

**PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL  
FACULDADE DE INFORMÁTICA  
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO**

**UM PROCESSO DE GERAÇÃO  
AUTOMÁTICA DE CÓDIGO  
PARALELO PARA  
ARQUITETURAS HÍBRIDAS  
COM AFINIDADE DE MEMÓRIA**

**MATEUS RAEDER**

Tese apresentada como requisito parcial  
à obtenção do grau de Doutor em  
Ciência da Computação na Pontifícia  
Universidade Católica do Rio Grande do  
Sul.

Orientador: Prof. Dr. Luiz Gustavo Leão Fernandes

**Porto Alegre  
2017**



## Ficha Catalográfica

R134p Raeder, Mateus

Um Processo de Geração Automática de Código Paralelo para  
Arquiteturas Híbridas com Afinidade de Memória / Mateus Raeder  
. – 2014.

129 f.

Tese (Doutorado) – Programa de Pós-Graduação em Ciência da  
Computação, PUCRS.

Orientador: Prof. Dr. Luiz Gustavo Leão Fernandes.

1. Programação Híbrida. 2. Afinidade de Memória. 3. Cluster de  
NUMA. I. Fernandes, Luiz Gustavo Leão. II. Título.








Pontifícia Universidade Católica do Rio Grande do Sul  
FACULDADE DE INFORMÁTICA  
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

## TERMO DE APRESENTAÇÃO DE TESE DE DOUTORADO

Tese intitulada "Um Processo de Geração Automática de Código Paralelo para Arquiteturas Híbridas com Afinidade de Memória", apresentada por Mateus Raeder, como parte dos requisitos para obtenção do grau de Doutor em Ciência da Computação, aprovada em 27/08/2014 pela Comissão Examinadora:

  
\_\_\_\_\_  
Prof. Dr. Luiz Gustavo Leão Fernandes  
Orientador PPGCC/PUCRS

  
\_\_\_\_\_  
Prof. Dr. Paulo Henrique Lemelle Fernandes PPGCC/PUCRS

  
\_\_\_\_\_  
Prof. Dr. Adenauer Correa Yamin UFPEL

  
\_\_\_\_\_  
Prof. Dr. Claudio Fernando Resin Geyer UFRGS

Homologada em ...../...../....., conforme Ata No. .... pela Comissão Coordenadora.

  
\_\_\_\_\_  
Prof. Dr. Luiz Gustavo Leão Fernandes  
Coordenador.

**PUCRS**

**Campus Central**

Av. Ipiranga, 6681 – P. 32 – sala 507 – CEP: 90619-900  
Fone: (51) 3320-3611 – Fax (51) 3320-3621  
E-mail: [ppgcc@pucrs.br](mailto:ppgcc@pucrs.br)  
[www.pucrs.br/facin/pos](http://www.pucrs.br/facin/pos)



## **DEDICATÓRIA**

Dedico este trabalho aos meus pais, Rogério e Marcia, e à minha esposa, Jordana.





“Um plano só é bom nas mãos de quem pode executá-lo.”  
(Mestre Yoda)



## AGRADECIMENTOS

Gratidão é um sentimento difícil de expressar através de um simples texto. Entretanto, é impossível deixar de agradecer a algumas pessoas que foram fundamentais para esta conquista.

A Deus, por abençoar todas as fases da minha vida, inclusive esta. Se não fosse pela vontade dEle, nada disto seria possível.

A meus pais, Marcia e Rogério, que sempre fizeram de tudo para que eu seguisse no caminho correto e me incentivaram incessantemente durante toda a minha vida. Muito obrigado por tudo que vocês fizeram/fazem por mim. Palavras são poucas para descrever o que vocês representam. Amo muito vocês!

À minha esposa, Jordana, que vivenciou comigo noites mal dormidas (ou nem dormidas), dias de correria, nervosismo, ansiedade e tudo mais o que acontece dentro da gente durante o desenvolvimento de uma tese. Obrigado por todo o incentivo e pensamento positivo. Certamente, este trabalho não estaria finalizado se tu não estivesse ao meu lado. Muito obrigado! Te amo!

Ao meu orientador, Gustavo, pela parceria e amizade de sempre. Muito obrigado pelos conselhos, por nortear minha vida acadêmica desde muito tempo e, principalmente, por acreditar em mim. Muito obrigado!

Aos meus amigos (e dindos) Andrielle e Ricardo, que são pessoas sem explicação. Sem sombra de dúvidas, não teria completado esta parte da minha vida se não fosse por vocês. Obrigado por me acompanharem nas madrugadas alucinantes de escrita, testes, revisões, formatações, Outbacks, ideias e muitas risadas. Este trabalho tem muito de vocês! Obrigado por tudo, principalmente pela amizade e altruísmo de sempre!

Tive ótimos professores durante a graduação, mas três foram os que mais tiveram contribuição para este trabalho e para minha formação acadêmica e profissional. Um deles é meu professor orientador, Gustavo, citado anteriormente. Os outros dois professores são o professor De Rose e o professor Paulo Fernandes. Embora talvez vocês não saibam, mas vocês foram peças fundamentais na minha caminhada, e aprendi muito com vocês. Agradeço pelo acompanhamento durante minha jornada acadêmica, desde as disciplinas cursadas até as propostas de tese, defesas de mestrado e doutorado e tudo mais. Muito obrigado!

Finalmente, a todos os que de alguma forma acreditaram em mim e me apoiaram nesta formação, meu muito obrigado!

# UM PROCESSO DE GERAÇÃO AUTOMÁTICA DE CÓDIGO PARALELO PARA ARQUITETURAS HÍBRIDAS COM AFINIDADE DE MEMÓRIA

## RESUMO

Nos últimos anos, avanços tecnológicos têm disponibilizado máquinas com diferentes níveis de paralelismo, produzindo um grande impacto na área de processamento de alto desempenho. Estes avanços permitiram aos desenvolvedores melhorar ainda mais o desempenho de aplicações de grande porte. Neste contexto, a criação de clusters de máquinas multiprocessadas com acesso não uniforme à memória (NUMA - Non-Uniform Memory Access), surge como uma tendência. Em uma arquitetura NUMA, o tempo de acesso a um dado depende de sua localização na memória. Por este motivo, gerenciar a localização dos dados é essencial em máquinas deste tipo. Neste cenário, o desenvolvimento de software para um cluster de máquinas NUMA deve explorar tanto a parte internodo (multi-computador, com memória distribuída) quanto a parte intranodo (multiprocessador, memória compartilhada) desta arquitetura. Este tipo de programação híbrida faz melhor uso dos recursos disponibilizados por arquiteturas NUMA. Entretanto, reescrever uma aplicação sequencial de modo que explore o paralelismo do ambiente de forma correta não é uma tarefa trivial, mas que pode ser facilitada através de um processo automatizado. Neste sentido, o presente trabalho apresenta um processo de geração automática e transparente de aplicações paralelas híbridas, sem que o usuário precise conhecer as rotinas de baixo nível das bibliotecas de programação paralela. Foi desenvolvida então, uma ferramenta gráfica para que o usuário crie seu modelo paralelo de forma dinâmica e intuitiva. Assim, é possível criar programas paralelos de tal forma que não é necessário ser familiarizado com bibliotecas comumente utilizadas por profissionais da área de alto desempenho (como o MPI, por exemplo). Através da ferramenta desenvolvida, o usuário desenha um grafo dirigido para indicar a quantidade de processos (nodos do grafo) e as formas de comunicação entre eles (arestas). A partir desse desenho, o usuário insere o código sequencial de cada processo

definido na interface gráfica, e a ferramenta gera o código paralelo correspondente. Além disto, mapeamentos de processos pesados e de memória foram definidos e testados em um cluster de máquinas NUMA, bem como um mapeamento híbrido. A ferramenta foi desenvolvida em Java e gera código paralelo com MPI em C++, além de aplicar políticas de afinidade de memória para máquinas NUMA através da biblioteca MAI (Memory Affinity Interface). Algumas aplicações foram desenvolvidas com e sem a utilização do modelo. Os resultados demonstram que o mapeamento proposto é válido, já que houve ganho de desempenho em relação às versões sequenciais, além de um comportamento similar a implementações paralelas tradicionais.

**Palavras-Chave:** Programação Híbrida, Afinidade de Memória, Cluster de NUMA.

# AN AUTOMATIC PARALLEL CODE GENERATION PROCESS FOR HYBRID ARCHITECTURES USING MEMORY AFFINITY

## ABSTRACT

Over the last years, technological advances provide machines with different levels of parallelism, producing a great impact in high-performance computing area. These advances allowed developers to improve further the performance of large scale applications. In this context, clusters of multiprocessor machines with Non-Uniform Memory Access (NUMA) are a trend in parallel processing. In NUMA architectures, the access time to data depends on where it is placed in memory. For this reason, managing data location is essential in this type of machine. In this scenario, developing software for a cluster of NUMA machines must explore the internode part (multicomputer, with distributed memory) and the intranode part (multiprocessor, with shared memory) of this architecture. This type of hybrid programming takes advantage of all features provided by NUMA architectures. However, rewriting a sequential application so that it exploits the parallelism of the environment correctly is not a trivial task, but can be facilitated through an automated process. In this sense, our work presents an automatic parallel code generation process for hybrid architectures. With the proposed approach, users do not need to know low level routines of parallel programming libraries. In order to do so, we developed a graphical tool, in which users can dynamically and intuitively create their parallel models. Thereby, it is possible to create parallel programs in such a way that is not required to be familiar with libraries commonly used by professionals of high performance computing area (such as MPI, for example). By using the developed tool, user draws a directed graph to indicate the number of processes (nodes of the graph) and the communication between them (edges). From this drawing, user inserts the sequential code of each process defined in the graphical interface, and the tool automatically generates the corresponding parallel code. Moreover, weight process and memory mappings were defined and tested on a NUMA machine cluster, as well as a hybrid mapping.

The tool was developed in Java and generates parallel code with MPI for C++, in the same way that it applies memory affinity policies for NUMA machines through the Memory Affinity Interface (MAI) library. Some applications were developed with and without our model. The obtained results evidence that the proposed mapping is valid, providing performance gains in relation to sequential versions and behaving in a very similar way to traditional parallel implementations.

**Keywords:** Hybrid Programming, Memory Affinity, Cluster of NUMA.



## LISTA DE FIGURAS

Figura 2.1 – Divisão de tarefas em uma arquitetura híbrida. . . . .	36
Figura 3.1 – Interface Gráfica do Hence. . . . .	49
Figura 3.2 – Exemplo de utilização do CODE 2.0 . . . . .	49
Figura 3.3 – Interface gráfica do Paralex . . . . .	50
Figura 3.4 – Interface gráfica do VPE . . . . .	51
Figura 3.5 – Exemplo da interface do GRAPNEL . . . . .	52
Figura 3.6 – Interface gráfica do GRIX . . . . .	52
Figura 3.7 – Exemplo de interface do VisualGOP . . . . .	53
Figura 3.8 – Interface gráfica do VIDIM . . . . .	54
Figura 3.9 – Interface gráfica do Kaira . . . . .	55
Figura 3.10 – Interface gráfica do GD-MPI . . . . .	56
Figura 4.1 – Visão geral do processo de mapeamento de processos pesados com padrões . . . . .	59
Figura 4.2 – Exemplos de fluxos de comunicação possíveis . . . . .	60
Figura 4.3 – Comunicação entre um processo e um grupo de processos . . . . .	61
Figura 4.4 – Exemplo de redundância de dados (“borda”) . . . . .	62
Figura 4.5 – Exemplo de estrutura de arquivo template gerado para cada processo	63
Figura 4.6 – Estrutura interna do mapeamento de processos pesados . . . . .	64
Figura 4.7 – Comunicação bidirecional entre um processo e um grupo de processos	65
Figura 4.8 – Exemplo de Pipe com 5 processos . . . . .	67
Figura 4.9 – Exemplo de <i>template</i> gerado . . . . .	68
Figura 4.10 – Exemplo de código: recebimento de <i>array</i> bidimensional de inteiros .	72
Figura 5.1 – Visão geral do mapeamento de memória . . . . .	75
Figura 5.2 – Exemplo de arquivo de configuração da MAI . . . . .	76
Figura 5.3 – Exemplo de código transformado, utilizando o arquivo <code>memory_management.h</code>	77
Figura 6.1 – Visão geral do mapeamento híbrido . . . . .	79
Figura 6.2 – Visão geral do mapeamento híbrido . . . . .	80
Figura 6.3 – Exemplo de um grafo com 5 nodos . . . . .	81
Figura 6.4 – Template gerado pelo mapeamento de processos pesados com pa- drões . . . . .	82
Figura 6.5 – REFER. CODE . . . . .	82
Figura 7.1 – Tela inicial do protótipo desenvolvido . . . . .	84

