostas que serão aguardadas é definida em *count*.

  Parâmetros de Entrada

  - **count**: Inteiro que indica a quantidade total de respostas que serão aguardadas;
  - **vetor_de_respostas**: Vetor contendo todas as requisições de resposta que serão aguardadas.

  Parâmetros de Saída

  - **index**: Inteiro que conterá a requisição que foi recebida;
  - **status**: Objeto *status*, utilizado para recuperar informações sobre o procedimento.

A.3 Criação de Processos

- **MPI_Comm_spawn**

  int `MPI_Comm_spawn( char *bin, char *argv[], int maxprocs, MPI_Info info, int root, MPI_Comm comm, MPI_Comm *intercomm, int array_of_errcodes[] );`

  Função utilizada para a criação de processos em tempo de execução. Ao executar ela, o processo *root* criará *maxprocs* que executarão o programa *bin*. Os processos criados

receberão os parâmetros *argv[]* de entrada e comunicarão com o processo pai através do intercomunicador *intercomm*. Caso ocorra algum erro durante a operação, ele é armazenado em *array_of_codes[]*. Esta função está organizada como segue:

Parâmetros de Entrada

- **bin**: Nome do programa que o(s) processo(s) criado(s) irá/irão executar;
- **argv**: Argumentos de entrada para o programa definido em **bin**;
- **maxprocs**: Número máximo de processos que serão criados;
- **info**: Conjunto de pares do tipo valor-chave informando ao sistema de execução onde e como iniciar os processos;
- **root**: *Rank* do processo que está realizando a chamada à função;
- **comm**: Intracomunicador no qual o processo *root* faz parte.

Parâmetros de Saída

- **intercomm**: Intercomunicador no qual ocorrerão as comunicações entre o processo *root* e o(s) processo(s) criado(s);
- **array_of_errcodes**: Vetor contendo os códigos de erro (um para cada processo) em caso de falha na operação.

APÊNDICE B – Gráficos de Resultados das Análises realizadas neste Trabalho

Figura 58 – Tempos de execução das versões ADIL e ADCL para execução com 4 até 32 processos – Matriz de tamanho 2048 × 2048.

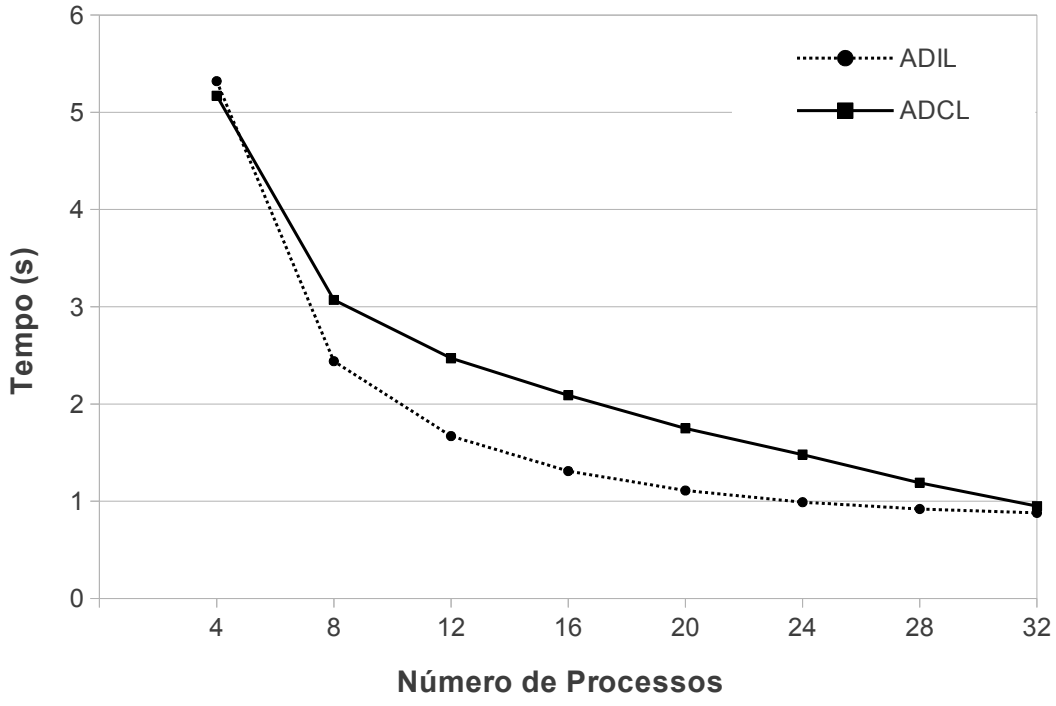


Figura 59 – Tempos de execução das versões ADIL e ADCL para execução com 4 até 32 processos – Matriz de tamanho 4096 × 4096.