Figura 7.2 – xemplo de grafo no protótipo desenvolvido . . . . .	84
Figura 7.3 – Janela de edição de código de cada processo . . . . .	87
Figura 7.4 – Exemplo de <i>HelloWorld</i> . . . . .	88
Figura 7.5 – Exemplo de grafos de diferentes aplicações . . . . .	89
Figura 8.1 – <i>N – Queens</i> (16x16) . . . . .	94
Figura 8.2 – <i>CountElements</i> para o tamanho 15000000000 . . . . .	96
Figura 8.3 – Resultado para a aplicação <i>CountElements</i> para o tamanho 15000000000 sem envio do array pelo processo <i>Master</i> . . . . .	99
Figura 8.4 – Resultados da aplicação <i>FindText</i> para 1000000000 textos . . . . .	100
Figura 8.5 – Resultado da aplicação <i>MultMatrix</i> para matrizes de dimensões 2000x2000, divididas em tarefas . . . . .	103
Figura 8.6 – Resultados obtidos para a aplicação de Filtro de Imagens . . . . .	103
Figura 8.7 – Resultado da aplicação <i>Search</i> . . . . .	108
Figura 8.8 – Resultados da aplicação <i>MatrixMult</i> utilizando políticas de alocação de memória . . . . .	108
Figura 8.9 – Resultados obtidos pela aplicação <i>Mandelbrot</i> utilizando política de alocação de memória . . . . .	110
Figura 8.10 – Resultados obtidos com a aplicação <i>Dijkstra</i> para um grafo de 40000 nodos . . . . .	110
Figura 8.11 – Tempos obtidos com a aplicação híbrida <i>MultMatrix</i> com 2 threads por processo . . . . .	111
Figura B.1 – <i>CountElements</i> para o tamanho 10000000000. . . . .	129
Figura B.2 – <i>CountElements</i> para o tamanho 5000000000 . . . . .	129

## LISTA DE TABELAS

Tabela 4.1 – Comparativo entre os padrões implementados . . . . .	73
Tabela 8.1 – Tempos obtidos com a aplicação <i>N – Queens</i> para $N=16$ . . . . .	95
Tabela 8.2 – Tempos obtidos para a aplicação <i>CountElements</i> para o tamanho 15000000000 . . . . .	97
Tabela 8.3 – Tempos obtidos para <i>CountElements</i> com tamanho 15000000000 sem envio do array pelo processo <i>Master</i> . . . . .	98
Tabela 8.4 – Tempos obtidos com a aplicação <i>FindText</i> para 1000000000 textos .	100
Tabela 8.5 – Tempos obtidos para a aplicação <i>MultMatrix</i> para matrizes de dimensões 2000x2000 . . . . .	101
Tabela 8.6 – Tempos obtidos para a aplicação <i>Crypt</i> para 1000000000 textos . . .	104
Tabela 8.7 – Tempos obtidos para a aplicação <i>BubbleSort</i> com array de tamanho 1000000 . . . . .	105
Tabela 8.8 – Tempos obtidos para a aplicação <i>QuickSort</i> com array de tamanho 1000000 . . . . .	106
Tabela 8.9 – Tempos para a aplicação <i>MatrixMult</i> utilizando diferentes políticas de alocação de memória . . . . .	109
Tabela 8.10 – Tempos obtidos pela aplicação <i>MultMatrix</i> híbrida com 2 <i>threads</i> por nodo. . . . .	112
Tabela 8.11 – Tempos obtidos pela aplicação <i>MultMatrix</i> híbrida com 20 <i>threads</i> por nodo. . . . .	112
Tabela 9.1 – Diferenças (em segundos) dos tempos obtidos com o mapeamento proposto. Valores positivos indicam um aumento no tempo de execução da aplicação, enquanto valores negativos indicam diminuição neste mesmo tempo. . . . .	116
Tabela A.1 – Tempos obtidos com a aplicação <i>N – Queens</i> para $N=4$ . . . . .	127
Tabela A.2 – Tempos obtidos com a aplicação <i>N – Queens</i> para $N=8$ . . . . .	127
Tabela A.3 – Tempos obtidos com a aplicação <i>N – Queens</i> para $N=12$ . . . . .	128



## LISTA DE SIGLAS

NUMA – *Non-Uniform Memory Access*

API – *Application Programming Interface*

LIG – *Laboratoire d'Informatique de Grenoble*

INRIA – *Institut National de Recherche en Informatique et en Automatique*

MAI – *Memory Affinity Interface*

GUI – *Graphical User Interface*

MPI – *Message Passing Interface*

UMA – *Uniform Memory Access*

COMA – *Cache-Only Memory Architecture*

PVM – *Parallel Virtual Machine*

SPMD – *Single Processor Multiple Data*

MPE – *MPI Parallel Environment*

CPU – *Central Processing Unit*

POSIX – *Portable Operating System Interface*

OPENMP – *Open Multi-Processing*

IDE – *Integrated Development Environment*



# SUMÁRIO

<b>1</b>	<b>INTRODUÇÃO</b>	<b>25</b>
1.1	MOTIVAÇÃO	27
1.2	JUSTIFICATIVA	27
1.3	OBJETIVOS E CONTRIBUIÇÕES	28
1.4	METODOLOGIA DE PESQUISA	30
1.4.1	QUESTÕES DE PESQUISA	30
1.4.2	HIPÓTESES	31
1.4.3	DESENVOLVIMENTO E AVALIAÇÃO	31
1.5	ESTRUTURA DO VOLUME	32
<b>2</b>	<b>PRESSUPOSTOS CONCEITUAIS</b>	<b>35</b>
2.1	ARQUITETURAS PARALELAS HÍBRIDAS	35
2.2	ARQUITETURAS NUMA	37
2.3	PROGRAMAÇÃO PARALELA DE ALTO DESEMPENHO	38
2.3.1	BIBLIOTECAS	39
2.3.2	PROGRAMAÇÃO HÍBRIDA	42
2.4	PADRÕES DE PROGRAMAÇÃO PARALELA ( <i>SKELETONS</i> )	43
<b>3</b>	<b>TRABALHOS RELACIONADOS</b>	<b>45</b>
3.1	NUMA	45
3.2	SKELETONS	47
3.3	PROGRAMAÇÃO PARALELA X INTERFACE GRÁFICA	48
3.4	CONSIDERAÇÕES FINAIS	56
<b>4</b>	<b>MAPEAMENTO DE PROCESSOS PESADOS COM PADRÕES</b>	<b>59</b>
4.1	VISÃO GERAL DO PROCESSO DE MAPEAMENTO	59
4.2	ESTRUTURA E CONTROLE	63
4.3	PADRÕES IMPLEMENTADOS ( <i>SKELETONS</i> )	64
4.3.1	MASTER/SLAVE	65
4.3.2	PIPE	67
4.3.3	DIVIDE AND CONQUER ( <i>D&amp;C</i> )	69
4.3.4	MODELO GENÉRICO	70

4.3.5	COMPARATIVO ENTRE OS PADRÕES .....	72
4.4	OBTENDO INFORMAÇÕES GLOBAIS .....	73
4.5	CONSIDERAÇÕES FINAIS .....	74
<b>5</b>	<b>MAPEAMENTO DE MEMÓRIA .....</b>	<b>75</b>
5.1	VISÃO GERAL DO PROCESSO DE MAPEAMENTO .....	75
5.2	ESTRUTURA E CONTROLE .....	77
<b>6</b>	<b>MAPEAMENTO HÍBRIDO .....</b>	<b>79</b>
6.1	VISÃO GERAL DO MAPEAMENTO HÍBRIDO .....	79
6.2	EXEMPLO SIMPLES DE UTILIZAÇÃO .....	81
6.3	CONSIDERAÇÕES FINAIS .....	82
<b>7</b>	<b>PROTÓTIPO DESENVOLVIDO .....</b>	<b>83</b>
7.1	INTERFACE GRÁFICA COM O USUÁRIO .....	83
7.2	GERADOR DE CÓDIGO .....	85
7.3	EDITANDO O CÓDIGO DE CADA PROCESSO .....	86
7.4	EXEMPLOS DE UTILIZAÇÃO .....	87
<b>8</b>	<b>EXPERIMENTOS E RESULTADOS OBTIDOS .....</b>	<b>91</b>
8.1	AMBIENTES DE TESTE .....	91
8.2	METODOLOGIA .....	91
8.3	AVALIAÇÃO DOS RESULTADOS .....	93
8.3.1	PROCESSOS PESADOS .....	93
8.3.2	MEMÓRIA .....	107
8.3.3	HÍBRIDO .....	111
<b>9</b>	<b>CONCLUSÕES E TRABALHOS FUTUROS .....</b>	<b>115</b>
9.1	TRABALHOS FUTUROS .....	118
	<b>REFERÊNCIAS .....</b>	<b>121</b>
	<b>APÊNDICE A – Resultado <i>N – Queens</i> .....</b>	<b>127</b>
	<b>APÊNDICE B – Resultado <i>CountElements</i> .....</b>	<b>129</b>



## 1. INTRODUÇÃO

Desde a criação dos primeiros computadores, existe uma preocupação em torná-los mais eficientes com o propósito de conseguir resolver problemas computacionais no menor tempo possível. Entretanto, limitações no avanço das tecnologias dos dispositivos eletrônicos, tais como memória e processador por exemplo [Zom96], restringem o desempenho de aplicações com alta carga computacional. Com isso, computadores significativamente mais rápidos (supercomputadores) foram criados, possuindo um custo cada vez maior e tornando-se acessíveis apenas a um pequeno e seletivo grupo de pessoas.

Surgem, então, os agregados de máquinas, que agrupam diversos computadores pessoais com o intuito de aumentar as unidades de processamento disponíveis para realizar a computação de maneira paralela. Esta arquitetura, que apresenta um custo muito mais baixo em comparação aos supercomputadores, é chamada de multicomputador (ou *cluster*). As aplicações, então, podem ter suas tarefas divididas entre as diferentes máquinas do *cluster*, que as executam em paralelo e geram um resultado final em um tempo menor.

Contudo, não foi somente através da interligação de diversos computadores pessoais que mais núcleos de processamento começaram a ser utilizados. No decorrer do tempo, computadores pessoais passaram a apresentar mais de um processador, aumentando internamente seu poder de processamento e permitindo que em uma mesma máquina ocorram diferentes processamentos em paralelo. Estas máquinas com mais de um processador são conhecidas como multiprocessadores. De maneira análoga, as aplicações podem ser subdivididas entre os diferentes processadores disponíveis, que em paralelo podem chegar a um resultado em um menor tempo.

Esta evolução proporcionou a transformação de aplicações que antes eram executadas sequencialmente em aplicações que agora são executadas em paralelo. Tornar um código sequencial em um código paralelo, no entanto, não é trivial. A programação para estes ambientes exige um bom conhecimento da arquitetura que será utilizada, pois a forma de comunicação entre as unidades de processamento de um *cluster* é muito diferente daquela encontrada em um multiprocessador. A sincronização e organização do fluxo de execução da aplicação interfere diretamente nos resultados obtidos, pois o programador deve, explicitamente, controlar como e quando ocorre a comunicação entre os processos.

Um *cluster* permite acelerar a execução de aplicações, que são divididas em tarefas e enviadas para as máquinas que o compõem, da mesma forma que um multiprocessador pode tornar a aplicação mais rápida através da utilização de seus núcleos de processamento em paralelo. Assim, naturalmente, o próximo passo no contexto do processamento paralelo de alto desempenho foi a união de diversas máquinas multiprocessadas – cada uma delas possuindo dois ou mais núcleos de processamento – em um multicomputa-

dor (*cluster*). Esta união criou uma nova categoria de arquitetura paralela, uma arquitetura híbrida, que proporciona um nível de paralelismo maior aos desenvolvedores.

Dentro deste cenário, uma arquitetura de multiprocessadores que merece uma atenção especial é a arquitetura NUMA (*Non-Uniform Memory Access*). Em máquinas NUMA, como o próprio nome dá indícios, o tempo de acesso à memória não é o mesmo para todos os nós. Cada nó possui um *pool* de páginas, e quando ocorre uma requisição de alocação, é necessário que se decida em qual *pool* de páginas e em qual nó a memória será alocada [Rib11]. Apesar de comumente a memória ser alocada no nó local da requisição, existem maneiras que podem alocar a memória em outros nós [BSF+91]. Uma alocação mais eficiente de memória pode ser obtida, por exemplo, quando o processo em execução e os dados encontram-se na memória do mesmo nó. O ideal é que o nó não precise acessar a memória de outro nó remotamente, evitando a utilização da rede de interconexão (que diminui o desempenho da aplicação).

Juntamente com esta arquitetura surge a chamada programação híbrida de alto desempenho, que se refere à programação para estes ambientes. Neste tipo de programação, a união das diferentes formas de comunicação encontradas nos *clusters* com aquelas encontradas nos multiprocessadores traz vantagens para o desenvolvimento de aplicações paralelas, permitindo um melhor aproveitamento dos recursos disponibilizados pela arquitetura híbrida.

Neste contexto, existem ferramentas que podem ser utilizadas para que a alocação seja realizada de uma maneira mais inteligente. Uma delas trata-se da NUMA API [Kle04], que permite que o programa indique onde a memória deve ser alocada, por exemplo. A NUMA API é composta por alguns utilitários, tais como a biblioteca *libnuma*, a *numactl*, o *numastat* e o *numademo*. Para utilizar os recursos disponibilizados pelas arquiteturas NUMA, no LIG (*Laboratoire d'Informatique de Grenoble*), ligado ao INRIA (*Institut National de Recherche en Informatique et en Automatique*) em Grenoble, foi desenvolvida a biblioteca MAI (*Memory Affinity Interface*) [RM10], que é baseada na *libnuma*, servindo como uma ferramenta de mais alto nível que facilita a utilização das políticas de memória da NUMA API.

Entretanto, uma vez que a programação unicamente para *clusters* e para multiprocessadores não se tratava de uma tarefa trivial, a programação híbrida agrega uma dificuldade ainda maior na paralelização das aplicações. Neste contexto, a criação de um processo que facilite a programação para ambientes híbridos surge como uma alternativa atraente para os desenvolvedores de aplicações paralelas, com ou sem experiência na área.

## 1.1 Motivação

O processamento paralelo surge com o intuito de melhorar o desempenho de aplicações que necessitam de respostas rápidas, como Meteorologia (previsão do tempo), Física (simulações com alta carga computacional), Bioinformática (*data-mining* de cadeia de proteínas), Computação Gráfica (representação de volumes tridimensionais), entre outros. Para isto, a tecnologia permitiu a criação de *cluster* de máquinas multiprocessadas (arquitetura híbrida) para prover maior poder computacional, trazendo consigo a necessidade de uma programação específica para este tipo de arquitetura: a programação híbrida.

A motivação para este trabalho vem de análises sobre a programação para arquiteturas híbridas. Para utilizar estes ambientes de maneira eficiente, é necessário que o desenvolvedor possua um bom conhecimento de programação paralela em geral, e também sobre aspectos inerentes às bibliotecas que proveem comunicação entre as unidades de processamento nos diferentes níveis da arquitetura. O programador precisa saber, entre as máquinas de um *cluster*, como e quando um processo A envia determinado dado para o processo B, por exemplo. Da mesma maneira, ele precisa saber como sincronizar esta comunicação, para que o processo B não fique aguardando indefinidamente que A envie um dado, e vice-versa. Dentro de cada máquina do *cluster*, é necessário o conhecimento específico de como o processador  $m$  vai comunicar-se com o processador  $n$ ; como a memória pode ser melhor alocada; como garantir que dados serão acessados unicamente por um processo ao mesmo tempo; e diversos outros aspectos de baixo nível essenciais para que a aplicação seja correta e eficientemente paralelizada.

A etapa de desenvolvimento de aplicações híbridas de alto desempenho passa por uma série de obstáculos, conforme supracitado. Assim sendo, este trabalho é fortemente motivado pela necessidade de prover ao desenvolvedor maior facilidade e objetividade no momento de criar suas aplicações neste escopo. O desenvolvedor não necessita saber como a comunicação é realizada; qual(is) linha(s) de código deve(m) ser escrita(s) para enviar uma mensagem do processo A para o processo B; qual a ordem de execução correta para que a aplicação não entre em *deadlock*; como prover a nível de código a sincronização de regiões críticas; como escrever em seu programa que se deseja uma alocação de memória mais eficiente, quando se está trabalhando com arquiteturas NUMA.

## 1.2 Justificativa

De fato, programadores podem aprender as funções providas pelas bibliotecas de comunicação e a maneira de utilizá-las. Este conhecimento, no entanto, leva um tempo considerável para ser obtido e plenamente utilizável, o que torna o processo de desenvol-

vimento das aplicações mais demorado e tedioso. Assim, a utilização do conhecimento de especialistas na área de programação paralela para o desenvolvimento de um processo para programação híbrida de alto desempenho é um grande passo, do qual diversos aprendizados podem ser extraídos.

Outro aspecto importante a destacar neste momento é a mudança no cenário de desenvolvimento. A utilização e o desenvolvimento de programas paralelos vem crescendo em diversas áreas, como engenharias, biologia, matemática etc. Neste sentido, facilitar o desenvolvimento de aplicações paralelas através de uma interface mais intuitiva e que abstraia conceitos mais complexos da programação (como criação de processos e comunicação) pode aumentar o interesse na área de computação de alto desempenho.

Além disto, de acordo com o que foi descrito anteriormente, o desenvolvimento paralelo é naturalmente mais custoso do que o desenvolvimento sequencial. Controle de *deadlocks*, sincronização dos processos, conhecimento de primitivas e funcionalidades de diferentes ferramentas de mais baixo nível e formas de divisão de tarefas são alguns exemplos de funções que podem ser abstraídas pelo programador no momento da criação de suas aplicações.

Todos estes fatos, juntamente com a criação de uma interface com o usuário de forma que as principais configurações para o modelo paralelo desejado sejam facilmente descritas, justificam o desenvolvimento do modelo proposto. No trabalho desenvolvido, foi utilizada uma interface gráfica para esta interação com o usuário. Entretanto, é possível que qualquer tipo de interface seja desenvolvida, desde que forneça as informações necessárias para a correta execução do modelo.

### 1.3 Objetivos e contribuições

O objetivo do presente trabalho é a criação de um processo que permita a geração automática de aplicações paralelas para arquiteturas híbridas com afinidade de memória (*e.g.*, *clusters* de máquinas NUMA).

O modelo desenvolvido provê paralelização automática devido ao fato de que as primitivas de mais baixo nível de bibliotecas de troca de mensagens não precisam ser de conhecimento do usuário final, bem como primitivas de alocação de memória. Neste sentido, os objetivos específicos são:

- realizar um estudo sobre programação híbrida, principalmente no que se refere a *clusters* de máquinas NUMA;
- investigar trabalhos que utilizem as propriedades de arquiteturas NUMA para melhorar o desempenho de suas soluções;

- definir um modelo para realizar o mapeamento de *threads* em locais específicos de forma transparente ao usuário final;
- definir um modelo para realizar o mapeamento de processos pesados para os nodos de um *cluster* de forma transparente ao usuário final;
- implementar uma ferramenta gráfica (protótipo) que permita ao usuário interagir com o modelo criado;
- desenvolver aplicações com diferentes características utilizando o modelo proposto;
- avaliar o desempenho das aplicações criadas com o modelo proposto.

A principal contribuição deste trabalho é que seu resultado é um modelo que permite aos desenvolvedores de aplicações paralelas abstraírem rotinas de comunicação e sincronização entre processos no momento do desenvolvimento. A utilização transparente de bibliotecas e funcionalidades específicas para arquiteturas com acesso não uniforme à memória (NUMA) também é um fator muito relevante, podendo trazer vantagens expressivas em diversas ocasiões. Além disto, o desenvolvimento de uma ferramenta gráfica permite que desenvolvedores com menos experiência na área de programação de alto desempenho criem suas aplicações utilizando o modelo criado.

O desenvolvimento de aplicações paralelas é alvo constante de estudos, e diversas alternativas já foram propostas para facilitá-lo. Uma das contribuições mais expressivas nesta área foi a criação dos modelos de programação, também chamados de *skeletons* [Col91]. Estes modelos referem-se a padrões de paralelismo que podem ser percebidos em diversas aplicações paralelas, que trazem ao desenvolvedor mais uma alternativa para a criação de seus sistemas. O processo (modelo) desenvolvido, em sua estrutura, disponibiliza alguns *skeletons* pré-definidos para o usuário, sendo outra contribuição do presente estudo.

Com isto, espera-se que a tarefa de desenvolver aplicações paralelas híbridas de alto desempenho torne-se mais fácil, ágil e objetiva. É importante ressaltar que as aplicações paralelas criadas automaticamente utilizando o modelo proposto não têm como pretensão apresentar um desempenho melhor do que aquelas implementadas sem a utilização da ferramenta (*i.e.*, não-automáticas). Para a paralelização automática e todo o controle realizado de maneira transparente, é natural que o desempenho das aplicações criadas seja afetado. Entretanto, a expectativa é de que o comportamento da aplicação paralela com a utilização do modelo seja o mesmo do que aquele apresentado pela mesma aplicação sem a utilização da ferramenta, com diferenças de tempo aceitáveis.

Outro ponto importante a destacar refere-se ao escopo de aplicações que este trabalho abrange. As aplicações alvo deste trabalho são aplicações estáticas, ou seja, aplicações que não recebem novos dados para serem processados no decorrer de sua execução. Assim sendo, o escopo do trabalho não contempla aplicações dinâmicas.

Finalmente, ainda sobre o escopo do trabalho, é importante ressaltar que escalonamento não faz parte do presente estudo. Isto significa que não existe preocupação em escolher determinadas máquinas para a realização de determinadas tarefas. Isto significa basicamente que se o cluster utilizado for heterogêneo, o modelo proposto neste trabalho funcionará corretamente, porém, não foram realizados testes para comparar se um processo automático seria melhor que um processo realizado manualmente neste caso.

## 1.4 Metodologia de Pesquisa

Com o intuito de atingir os objetivos do trabalho, esta seção apresenta a metodologia empregada na presente pesquisa. São apresentadas algumas Questões de Pesquisa (Seção 1.4.1), algumas Hipóteses (Seção 1.4.2) e, finalmente, a forma de desenvolvimento e avaliação do presente estudo (Seção 1.4.3).

### 1.4.1 Questões de pesquisa

Quatro questões de pesquisa foram elencadas para nortear o desenvolvimento do trabalho e cumprir os objetivos almejados. São elas:

**Questão de Pesquisa 1 (Q1):** É possível desenvolver um modelo que permita a geração automática de aplicações híbridas para *clusters* de máquinas NUMA?

**Questão de Pesquisa 2 (Q2):** É possível que desenvolvedores de aplicações paralelas possam utilizar este modelo de criação automática sem conhecimento sobre primitivas de comunicação e de bibliotecas de mais baixo nível?

**Questão de Pesquisa 3 (Q3):** É possível realizar esta geração automática utilizando padrões de programação paralela (*skeletons*)?

**Questão de Pesquisa 4 (Q4):** O processo de geração automática de aplicações híbridas não acarreta em uma perda significativa de desempenho, de modo que inviabilize sua utilização?

**Questão de Pesquisa 5 (Q5):** É possível utilizar um ambiente visual (gráfico) para auxiliar no desenvolvimento de aplicações paralelas, abstraindo rotinas de baixo nível das linguagens de programação de alto desempenho?

Questão de pesquisa: programação paralela com interface gráfica ficou um bom tempo esquecida. Mudança do cenário - crescimento de usuários de outras áreas.

### 1.4.2 Hipóteses

As hipóteses levantadas objetivam responder às questões de pesquisa. São elas:

**Hipótese 1 (H1):** As primitivas de comunicação (troca de mensagens) entre os diferentes nodos de um *cluster* podem ser totalmente abstraídas.

**Hipótese 2 (H2):** A aplicação de políticas de alocação eficiente de memória em máquinas NUMA pode ser realizada de maneira transparente.

**Hipótese 3 (H3):** Os recursos computacionais de um *cluster*, bem como a interação entre eles, podem ser descritos a partir de uma interface gráfica.

**Hipótese 4 (H4):** Padrões de programação paralela já conhecidos podem estar disponíveis para o usuário.

**Hipótese 5 (H5):** Se bem estruturado, um modelo automático para criação de aplicações híbridas pode gerar aplicações com um desempenho próximo ao obtido em aplicações criadas manualmente.

Mais especificamente: as hipóteses H1 e H2 visam responder à questão Q1; a hipótese H3 visa responder às questões Q2 e Q5; a hipótese H4 visa responder à questão Q3, e; a hipótese H5 visa responder à questão Q4.

### 1.4.3 Desenvolvimento e Avaliação

O desenvolvimento da pesquisa foi realizado seguindo os passos abaixo:

1. levantamento conceitual sobre arquiteturas paralelas híbridas e programação paralela em geral;
2. levantamento de trabalhos relacionados ao que estava sendo proposto, principalmente sobre utilização de arquiteturas NUMA, padrões de programação paralela e interfaces gráficas para auxílio na programação paralela;
3. criação do modelo de mapeamento automático de processos pesados (MPI [MPI17]) para as máquinas de um *cluster*, utilizando padrões de programação paralela;
4. desenvolvimento de aplicações teste para avaliar o modelo de mapeamento de processos pesados (MPI) nos nodos de um *cluster*;
5. criação do modelo de mapeamento automático de memória em aplicações para máquinas NUMA;
6. desenvolvimento de aplicações teste para o modelo de mapeamento de *threads*, utilizando diferentes políticas de alocação de memória em máquinas NUMA;

7. integração dos modelos de mapeamento de *threads* e de processos pesados, criando um modelo para mapeamento híbrido;
8. desenvolvimento de aplicações teste para modelo de mapeamento híbrido;
9. criação de uma interface gráfica com o usuário para possibilitar sua interação com o modelo criado;
10. avaliação dos resultados obtidos com os testes.