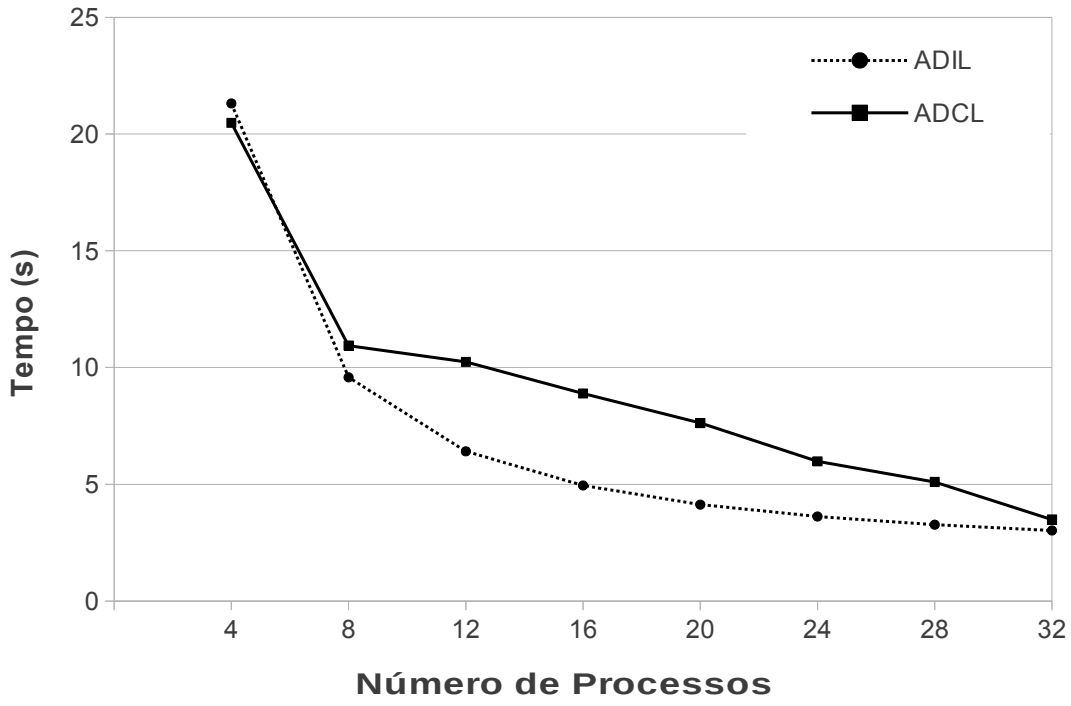


Figura 60 – Tempos de execução das versões ADIL e ADCL para execução com 4 até 32 processos – Matriz de tamanho 6144 × 6144.

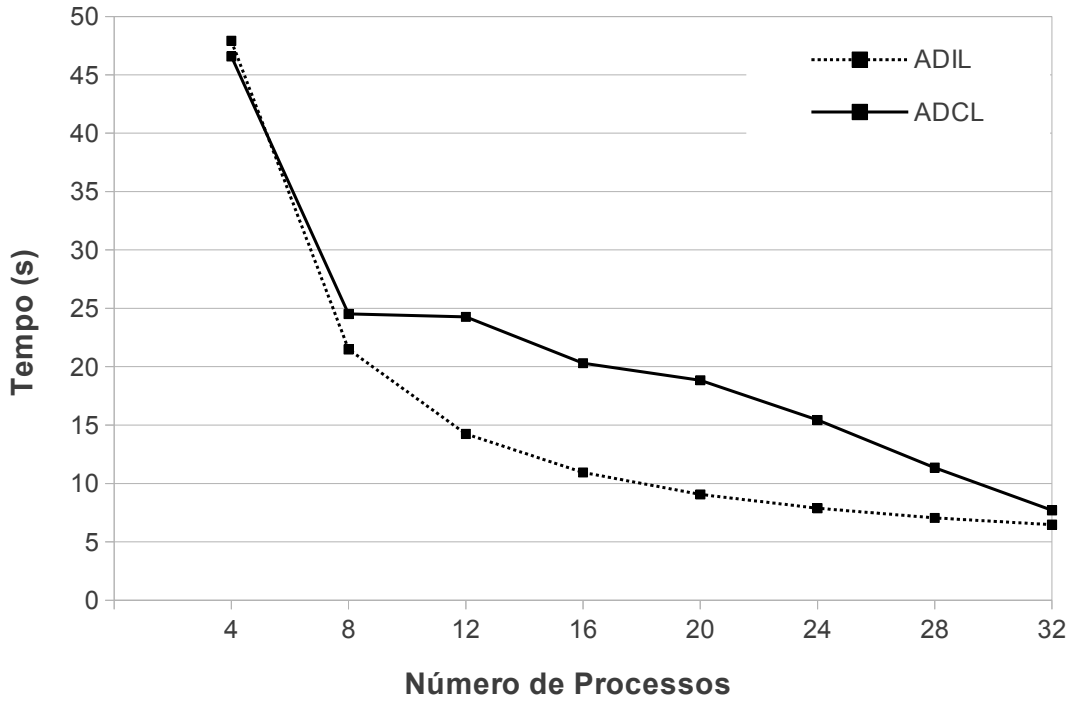


Figura 61 – Tempos de execução das versões ADIL e ADCL para execução com 4 até 32 processos – Matriz de tamanho 8192 × 8192.