As etapas 1 e 2 são de extrema importância para as demais, pois trazem a fundamentação teórica necessária e delimitam o que já existe na literatura. Através dos demais passos elencados, tem-se que: a hipótese H1 será avaliada através da etapa 3; a etapa 5 contribui para a avaliação da hipótese H2; a hipótese H3 será avaliada pela etapa 9; as etapas 3 e 9 são complementares para a avaliação da hipótese H4, e; a hipótese H5 será avaliada com a combinação das etapas 4, 6, 7, 8 e 10.

As implementações escolhidas para as etapas de desenvolvimento de aplicações teste foram baseadas em critérios específicos de cada modelo, objetivando avaliar diferentes configurações e possibilidades existentes, e serão detalhadas na Seção 8.

## 1.5 Estrutura do volume

O restante do documento está assim estruturado:

- O Capítulo 2 apresenta um pouco da fundamentação teórica necessária para um melhor entendimento do trabalho;
- Alguns trabalhos relacionados são descritos no Capítulo 3, no qual é possível ter uma visão sobre o que foi e está sendo pesquisado na área de conhecimento do trabalho;
- Um modelo para o mapeamento de processos pesados utilizando padrões é detalhado no Capítulo 4;
- O Capítulo 5, por sua vez, descreve um processo criado para o mapeamento automático da memória;
- A união dos dois mapeamentos (processos e memória) são descritas no Capítulo 6, onde é apresentado o mapeamento híbrido;
- O protótipo desenvolvido para que o usuário possa interagir com os modelos de mapeamento é descrito no Capítulo 7;
- Experimentos e resultados obtidos são apresentados e analisados no Capítulo 8;



- Finalizando o trabalho, algumas conclusões e trabalhos futuros são descritas no Capítulo 9.



## 2. PRESSUPOSTOS CONCEITUAIS

Este capítulo apresenta a fundamentação teórica acerca de alguns pontos importantes para o entendimento do trabalho desenvolvido. Neste sentido, alguns aspectos sobre arquiteturas paralelas, ferramentas para programação paralela, arquiteturas híbridas, programação para arquiteturas NUMA e padrões de programação paralela (*skeletons*) são descritos.

### 2.1 Arquiteturas Paralelas Híbridas

Há algumas décadas, as máquinas pessoais eram as únicas responsáveis pela resolução de problemas, não importando em quanto tempo a computação seria realizada. Entretanto, a execução de inúmeras destas aplicações, principalmente aquelas que usam muitos dados e/ou muitas operações matemáticas, levava um tempo razoavelmente grande para ser finalizada. Neste contexto, não só o tempo de execução das aplicações era um fator crítico, mas também o fato de que eventuais erros nos resultados só são descobertos após a execução, que poderia levar dias, semanas ou até mesmo meses.

O processamento paralelo surge, então, com o intuito de melhorar o desempenho de tais aplicações e proporciona um custo relativamente pequeno para a aquisição de um ambiente com poder computacional elevado. Para conseguir um maior poder de processamento, máquinas pessoais são agrupadas e interligadas por uma rede de interconexão, formando uma arquitetura que é conhecida como *cluster* (ou multicomputador). Em um *cluster*, o processamento é dividido entre as várias máquinas agregadas que o compõem (chamadas de nós).

Com este tipo de arquitetura, as aplicações podem ser divididas em partes menores, que são enviadas para cada um dos nós do *cluster*. Todos os nós computam sua parte em paralelo, e os resultados de cada uma delas são reunidos no final para gerar o resultado. Para que isto seja possível, naturalmente, as máquinas devem comunicar-se umas com as outras, o que acontece através de uma rede de interconexão. Neste sentido, cada nó do *cluster* possui seu próprio espaço de endereçamento de memória, não sendo acessível aos outros nós. Por este motivo, diz-se que um *cluster* é uma arquitetura com memória distribuída. Para que os nós se comuniquem, então, rotinas de trocas de mensagens são necessárias: quando um nó deseja enviar dados para outro, ele executa uma primitiva *send*; da mesma maneira, quando um nó deseja receber dados de outro nó, ele executa uma primitiva *receive* [RR13].

O avanço tecnológico ainda proporcionou outro tipo de arquitetura, na qual a mesma máquina possui mais de um processador. Esta arquitetura é conhecida como *mul-*

*ticore* (ou multiprocessador). Nesta abordagem, o processamento das aplicações pode ser otimizado, uma vez que existe mais de um núcleo de processamento que pode ser utilizado em paralelo [RR13]. Então, as aplicações podem ser divididas em partes menores e serem executadas na mesma máquina, uma por cada processador disponível. Diferentemente dos *clusters*, esta arquitetura dotada de vários cores possui o mesmo espaço de endereçamento. Com isto, a comunicação entre os processadores pode ser realizada através de escritas e leituras na memória.

Ambas arquiteturas (multicomputadores e multiprocessadores) conseguem diminuir o tempo de processamento das aplicações devido ao fato de proporcionarem processamento em paralelo, cada uma de sua maneira.

Com o passar dos anos, devido ao grande crescimento da arquitetura *multicore*, inevitavelmente surgiram os *clusters* nos quais os nós são máquinas multiprocessadas. A utilização das duas abordagens em conjunto emerge como uma alternativa interessante para o Processamento de Alto Desempenho, pois a quantidade de unidades de processamento disponíveis aumenta significativamente: um *cluster* com  $N$  nós monoprocessados possui  $N$  unidades de processamento; um *cluster* com  $N$  nós multiprocessados, cada um deles com  $M$  processadores, possui  $N \times M$  unidades de processamento. A Figura 2.1 ilustra de maneira simplificada uma arquitetura deste tipo, conhecida como híbrida, com cada nó de um *cluster* possuindo quatro processadores.

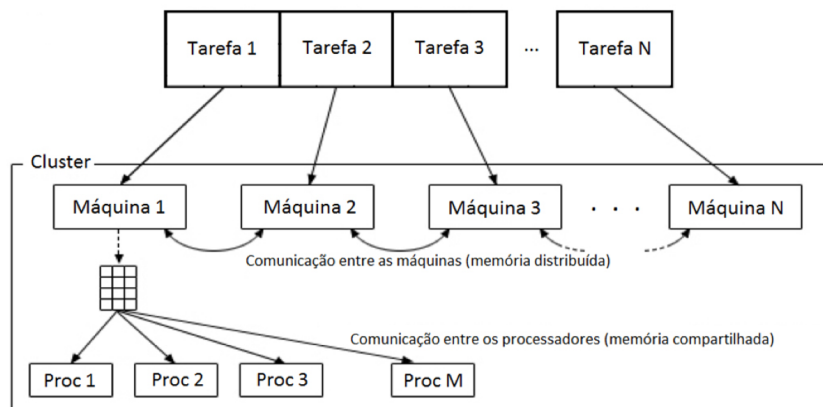


Figura 2.1 – Divisão de tarefas em uma arquitetura híbrida.

Quando uma aplicação é desenvolvida para uma arquitetura híbrida projetada para alto desempenho, existe uma hierarquia de comunicação, conforme pode ser visualizado na Figura 2.1. Geralmente, as tarefas (partes da aplicação) são divididas em grandes blocos, e enviadas para cada uma das  $N$  máquinas do *cluster*.

Dentro de cada máquina, então, a tarefa é dividida novamente, e cada uma destas novas partes é enviada para um dos  $M$  processadores existentes. Neste cenário, o paradigma de comunicação também é híbrido: a comunicação entre as diversas máquinas do *cluster* é realizada através de bibliotecas com rotinas específicas para memória distribuída,

enquanto a comunicação interna em cada máquina é realizada através de bibliotecas com rotinas específicas para memória compartilhada.

Esta nova abordagem híbrida tornou-se uma forte tendência na construção de máquinas paralelas, pois traz vantagens dos dois paradigmas. Entretanto, o advento deste novo paradigma incorpora novos desafios à computação paralela. Alguns destes são:

- maior complexidade na implementação, pois não basta somente transformar a aplicação sequencial em paralela, sendo necessário o conhecimento dos pontos passíveis de paralelismo, e qual tipo de paralelismo será realizado em cada parte;
- dependência entre a implementação e o tipo de máquina para a qual se está desenvolvendo, pois se a implementação for realizada de maneira inadequada, a aplicação paralela será altamente dependente da arquitetura híbrida escolhida, não sendo facilmente portada para outras máquinas;
- prover balanceamento de carga de forma adequada entre os recursos disponíveis, procurando não subutilizar, nem sobrecarregar um recurso;

Assim sendo, torna-se extremamente importante o domínio deste tipo de programação, para que haja um real aproveitamento de todo o poder de processamento disponibilizado pela arquitetura.

## 2.2 Arquiteturas NUMA

Conforme dito anteriormente, *clusters* possuem cada módulo de memória localizado próximo de um processador, enquadrando-se na classificação de memória distribuída. A comunicação, então, deve ser feita através da troca de mensagens. Por outro lado, quando se refere à memória compartilhada, existe apenas um único espaço de endereçamento para todos os processadores. A comunicação entre os eles, portanto, pode ser feita diretamente com operações de escrita e leitura da memória.

Apesar da existência de um espaço de endereçamento global (único), dependendo do tipo de acesso à memória, multiprocessadores podem ser classificados em [DRN03, De 06]: UMA, NUMA e COMA.

Arquiteturas UMA (*Uniform Memory Access*) possuem uma memória centralizada e normalmente implementada em um único bloco. O acesso a tal memória ocorre de maneira uniforme por parte de todos os processadores. O uso de barramento é frequente neste tipo de arquitetura, restringindo o número de transações a uma única por vez.

No tipo de arquitetura NUMA (*Non-Uniform Memory Access*) existem pares processador/memória conectados entre si através de um barramento, portanto cada processador

terá a sua memória local. Todos os pares comunicam-se uns com os outros através de uma rede de interconexão. Apesar de uma memória aparentemente distribuída, neste tipo de classificação o espaço de endereçamento é único (constituído por todas as memórias dos pares). Quando um processador acessa sua memória local o tempo de acesso é reduzido, mas quando um processador requisita um endereço que se encontra em uma memória remota (não local) o tempo de acesso aumenta, pois a transação deverá ser realizada através da rede de interconexão e não diretamente pelo barramento.

As arquiteturas COMA (*Cache-Only Memory Architecture*) assemelham-se às arquiteturas NUMA, porém possuem memórias cache conectadas aos processadores. Estas memórias possuem uma capacidade de armazenamento relativamente maior do que uma memória cache normal.

Este trabalho foca em arquiteturas híbridas que possuem máquinas do tipo NUMA. Neste cenário, alguns detalhes são importantes para que um melhor aproveitamento da arquitetura possa ser realizado.

O entendimento de que o acesso à memória próxima a outro processador (chamado de acesso remoto) é mais custoso do que o acesso a memória que está mais próxima de si (acesso local) é importante no contexto NUMA. Assim, existem algumas maneiras que permitem aos programadores definirem em quais locais da memória os dados devem ser armazenados.

Comumente, a política *first-touch* é aplicada em arquiteturas NUMA [RMC+09]. Esta política aloca a memória no nodo NUMA que realizou o primeiro acesso ao dado. Entretanto, nem sempre esta localização dos dados é útil para a aplicação, podendo causar acessos remotos indesejados.

Neste sentido, escolher uma política de afinidade de memória pode ser uma alternativa interessante no desenvolvimento de aplicações para este tipo de arquitetura. Na Seção 2.3.2, é descrita a biblioteca *Memory Affinity Interface* (MAI) [RM10], que apresenta algumas alternativas para a alocação mais eficiente de memória em arquiteturas NUMA.

## 2.3 Programação Paralela de Alto Desempenho

Esta seção apresenta algumas bibliotecas e ferramentas úteis no desenvolvimento de aplicações paralelas. Além disto, alguns pontos sobre programação para ambientes híbridos são descritos.

### 2.3.1 Bibliotecas

Para a implementação de aplicações paralelas, algumas ferramentas apresentam rotinas e primitivas de comunicação entre os processos, de acordo com a hierarquia em que estes se encontram.

A ferramenta mais utilizada para prover comunicação através de trocas de mensagens é o MPI (*Message Passing Interface*) [MPI17, SOHL<sup>+</sup>98], enquanto para a comunicação através de operações em memória as mais conhecidas e utilizadas são *Threads* [KSS96, LB95] e *OpenMP (Open Multi Processing)* [CJP07, DM98].

#### MPI

O MPI é um padrão de troca de mensagens que permite a comunicação entre os processos de uma aplicação paralela. Ele foi criado em 1993 por várias empresas que estão inseridas no mercado de computação (IBM, Cray, Intel etc.), universidades e laboratórios de pesquisa. Este padrão foi definido para suprir a necessidade de uma biblioteca portátil que funcionasse de maneira eficiente, aproveitando ao máximo as vantagens providas pelas diferentes arquiteturas disponíveis por diferentes fabricantes.

O padrão MPI foi desenvolvido baseando-se nas diferentes bibliotecas existentes, como PVM (*Parallel Virtual Machine*) [PVM17], por exemplo, passando a ter um papel importante no contexto de computação paralela. Com ele, os fabricantes de máquinas paralelas puderam implementar um conjunto bem definido de rotinas que suprissem a necessidade de diferentes tipos e modelos de máquinas. Este padrão permite utilizar o modelo SPMD (*Single Processor Multiple Data*), sendo que todos os processos executam o mesmo código, e é o programador que deve escolher explicitamente quais processos devem executar quais trechos de código. Pode-se citar algumas vantagens do MPI, tais como eficiência, portabilidade, transparência, segurança e escalabilidade.

Um programa MPI consiste em processos autônomos, que executam o mesmo código. A comunicação é feita através de duas operações básicas: *send* e *receive*. É um método que introduz conceitos como o *rank*, que se trata de uma identificação única para cada processo, sempre de forma crescente. Assim, cada processador pode executar a parte do código que lhe for designada. Além disso, o MPI oferece diversas formas de implementar a comunicação ponto-a-ponto (de forma síncrona ou assíncrona, bloqueante ou não bloqueante) e a comunicação coletiva, como mensagens em *broadcast* e barreiras. Algumas primitivas básicas do MPI são listadas abaixo:

- `MPI_Status`: uma estrutura que armazena informações sobre a comunicação realizada, sendo utilizado (principalmente) nas primitivas de recebimento;

- `MPI_Init`: inicializa o ambiente MPI, recebendo como parâmetros o `argc` e o `argv` do programa. Com isto, o MPI cria os processos necessários de acordo com o que foi informado para aplicação;
- `MPI_Comm_size`: indica quantos processos fazem parte da comunicação;
- `MPI_Comm_rank`: identifica os processos unicamente, possibilitando o envio de mensagens diretamente entre as máquinas;
- `MPI_Send`: primitiva de envio de dados, que coloca na rede as informações que necessitam ser comunicadas;
- `MPI_Recv`: primitiva de recebimento de dados, que retira da rede as informações que foram enviadas por algum processo;
- `MPI_Finalize`: finaliza o ambiente MPI, terminando com os processos alocados anteriormente.

A utilização do MPI não se trata de uma tarefa trivial. A manipulação das primitivas para que a comunicação seja feita de forma correta não é simples, e o presente trabalho visa facilitar o processo de desenvolvimento do usuário que deseja utilizar estas primitivas de maneira mais ágil e transparente.

Atualmente são encontradas implementações de bibliotecas MPI em C, C++, Fortran, Java, dentre outras. Uma ferramenta em particular que é muito utilizada com as implementações MPI é a MPE (*MPI Parallel Environment*) [CGL98]. Esta disponibiliza ao usuário a possibilidade de monitorar seu ambiente, fazer animações mostrando a sobrecarga de comunicação, além de outras coisas.

### *Threads*

Um processador executa diversas tarefas simultaneamente, isto é, vários processos que compartilham a CPU, apoderando-se de determinadas fatias de tempo para a sua execução.

Neste contexto, um processo "é uma instância única de uma aplicação que está sendo executada"[MSD17], ou seja, é o executável de um programa carregado em memória (aplicação). *Threads* podem ser consideradas como subprocessos. São fluxos de execução que rodam em uma aplicação. Um programa sem *threads* possui um único fluxo de execução, realizando uma tarefa por vez. Em um programa com a utilização de *threads*, por sua vez, tem-se a execução de mais de uma tarefa simultaneamente [KSS96].

Pode-se, então, dividir a execução de aplicações em múltiplos caminhos (*threads*) que rodam concorrentemente na memória compartilhada e compartilham os mesmos recursos do processo pai. Pensando em uma máquina multiprocessada, é possível que se



obtenha um ganho de desempenho dividindo a aplicação em *threads*, que serão responsáveis por executar certa tarefa. Desta maneira, cada uma delas pode ser alocada em um processador da máquina acessando a memória compartilhada para a obtenção de informações dos outros processos. O uso de *threads* é de baixo custo para o sistema operacional e de fácil criação, manutenção e gerenciamento.

A biblioteca de *threads* mais utilizada hoje em dia é a *POSIX Threads* ou *Pthreads*. Esta biblioteca foi desenvolvida baseando-se no padrão POSIX IEEE, que é uma interface de programação padrão para as *threads*. Ela pode ser encontrada nos sistemas UNIX e oferece diversas facilidades ao programador, além de ser gratuita [Bar].

## OpenMP

OpenMP agrupa um conjunto de especificações e padrões desenvolvidos para diretivas de compiladores, rotinas de biblioteca e variáveis de ambiente, com a finalidade de ser utilizado para a construção de um programa paralelo destinado a rodar em uma arquitetura com memória compartilhada. O OpenMP foi desenvolvido por uma série de empresas de software e hardware, universidades e grupos governamentais pelos meados de 1997. Foi definido a fim de criar padrões quanto ao desenvolvimento de interfaces de programação para máquinas paralelas com memórias compartilhadas tornando, assim, portáveis tais aplicações.

Este padrão foi desenvolvido baseado em alguns objetivos, como a portabilidade, a facilidade de uso e a definição de certas diretivas para o uso do programa paralelo. Com o primeiro, era garantido que as aplicações desenvolvidas poderiam ser executadas em máquinas de diferentes fabricantes, as quais possuíam algumas características diferentes uma das outras. O segundo, era um fator que atrairia o programador a vir a utilizar uma implementação OpenMP, divulgando cada vez mais o padrão, além de oferecer facilidades como a possibilidade de, incrementalmente, paralelizar uma aplicação. O último estabelecia um conjunto simples de diretivas para a programação em uma máquina paralela com memória compartilhada. Com estes objetivos em mente o modo de como o paralelismo OpenMP funcionaria pode ser definido.

OpenMP é um paralelismo baseado em *threads*, levando em consideração que um processo pode ser dividido em múltiplos fluxos de instruções para que o paralelismo se concretize e suporta a dinamicidade na mudança de número de *threads* durante a execução do programa. Este padrão provê um modelo *fork/join*, com a existência de uma *thread* mestre que pode criar várias *threads* para a realização de uma tarefa em paralelo durante a execução de um trecho do programa e, ao término do trabalho, as *threads* podem ser destruídas, voltando a existir somente a *thread* mestre. Este processo pode ser repetido quantas vezes forem necessárias. Outra característica do padrão é o paralelismo explícito, ou seja, o programador deve definir quais as áreas do programa deverão ser executadas

em paralelo. Deve-se ressaltar que o OpenMP permite um paralelismo de dados, mas não suporta o paralelismo de tarefas.

Existem diversos compiladores para OpenMP, e algumas linguagens suportadas são Fortran, C e C++. Compiladores para programas Java que utilizem diretivas OpenMP também são encontrados.

### 2.3.2 Programação Híbrida

Conforme descrito anteriormente, a programação em ambientes híbridos (como é o caso de *clusters* de máquinas NUMA) insere novos desafios. Na programação paralela existe uma busca constante por maior desempenho das aplicações desenvolvidas. Assim, com arquiteturas mais complexas, surgem novas ferramentas que visam proporcionar aos programadores funcionalidades importantes. É o caso da biblioteca MAI, descrita a seguir.

#### MAI - *Memory Affinity Interface*

A interface MAI [RM10] proporciona funcionalidades que permitem a alocação da memória em máquinas NUMA de maneira mais inteligente, de acordo com a definição de algumas políticas bem definidas. Esta interface é desenvolvida sobre a NUMA API [Cas09], retirando a necessidade de conhecimento de chamadas de mais baixo nível. A NUMA API possui algumas bibliotecas principais, como a `libnuma` e a `numactl`, e permitem uma alocação mais correta dos dados. Entretanto, a utilização da NUMA API não é de conhecimento da grande maioria dos programadores, sendo a interface MAI uma alternativa para utilizar as funcionalidades da NUMA API de forma mais simplificada.

Desenvolvida no LIG (*Laboratoire d'Informatique de Grenoble*) [LIG17], a interface objetiva a alocação eficiente de estruturas, como *arrays* unidimensionais e bidimensionais. São previstas algumas políticas de alocação de memória, e dentre elas podem ser destacadas:

**cyclic**: alocação de páginas de memória é feita em círculos;

**cyclic\_block**: alocação de blocos de memória é feita em círculos;

**bind\_all**: aloca a memória nos nodos indicados no arquivo de configuração. Entretanto, é o Sistema Operacional que escolhe em qual nodo alocar;

**bind\_block**: aloca blocos de memória mais próximo aos nodos definidos no arquivo de configuração.

Neste sentido, com a utilização da MAI, espera-se obter uma melhor localidade dos dados, melhorando o desempenho geral das aplicações. Neste sentido, algumas das funções utilizadas da interface são:

**init:** inicializa o ambiente NUMA, definindo quantidade de *threads* e nodos NUMA, através do arquivo de configuração;

**final:** finaliza o ambiente NUMA;

**get\_num\_nodes:** retorna o número de nodos NUMA definidos no arquivo de configuração;

**get\_num\_threads:** retorna a quantidade de *threads* informada no arquivo de configuração

**alloc\_1D:** reserva memória virtual para um *array* unidimensional, fazendo com que as páginas correspondentes sejam *untouched*, garantindo que as páginas só serão alocadas quando forem escritas;

**alloc\_2D:** reserva memória virtual para um *array* bidimensional, fazendo com que as páginas correspondentes sejam *untouched*, garantindo que as páginas só serão alocadas quando forem escritas;

**cyclic, cyclic\_block, bind\_all e bind\_block:** aplica a alocação correspondente;

**set\_thread\_id\_omp:** vincula uma *thread* a um processador, indicando que esta não deve ser migrada;

**bind\_threads:** aplica **set\_thread\_id\_omp**.

Este conjunto de funções e chamadas da biblioteca MAI foi utilizado no desenvolvimento do trabalho, porém, de maneira transparente para o usuário.

## 2.4 Padrões de Programação Paralela (*Skeletons*)

Padrões de programação paralela são utilizados para uma melhor organização do código fonte, alcançando assim um melhor resultado. Com tais padrões diferentes grupos de programas paralelos podem ser classificados, facilitando o estudo e o entendimento da abordagem escolhida.

Murray Cole é um dos principais autores e pesquisadores sobre padrões, que também são conhecidos como *skeletons*. Em [Col04], Cole apresenta um manifesto sobre a programação paralela com a utilização e detalhamento de princípios já definidos sobre *skeletons*, com a finalidade de mostrar a forma correta de utilização e princípios a serem seguidos.

Padrões de programação podem ser vistos em diferentes aplicações, e trazem consigo a ideia de reutilização, uma vez que podem ser parametrizados de diversas formas, possibilitando a criação de diferentes aplicações. Assim, é possível definir um formato específico no qual diversas aplicações vão se encaixar, facilitando o entendimento e a programação de aplicações paralelas.

Existem diversos padrões de programação paralela e definições de diferentes autores sobre o assunto. Alguns dos principais *skeletons* encontrados na literatura são:

**Map:** ocorre quando um elemento pode ser dividido em vários, e uma operação pode é realizada sobre cada uma das partes. Ao final, as partes são reunidas em um resultado final;

**Farm:** também conhecido como mestre/escravo, no qual um processo divide diferentes tarefas para os demais processos. Também é conhecido como saco de tarefas;

**Pipe:** um dos mais conhecidos padrões da literatura, onde existe a ideia da divisão do processamento em estágios. Atinge-se o paralelismo quando os diferentes estágios (mapeados entre os processos) estão todos ocorrendo em paralelo;

**Divide & Conquer (D&C):** neste padrão, os dados são divididos até que uma determinada condição de parada seja alcançada. Enquanto a condição não for alcançada, os dados continuam sendo divididos, e o processo é recursivamente realizado, o que gera uma espécie de árvore. Quando a condição de parada é atingida, realiza-se a fase da conquista, onde os dados são processados e enviados ao processo “pai” na árvore, até que chegue na raiz, que finaliza a computação.

Neste contexto, *skeletons* são bastante úteis para a criação de um modelo de geração automática de código paralelo. Assim sendo, o presente trabalho utiliza a ideia de padrões de programação paralela, aproveitando-se das facilidades e do controle providos por eles.

### 3. TRABALHOS RELACIONADOS

Este capítulo apresenta trabalhos de diferentes autores que, de alguma maneira, estão relacionados com o assunto da presente pesquisa. Com a finalidade de facilitar a leitura e a organização do texto, os trabalhos elencados foram divididos em 3 seções:

- **NUMA**: apresenta alguns trabalhos interessantes relacionados à afinidade de memória em arquiteturas do tipo NUMA através de diferentes ferramentas e abordagens;
- **Skeletons**: mostra a importância e o empenho de diferentes autores para a utilização de modelos de programação paralela;
- **Programação Paralela x Interface Gráfica**: a seção traz trabalhos que utilizam uma GUI (*Graphical User Interface*) para auxiliar a programação paralela.

Ao final do capítulo, é realizada uma análise crítico-comparativa dos referidos trabalhos em relação ao proposto neste estudo. O principal objetivo deste capítulo é trazer ao leitor uma boa visão do foi e do que está sendo pesquisado no contexto desta tese, situando o presente trabalho dentre os demais.