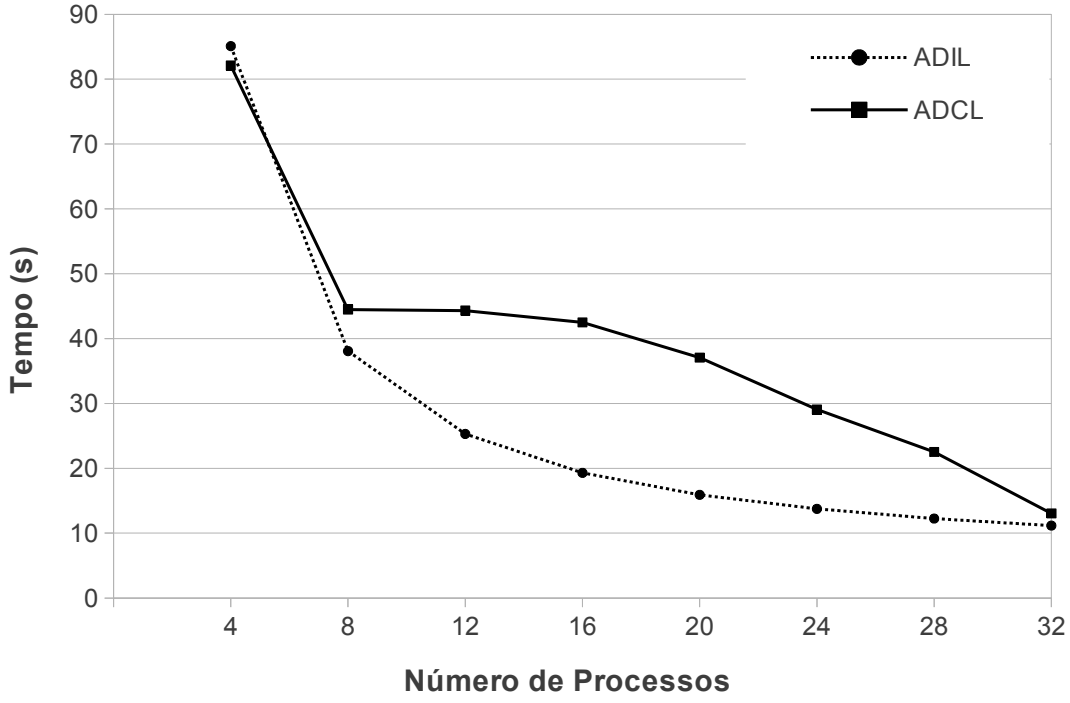


Figura 62 – Tempos de execução das versões ADIL e ADCL para execução com 4 até 32 processos – Matriz de tamanho 16384 × 16384.

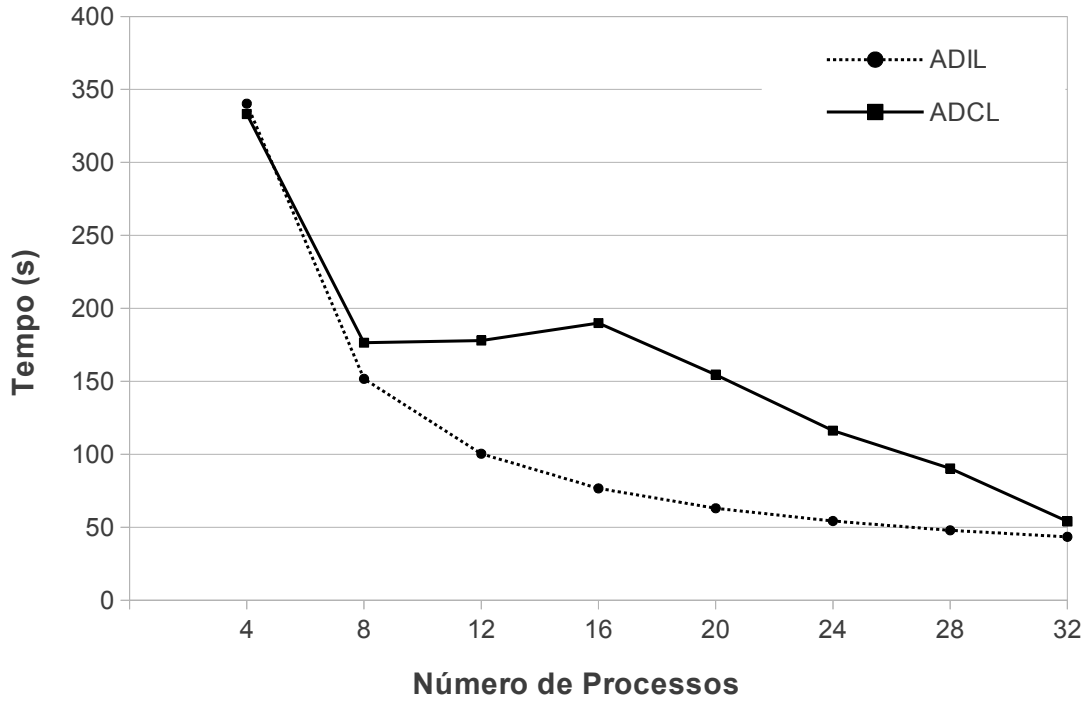


Figura 63 – Tempos de execução (em segundos) de 4 a 32 processos para ambos os cenários – Matriz de tamanho 2048 × 2048.