#### 3.1 NUMA

No trabalho apresentado em [RMM<sup>+</sup>08], os autores avaliam diferentes estratégias para explorar a afinidade de memória em máquinas NUMA. Além de apresentarem os principais conceitos de arquiteturas NUMA, os autores enfatizam a importância da afinidade de memória nestes ambientes. Duas aplicações serviram de estudo de caso: *benchmark NAS* [Lin05] e *Ondes 3D* [DAD<sup>+</sup>08]. Para melhorar o desempenho das aplicações implementadas, chamadas de sistema explícitas da API NUMA foram incluídas no código das aplicações, sendo elas: *mbind* e *sched\_setaffinity* [RMM<sup>+</sup>08]. Realizando testes com 4 estratégias de alocação de memória diferentes (*First-Touch*, *Parallel-Init*, *Round-Robin* e *Memory-Bind*), os autores puderam observar ganhos de até 80% com as estratégias de afinidade definidas.

Além de sistemas apenas para máquinas NUMA, muitos autores estão encaminhando suas pesquisas para clusters de máquinas NUMA. Um exemplo é o trabalho descrito em [PJN08]. Nele, é apresentado o ambiente MPC (*MultiProcessor Communications*), que objetiva prover aos programadores uma API que use aspectos de programação para arquiteturas de memória distribuída e compartilhada. De acordo com os autores, existem diversas abordagens utilizadas para programar para clusters de máquinas multiprocessadas a fim de explorar eficientemente a arquitetura, sendo as listadas [PJN08]:

- **Implementações MPI Avançadas:** mesmo com implementações otimizadas para arquiteturas específicas, o que aumenta o desempenho da aplicação, estas implementações não exploram o que as máquinas NUMA podem oferecer;
- **Implementações OpenMP para Clusters:** embora algumas implementações como Intel Cluster OpenMP [Hoe06] e OpenMPD [LSB07] permitam que o OpenMP opere em ambientes de memória distribuída, a utilização deste tipo de abordagem não oferece a eficiência esperada, pois a utilização de DSM (*Distributed Shared Memory*) requer mecanismos de consistência que levam a uma perda significativa de desempenho;
- **Implementações híbridas MPI+OpenMP:** os autores indicam que, apesar de ser uma abordagem aparentemente promissora, alguns problemas podem ocorrer quando da utilização desta combinação, devido ao fato de que nem sempre as implementações com MPI são totalmente *thread-safe* e pelas técnicas de “*busy waiting*” utilizadas na detecção de eventos;
- **Virtualização de Processos:** esta solução é apontada como eficiente, por separar tarefas e processos, mapeando-as para threads. Implementações como AMPI [HLK03] e TOMPI [Dem97] utilizam esta abordagem, assim como a solução apresentada pelos autores (MPC).

O MPC possui os mecanismos scheduler e allocator, que controlam localidade dos dados, migração de *threads* e outros fatores importantes quando se utiliza um *cluster* de máquinas NUMA. Ainda, segundo os autores, a alteração de aplicações MPI e/ou com a utilização de *threads* (POSIX) já existentes para o ambiente MPC é simples [PJN08].

A interface MAI, que lida com políticas de alocação de memória em máquinas NUMA, é o foco do trabalho descrito em [RMC<sup>+</sup>09]. Neste trabalho, a interface MAI (*Memory Affinity Interface*) é apresentada, juntamente com suas políticas de alocação de memória. Os resultados para 3 aplicações mostram claramente os ganhos da alocação otimizada de memória nos ambientes testados, mostrando uma melhora de cerca de 31% em relação à política *first-touch* (padrão do Linux).

No mesmo contexto, os autores de [CFR<sup>+</sup>09] apresentaram detalhes da aplicação ICTM (*Interval Categorizer Tessellation Model*) implementada com a biblioteca MAI. O ICTM é uma aplicação Geofísica, que categoriza regiões geográficas de acordo com suas características, obtidas através de imagens de satélite. Os autores desenvolveram duas versões: uma apenas com OpenMP e outra com a utilização da MAI. A versão OpenMP da aplicação já apresentava um desempenho bastante superior à aplicação sequencial. No entanto, os resultados da implementação com aplicação de políticas de afinidade de memória (através da MAI) apresentaram um desempenho ainda maior. As políticas aplicadas, neste caso, foram *bind\_all*, *bind\_block*, *cyclic* e *cyclic\_block*. Esta versão da aplicação foi denominada NUMA-ICTM.

O trabalho [RGR<sup>+</sup>11] dá continuidade à implementação supracitada, criando uma nova versão do NUMA-ICTM [CFR<sup>+</sup>09], desta vez para *clusters* de máquinas NUMA. A aplicação ICTM foi implementada utilizando um paradigma híbrido de paralelismo (MPI+OpenMP) juntamente com as políticas de alocação eficiente de memória (MAI). Os resultados mostram uma melhora considerável no desempenho, chegando a um *speedup* perto de 40 com a utilização de 50 núcleos de processamento.

## 3.2 Skeletons

Diversos *frameworks* baseados em *skeletons* (ASkF - *Algorithmic Skeleton Frameworks*) são encontrados na literatura, cada um com suas características, implementações e padrões de programação implementados. Em [Col04] é descrita a primeira versão de uma biblioteca chamada eSkel (*edinburgh Skeleton library*), implementada sobre C e MPI. A motivação para o desenvolvimento da eSkel foi, principalmente, o manifesto de Colen (também descrito em [Col04]). Após descrever os detalhes da implementação da biblioteca, o autor retorna aos princípios fundamentais da criação de sistemas que utilizam *skeletons*. Em [BCGH05] os autores apresentam uma versão mais atualizada da biblioteca.

Por ser implementado sobre MPI, operações do eSkel devem ser executadas somente depois da inicialização do ambiente paralelo. *Farm*, *Pipeline* e *Divide & Conquer* são os principais *skeletons* presentes na biblioteca. Alguns deles (como *Farm* e *Pipeline*) possuem “famílias”, que se tratam do mesmo modelo de programação, porém com algum(s) parâmetro(s) ou tipo de interação diferente(s) [BCGH05].

Outra biblioteca que disponibiliza a utilização de *skeletons* é descrita em [FSCL06]. Os autores citam que a utilização de bibliotecas como MPI e PVM não são de fácil manipulação, motivando a utilização de modelos de programação paralela já conhecidos. Os autores apontam *readability* e *efficiency* como desafios a serem atingidos. *Readability* refere-se à transparência do paralelismo, enquanto *efficiency* refere-se ao desempenho da aplicação gerada, que deve estar de acordo com a implementação utilizando apenas as bibliotecas de mais baixo nível. Quaff apresenta quatro *skeletons* disponíveis: *scm*, *pipeline*, *farm* e *pardo*<sup>1</sup>, além de permitir que padrões sejam aninhados. A maior diferença apontada pelos autores é que Quaff utiliza a linguagem C++ e, objetivando diminuir o *overhead* da abstração, utiliza técnicas de meta-programação por *templates* (*template meta-programming*).

---

<sup>1</sup>Para maiores detalhes sobre os *skeletons* que não foram descritos neste trabalho, sugere-se a leitura de [FSCL06]

### 3.3 Programação Paralela x Interface Gráfica

Esta seção objetiva realizar um levantamento das ferramentas relacionadas mais conhecidas no meio acadêmico. Além dos trabalhos descritos, diversos outros ainda poderiam ser elencados. Entretanto, acredita-se que os que foram escolhidos contemplam o que há na literatura sobre o assunto.

Embora o foco do trabalho apresentado seja os modelos de mapeamento para a geração automática de código paralelo, a pesquisa sobre aplicações gráficas trouxe à tona diversas possibilidades e situou o desenvolvimento do modelo proposto, bem como o do protótipo.

Em [BDG<sup>+</sup>91], os autores apresentam o ambiente de programação HeNCE (*Heterogeneous Network Computing Environment*). Desenvolvido sobre PVM, HeNCE possui uma interface gráfica com o usuário que é utilizada para compilar, executar, debugar, dentre outras tarefas intrínsecas ao desenvolvimento paralelo. O usuário pode informar à ferramenta em qual(is) máquina(s) seu programa será executado. Assim, de acordo com algumas características informadas e uma matriz de custos (que também pode ser definida), HeNCE realiza um procedimento de escalonamento entre as máquinas.

O usuário deve desenhar um grafo representando as dependências e o fluxo da sua aplicação, desenhando os nodos e interligando-os com arcos. Este grafo deve ser inserido na interface gráfica, ilustrada na Figura 3.1. Em cada nodo informado, deve ser inserida a informação do método a ser executado e dos seus parâmetros de entrada. Os nomes dos parâmetros são de fundamental importância, pois é desta forma que o algoritmo do HeNCE saberá qual valor deve ser recebido como entrada em cada método. Para a execução do programa, HeNCE necessita do grafo construído (e anotado) e do código dos métodos a serem chamados. HeNCE, então, configura a máquina virtual de acordo com o informado pelo usuário e executa o programa.

CODE 2.0 é apresentado em [NB92]. Trata-se da terceira versão da ferramenta, precedida pelas versões CODE 1.0 e CODE 1.2. Os autores relatam que CODE 2.0 é de fácil utilização, sendo um dos motivos o fato de que o usuário desenha um grafo para representar os nodos e suas comunicações. Além disto, não é necessário conhecer primitivas específicas de comunicação. A Figura 3.2 ilustra um exemplo de utilização do CODE 2.0.

Os nodos (ou *Unit of Computational nodes* - UCs [NB92]) representam computações sequenciais (em linguagem C, por exemplo), nas quais os dados de entrada são computados e o resultado é a saída do nodo. Na Figura 3.2 está sendo realizada uma implementação para a Equação de Laplace, computando valores de um array bidimensional. Dois grafos são apresentados: main (Figura 3.2a) e Laplace (Figura 3.2b). Resumidamente, ReadInputs é responsável por obter as informações necessárias para informar ao grafo Laplace (S, Goal e NProcs). No grafo Laplace, o valor Goal serve como entrada para InitMat,



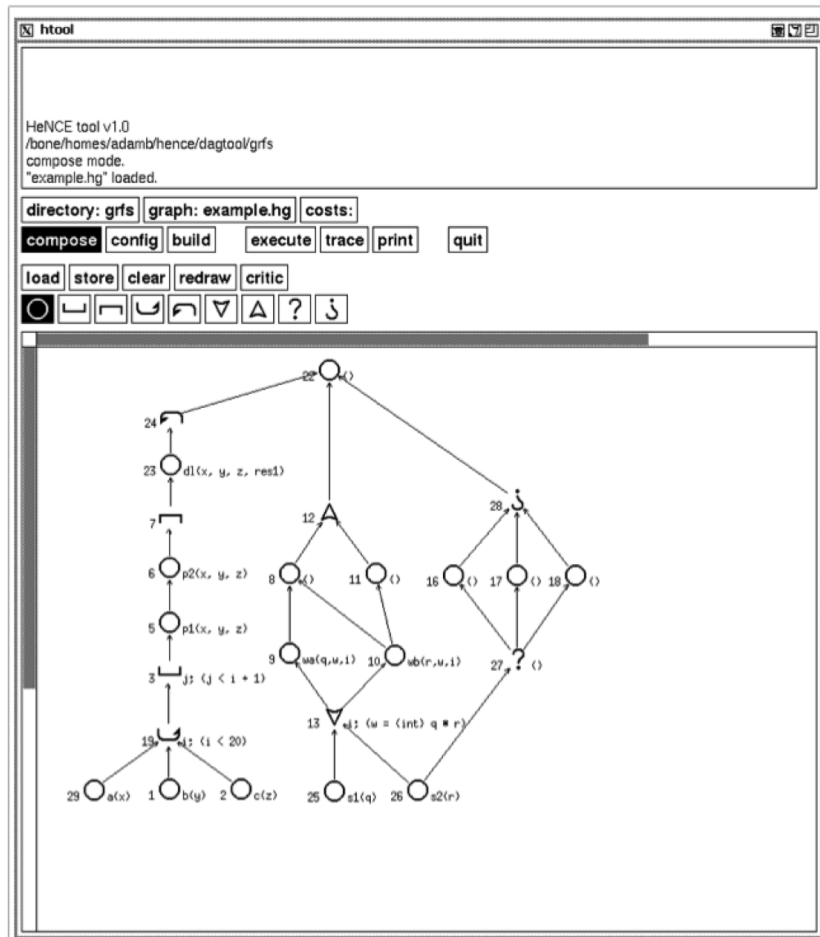


Figura 3.1 – Interface Gráfica do Hence [BDG<sup>+</sup>91].