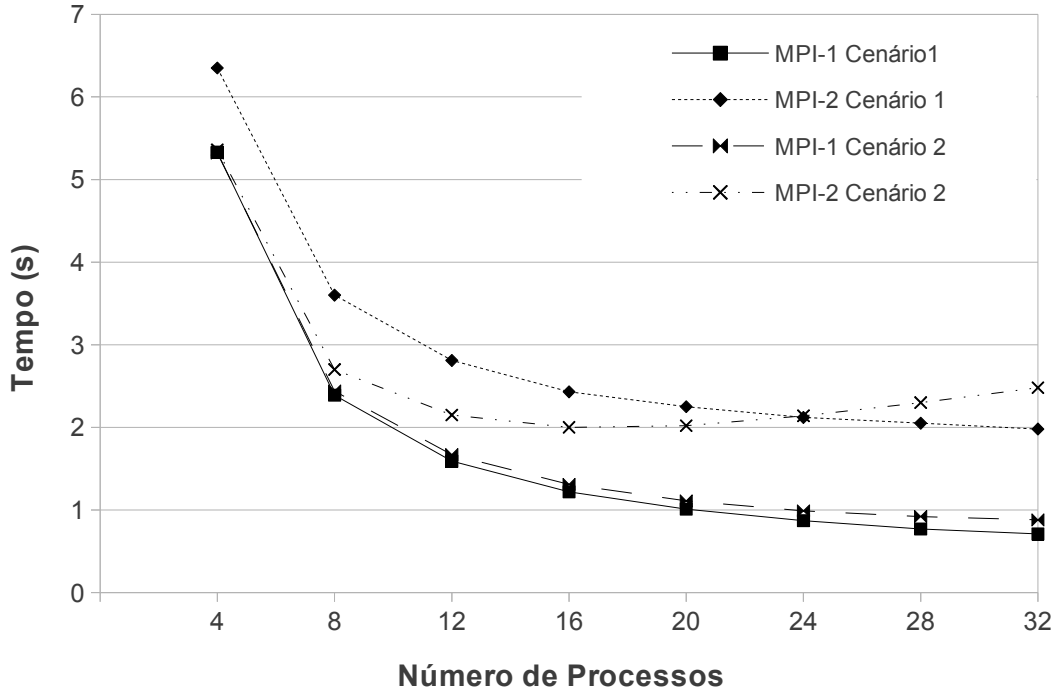


Figura 64 – Tempos de execução (em segundos) de 4 a 32 processos para ambos os cenários – Matriz de tamanho 4096 × 4096.

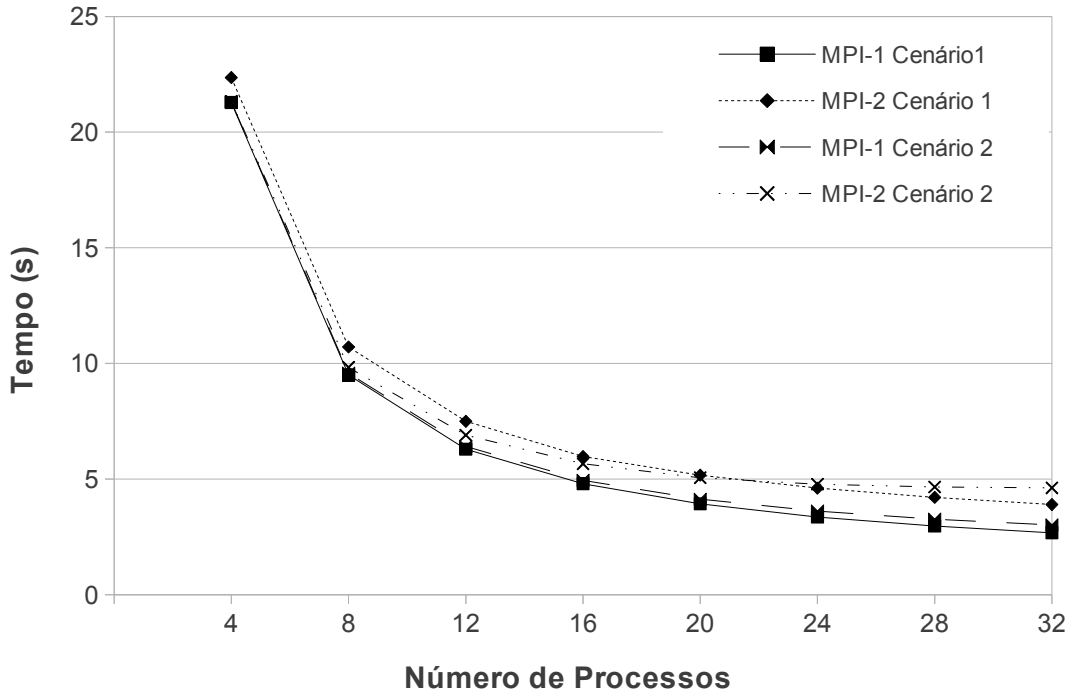


Figura 65 – Tempos de execução (em segundos) de 4 a 32 processos para ambos os cenários – Matriz de tamanho 6144 × 6144.

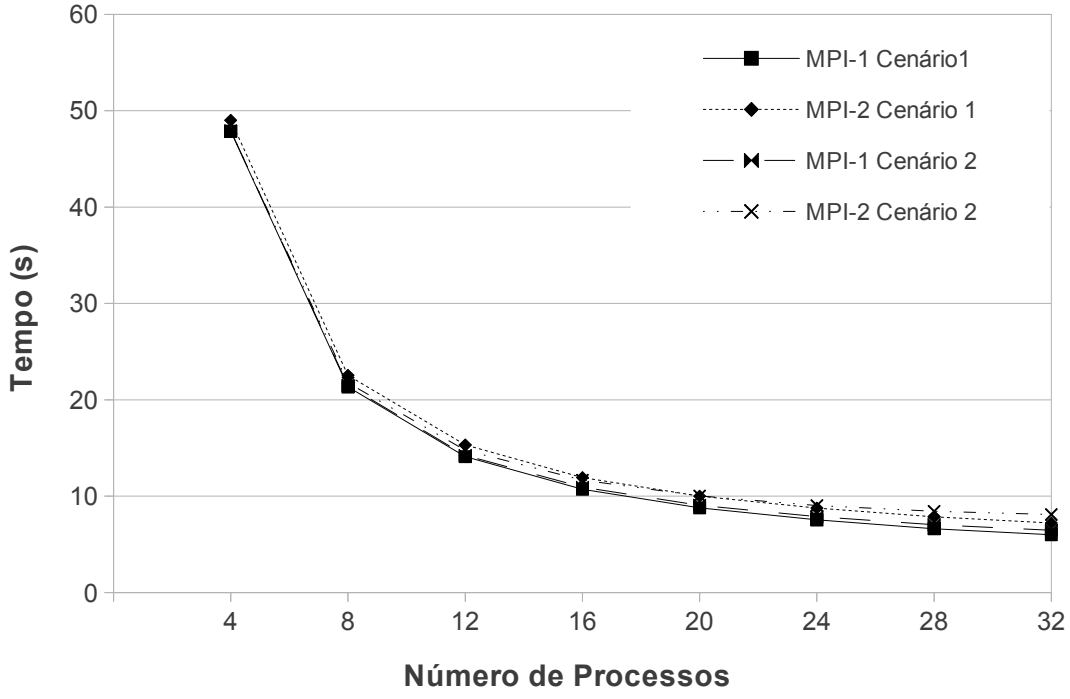


Figura 66 – Tempos de execução (em segundos) de 4 a 32 processos para ambos os cenários – Matriz de tamanho 8192 × 8192.

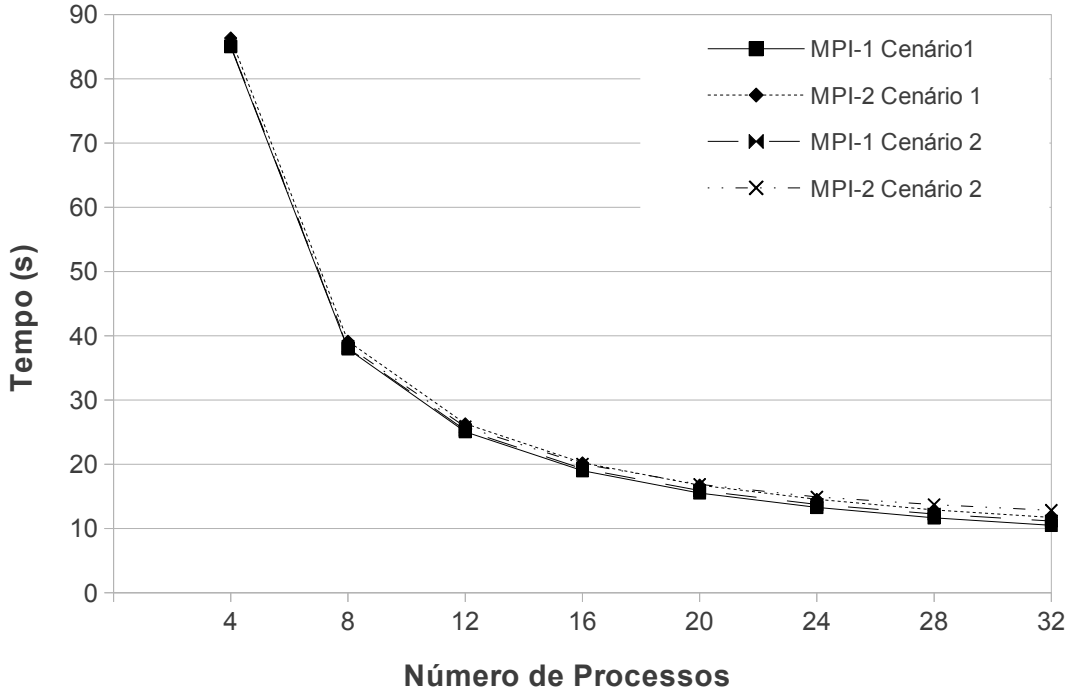


Figura 67 – Tempos de execução (em segundos) de 4 a 32 processos para ambos os cenários – Matriz de tamanho 16384 × 16384.

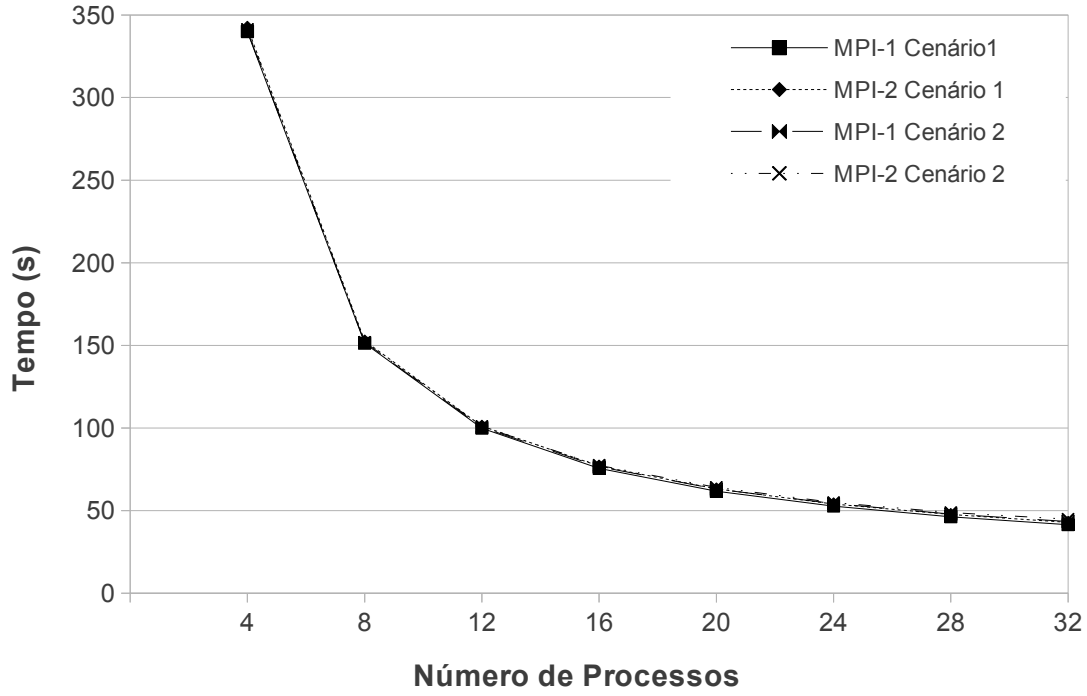


Figura 68 – Tempos de execução em segundos para 4 processos principais e 2 auxiliares.