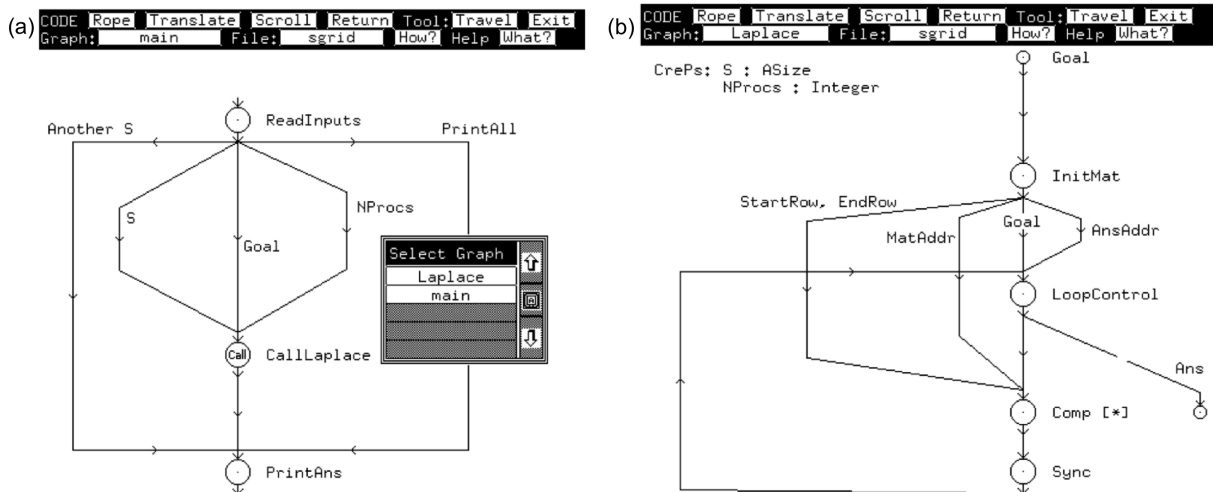


Figura 3.2 – Exemplo de utilização do CODE 2.0 [NB92].

enquanto os demais são declarados como globais para todo o grafo. O ponto mais importante trata-se da ocorrência da anotação “[ \* ]” ao lado do nodo Comp (Figura 3.2b). Trata-se de um indicativo de que este nodo será dividido em NProcs nodos. O código produzido pela ferramenta CODE 2.0 é escrito na linguagem de programação Ada.

De maneira similar aos trabalhos apresentados anteriormente, Paralex [BAA<sup>+</sup>92] utiliza a ideia de grafos para a criação do modelo paralelo. Os autores denominam os componentes do grafo de *nodos* e *links*. Os nodos realizarão a computação e os *links* (arcos no grafo) representam o fluxo dos dados. As computações sequenciais de cada nodo podem ser realizadas em linguagens como C, C++, Fortran, Pascal, Modula ou Lisp. A Figura 3.3 mostra a interface gráfica do Paralex.

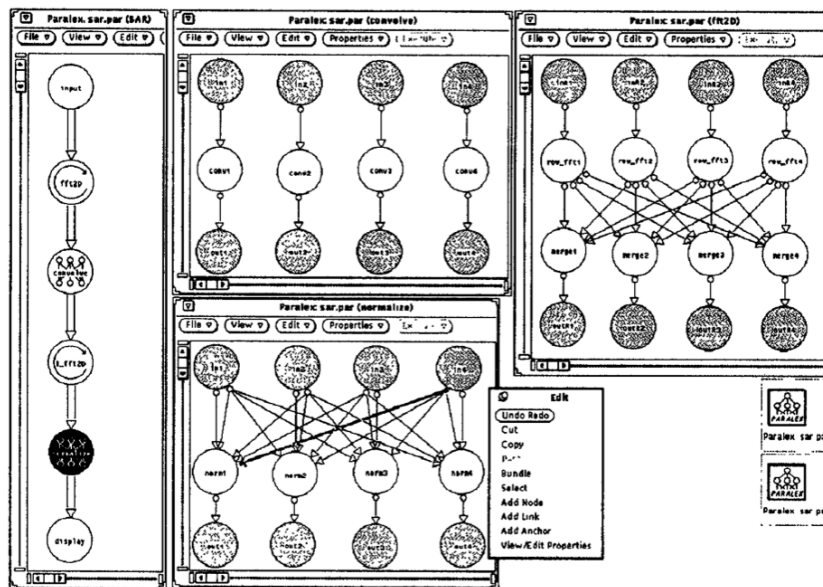


Figura 3.3 – Interface gráfica do Paralex [BAA<sup>+</sup>92].

É possível verificar uma grande semelhança com as duas outras ferramentas. Para incluir código em um nodo, deve-se escolher a opção Edit no topo da ferramenta. Segundo os autores, a ferramenta mostra que aplicações paralelas complexas podem ser desenvolvidas sem a necessidade de programações de baixo nível. A diferença principal entre HeNCE e o Paralex, apontada pelos próprios autores, é o fato de que HeNCE não possui tolerância a falhas e balanceamento dinâmico de maneira automática. Paralex, por sua vez, permite que o usuário escolha um nível de tolerância a falhas (mensagens perdidas e processos que parem de executar).

Diferentemente dos trabalhos anteriores, em [MQ93], McCallum e Quinn apresentam um ambiente para o desenvolvimento de aplicações paralelas tendo como foco a linguagem Dataparallel C [HQL<sup>+</sup>91]. A motivação para o desenvolvimento da ferramenta foi, principalmente, o desenvolvimento de aplicações mais eficientes em termos de comunicação. A ferramenta foi desenvolvida para dar um retorno imediato ao usuário sobre as comunicações inseridas no programa compilado. Utilizando Dataparallel C, o compilador gera as comunicações necessárias, de acordo com o que definido no código. Em determinadas ocasiões, uma simples alteração no código pode evitar que o compilador gere um tipo de comunicação mais custosa, o que degrada o desempenho da aplicação como um todo. Assim sendo, Intercom oferece anotações no código compilado, que informa o tipo e a

localização de cada comunicação gerada pelo compilador. Assim, o usuário pode analisar seu código e, eventualmente, alterar o que for necessário.

Outro ambiente visual para a programação para o paradigma de troca de mensagens é descrito em [ND94]. Neste trabalho, os autores apresentam VPE, que a exemplo do HeNCE, utiliza PVM como ferramenta de programação paralela. Assim como HeNCE, VPE possui as vantagens da utilização do PVM, e o ambiente paralelo desenvolvido permite compilação, execução da aplicação, animação da aplicação e configuração do ambiente. Um ponto ressaltado pelos autores é que o VPE não utiliza as primitivas do PVM diretamente, mas sim, primitivas que podem ser aplicadas tanto para PVM quanto para MPI. Na Figura 3.4 é possível observar a interface gráfica da ferramenta.

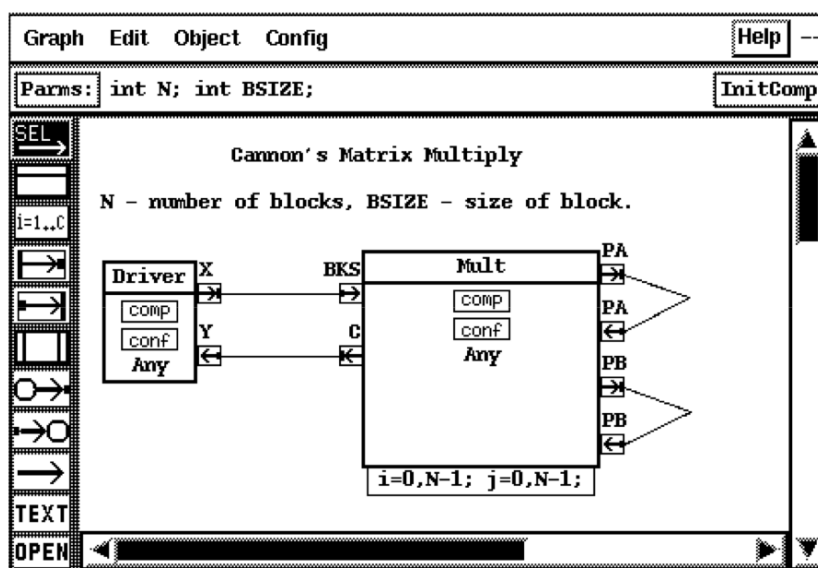


Figura 3.4 – Interface gráfica do VPE [ND94].

Percebe-se, novamente, que a modelagem dos nodos e da comunicação entre eles é realizada através de um grafo e seus arcos. Usuários informam seus códigos sequenciais em C ou Fortran.

Outra linguagem de programação visual é apresentada em [KDF96]. Chamada de GRAPNEL (GRAPhical Process's NEt Language), a linguagem também é voltada para a programação em ambientes com troca de mensagens, como PVM e MPI. Um processo na ferramenta GRAPNEL pode ser um único nodo ou um grupo de processos, o que faz com que a comunicação presente possa ser tanto ponto-a-ponto quanto em grupo. No modelo GRAPNEL, a comunicação é feita através de "portas", e estas podem ser apenas de entrada, apenas de saída ou de entrada e saída. A Figura 3.5 demonstra um exemplo da interface do GRAPNEL.

Na Figura 3.5a observa-se a definição dos processos e a comunicação entre eles. Quando o usuário acessa o processo, uma nova janela é apresentada, para que estrutura interna do processo seja descrita (como mostra a Figura 3.5b). Na estrutura interna, cada símbolo possui um significado diferente. Quando o usuário seleciona algum símbolo, uma

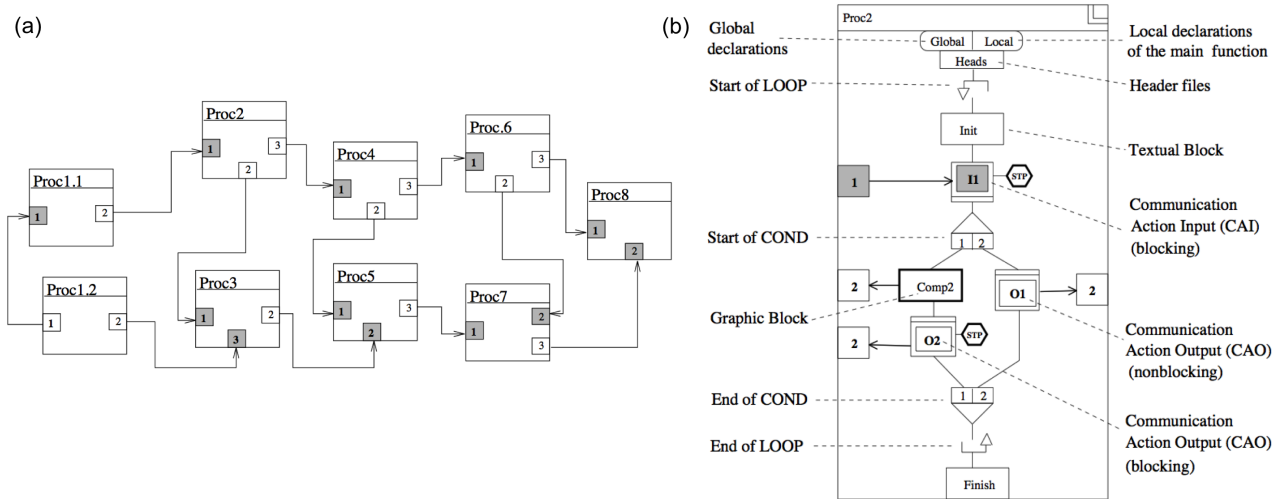


Figura 3.5 – Exemplo da interface do GRAPNEL [KDF96].

nova janela é exibida para que trechos de código em C sejam incluídos. Além disso, algumas topologias de processos e comunicações são pré-definidas na ferramenta, como *Tree*, *Array* e *Farm*, por exemplo.

Não distante dos demais trabalhos descritos, GRIX (GRICSS, *GR*aphical *I*nter-processor *C*ommunication *S*upport *S*ystem) [SMT00] apresenta uma interface visual com intuito de reduzir o trabalho dos usuários no momento de realizar a comunicação entre os nodos. A Figura 3.6 ilustra duas telas do GRIX.

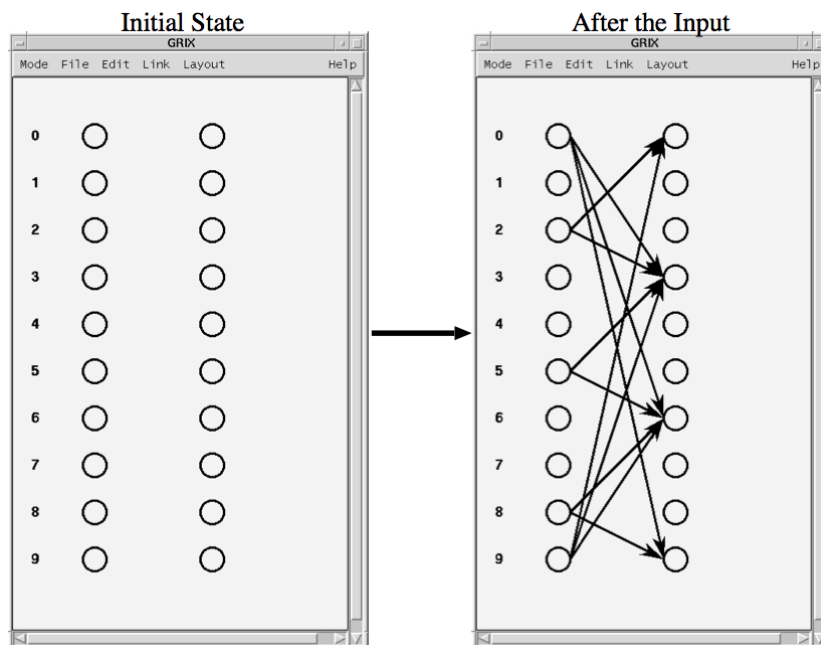


Figura 3.6 – Interface gráfica do GRIX [SMT00].