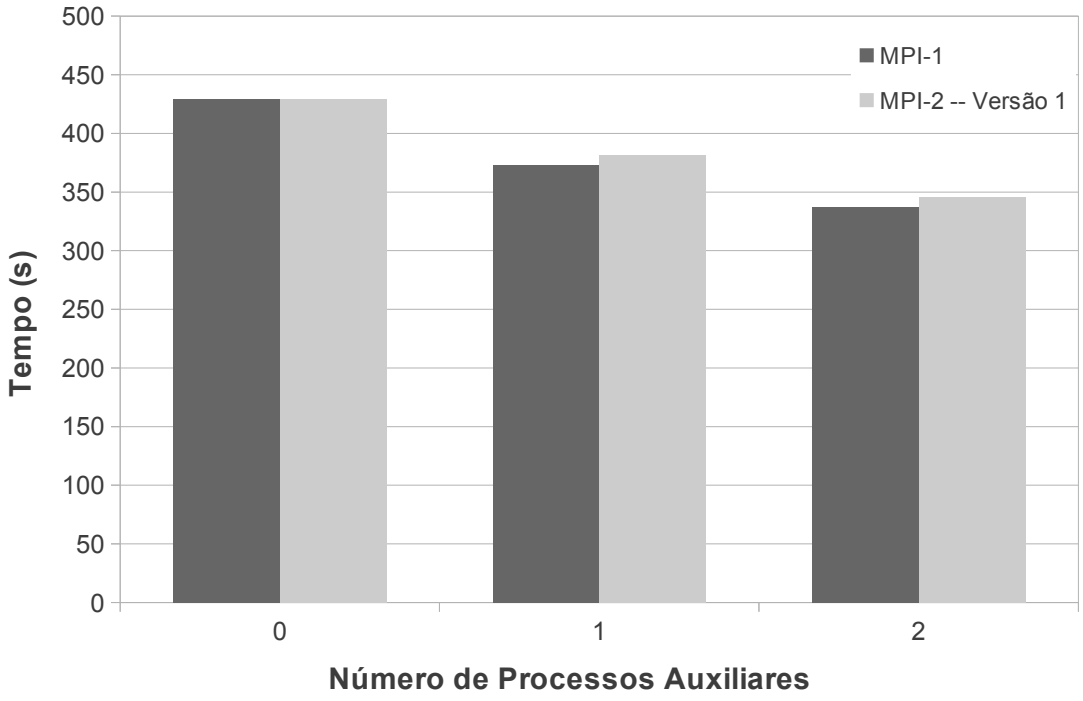


Figura 69 – Tempos de execução em segundos para 8 processos principais e 6 auxiliares.

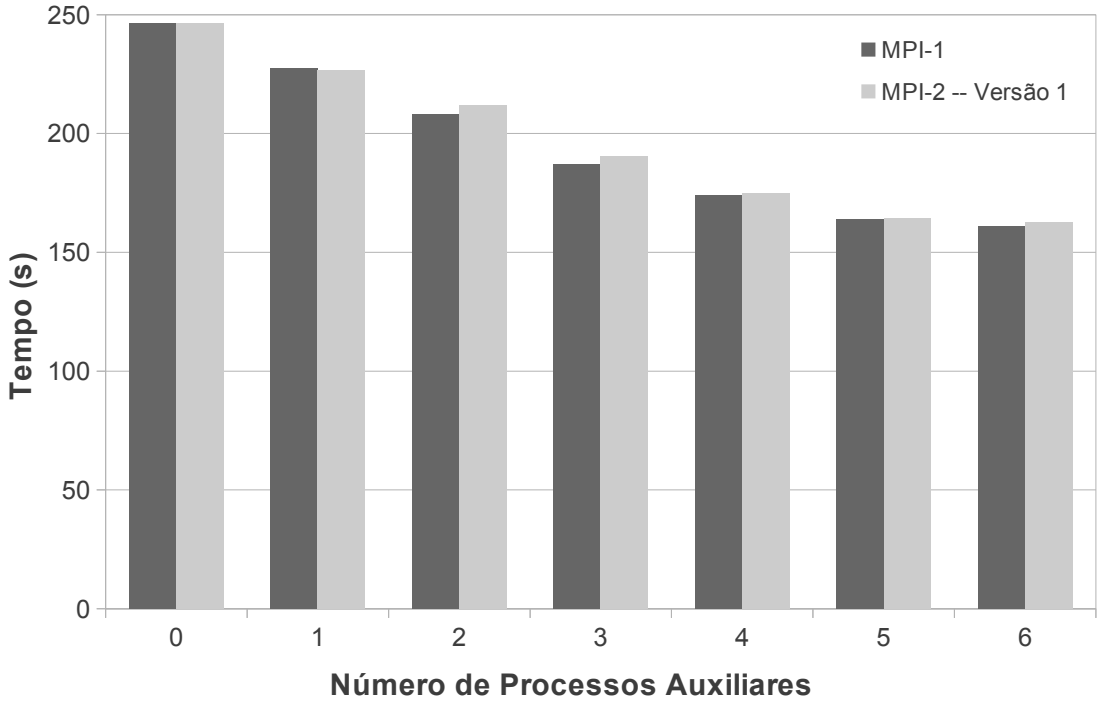


Figura 70 – Tempos de execução em segundos para 12 processos principais e 10 auxiliares.

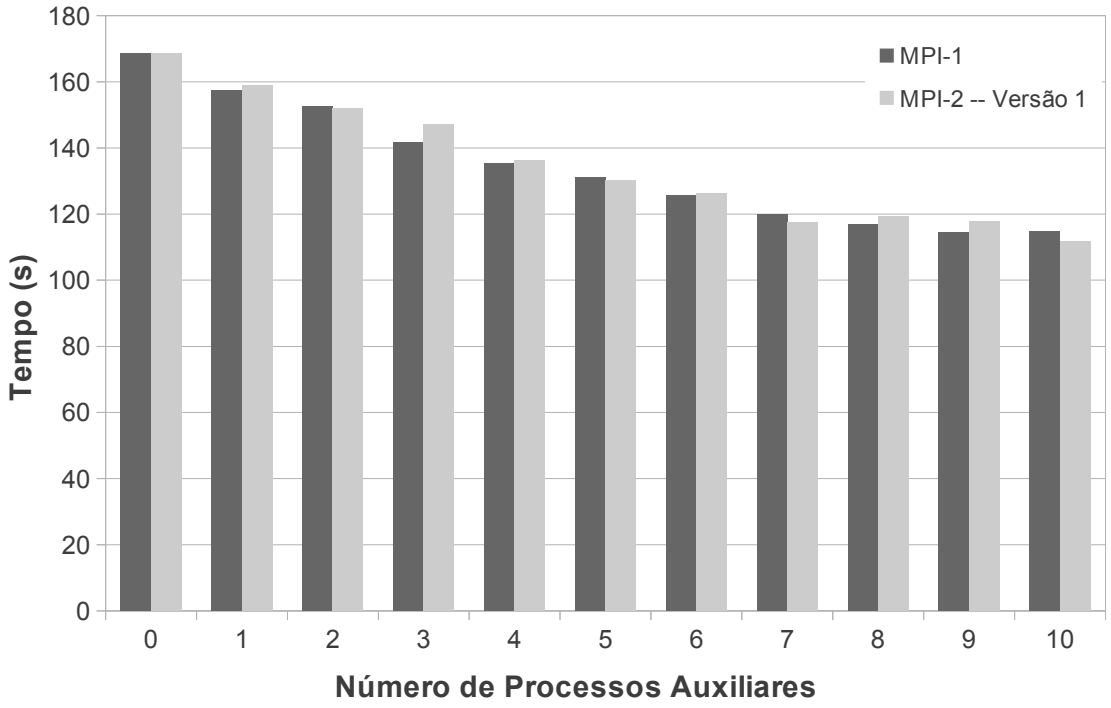


Figura 71 – Tempos de execução em segundos para 16 processos principais e 14 auxiliares.

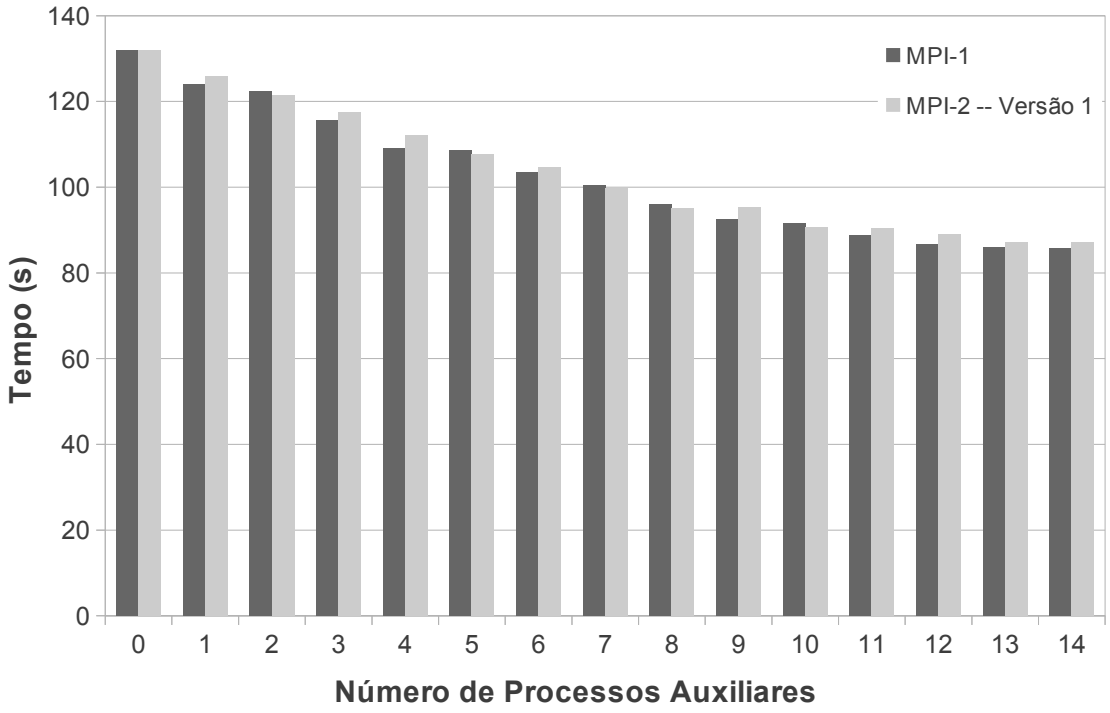


Figura 72 – Tempos de execução em segundos para 20 processos principais e 12 auxiliares.

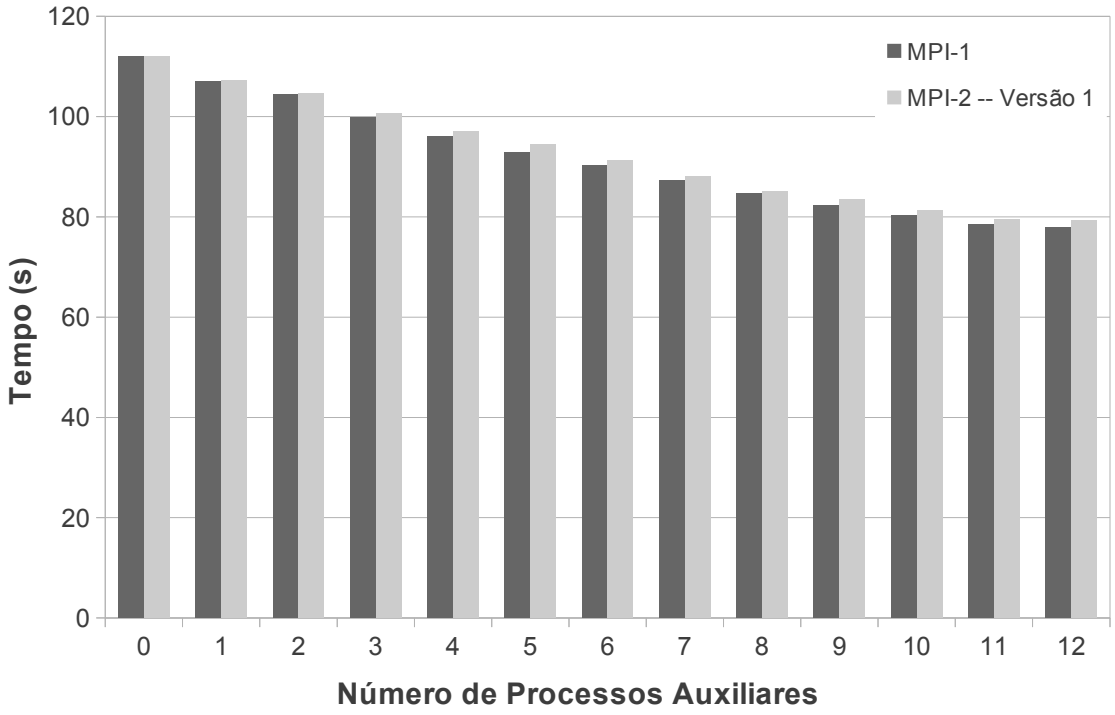


Figura 73 – Tempos de execução em segundos para 24 processos principais e 8 auxiliares.

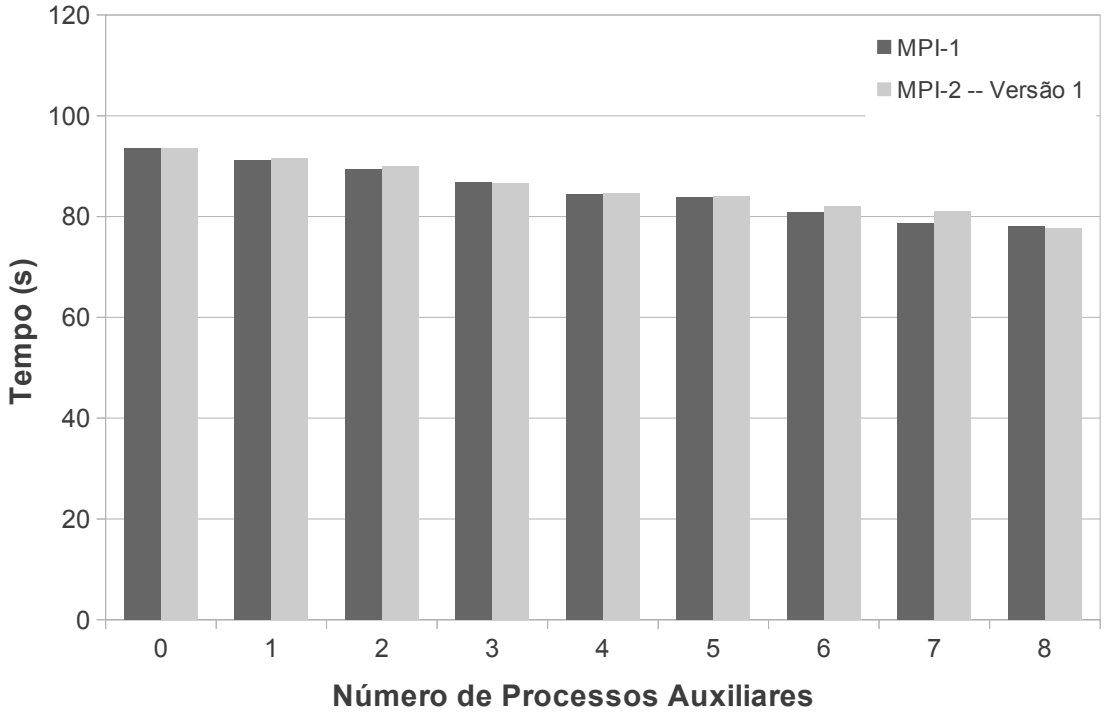


Figura 74 – Tempos de execução em segundos para 28 processos principais e 4 auxiliares.