Cada círculo representa um nodo. Na Figura 3.6a, a tela inicial para uma execução com 10 nodos é apresentada. Cada nodo da coluna da esquerda representa o mesmo nodo na coluna da direita: os da esquerda são os que vão realizar operações de envio e

os da direita, intuitivamente, operações de recebimento. A Figura 3.6b mostra um exemplo de comunicações informadas pelo usuário na interface gráfica (através de eventos de mouse). Alguns tipos de comunicação comuns são previamente definidos na ferramenta, como Scatter e Broadcast, por exemplo. Depois de especificada a comunicação, GRIX gera código para PVM e MPI.

Uma abordagem diferente das demais é apresentada pelos autores de [AdTGR99]: MPI-Delphi. A ideia foi de reunir um dos ambientes gráficos mais conhecidos para o Sistema Operacional Windows com MPI, resultando em um ambiente gráfico para o desenvolvimento de aplicações paralelas para clusters. Para o desenvolvimento da parte paralela do MPI-Delphi, os autores utilizaram W32MPI v.0.9b [BDGS93], uma implementação do MPI para Windows. Como a utilização do Sistema Operacional Windows em todos os nodos do cluster acarretaria perda de desempenho nas comunicações, o MPI-Delphi utiliza a linguagem Object Pascal para a interface gráfica e a linguagem C para realizar as computações em estações Linux.

Em [CCCZ05], é apresentado o VisualGOP, um ambiente visual de programação bastante completo, que utiliza grafos para a definição do modelo da aplicação paralela. Os componentes principais são: Graph Editing Panel, no qual o usuário cria o grafo que representa a aplicação e Coding Editing Panel, no qual o usuário informa o código local de cada nodo. Entretanto, a ferramenta possui diversas outras funcionalidades, como, por exemplo, compilação, execução da aplicação, mapeamento de processos por prioridades etc.

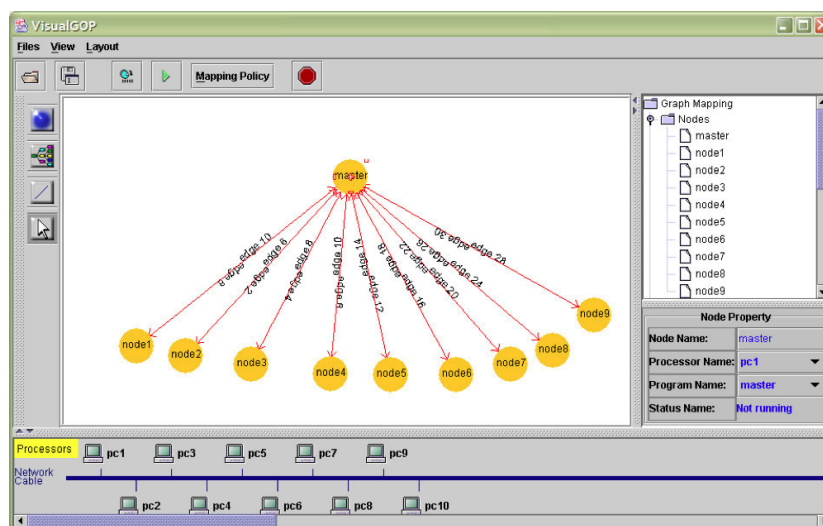


Figura 3.7 – Exemplo de interface do VisualGOP [CCCZ05].

A Figura 3.7 mostra um exemplo de modelo no qual um nodo mestre divide um array igualmente entre os demais nodos, que computam e retornam a resposta para o mestre. O código local de cada nodo pode ser escrito em C, C++ ou Java.

Como é possível perceber nos trabalhos apresentados anteriormente, o grande auge da pesquisa e do desenvolvimento de ferramentas gráficas para a programação paralela ocorreu nos anos 90. Dos 9 estudos supracitados, 7 são datados da década em questão. Além disto, os outros 2 trabalhos datam de 2000 e de 2005. Desde então, não são mais encontradas tantas pesquisas na área, mostrando que a área de programação paralela com interface gráfica tornou-se um pouco esquecida.

Mais recentemente, em 2013, Vanya Yaneva apresentou uma ferramenta chamada Vidim [Yan13]. Trata-se de uma linguagem de programação paralela visual que utiliza o modelo de programação paralela PGAS (*Partitioned Global Address Space*) e a linguagem de programação Mesham. Um exemplo da interface gráfica pode ser visualizada na Figura 3.8.

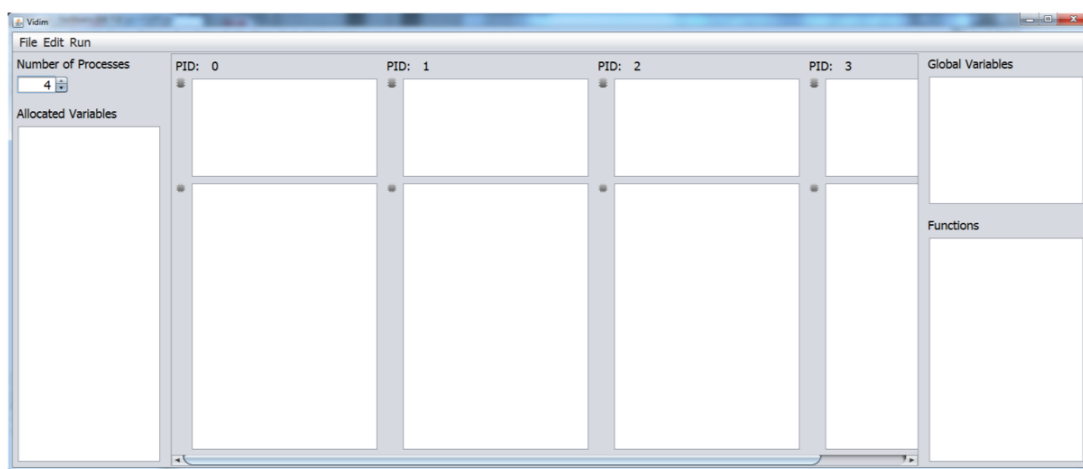


Figura 3.8 – Interface gráfica do VIDIM [Yan13].

Na parte superior esquerda da tela, o usuário informa a quantidade de nodos a serem utilizados. Logo abaixo, existe uma área onde as variáveis podem ser declaradas. Quando uma variável é declarada, seu tipo, tamanho, quais processos possuem esta variável e mais algumas informações. Vidim possui suporte à comunicação coletiva, para a qual o usuário deve informar o a variável a ser enviada, o local onde ela será recebida (variável destino) e o tipo da comunicação. Na versão atual, apenas a comunicação coletiva allreduce é implementada.

No ano de 2014, os autores de [BBMŠ14] propuseram uma ferramenta chamada Kaira. O objetivo da ferramenta é simplificar o desenvolvimento de programas paralelos em arquiteturas com memória distribuída com a utilização de MPI. Os autores relatam a complexidade de criar programas paralelos, tanto para os que possuem conhecimento na área quanto, principalmente, para os que não o tem. No trabalho, os autores utilizaram Coloured Petri Nets (Redes de Petri Coloridas) [JK09] como base para o desenvolvimento da aplicação visual. A Figura 3.9 ilustra um exemplo da interface da ferramenta Kaira.

A comunicação é informada pelo usuário de forma visual, mas as partes sequenciais dos códigos (escritas em C++) são inseridas manualmente no modelo gráfico. Conforme

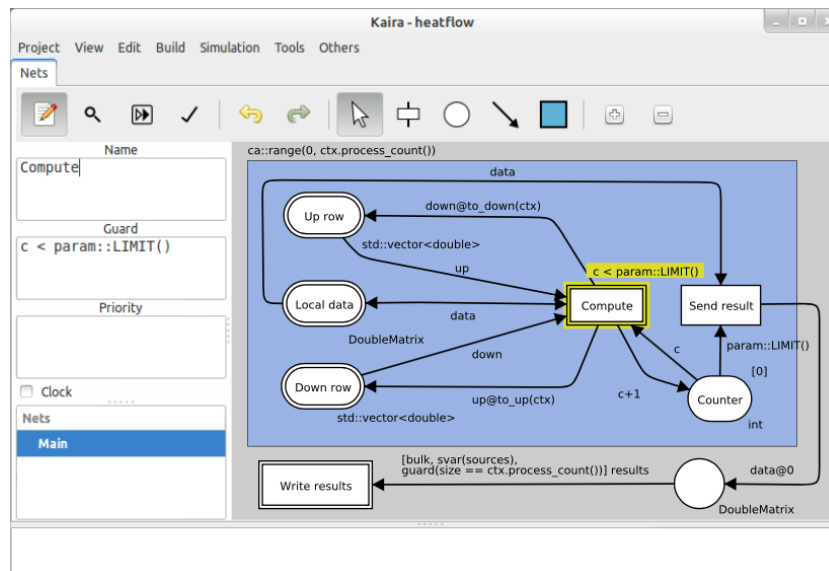


Figura 3.9 – Exemplo da interface visual do Kaira [BBMŠ14]

pode ser visto na Figura 3.9, a utilização de lugares e transições denotam espaços de memória e ações, respectivamente. As anotações textuais informam a comunicação entre os nodos através de marcações com , indicando um envio para determinado processo. Apesar de ser uma aplicação aparentemente interessante em termos de programação paralela de forma visual, os autores não mostram qualquer tipo de testes e resultados avaliativos da ferramenta.

Os estudos mais recentes sobre o assunto datam de 2016. O primeiro deles trata-se do trabalho apresentado em [FAGC16]. Nele, os autores propuseram um ambiente chamado GD-MPI (Graphical Development Environment for the construction of Parallel MPI Applications). O ambiente apresenta uma ferramenta que possibilita o desenvolvimento visual de aplicações paralelas com MPI através de interações com uma interface gráfica. O MPI utilizado para o desenvolvimento das aplicações é o Java-MPI e o sistema é utilizado através de uma aplicação Web. A Figura 3.10 mostra um exemplo da interface do GD-MPI.

Na ferramenta em questão, a criação do modelo paralelo é feita através da utilização de diferentes ícones, que são configurados pelo usuário de acordo com o que ele deseja. Os ícones, então, definem uma computação a ser realizada, o formato de saída dos dados (saída na tela ou escrita em um arquivo) e as operações coletivas como (broadcast, scatter e gather). Como testes, os autores apresentaram as implementações de duas aplicações: Mandelbrot Set e Merge Sort. Entretanto, os autores não mostram os resultados dos programas gerados, não sendo possível, assim, obter informações sobre o desempenho da ferramenta.

Outro estudo apresentado em 2016 pode ser encontrado em [RE16]. Nele, os autores propõem uma linguagem visual para o desenvolvimento de aplicações paralelas de alto desempenho. A linguagem desenvolvida é baseada em grafos direcionais, no qual os

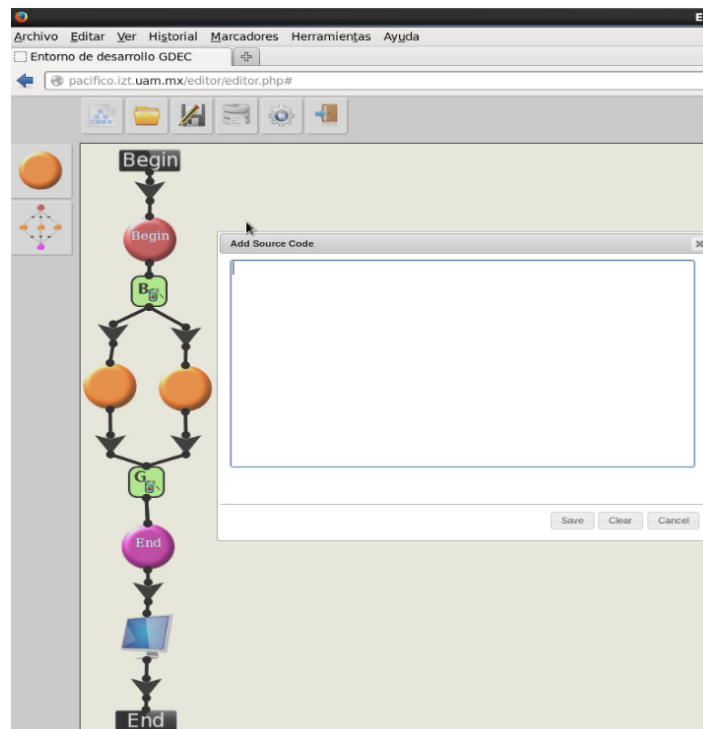


Figura 3.10 – Interface do GD-MPI [FAGC16]

vértices representam, basicamente, processamentos e as arestas representam comunicações. Os vértices de processamento possuem entradas de dados (vértices) e possuem código em uma linguagem de programação. O estudo apresentado é bastante incipiente, mas mostra que a comunidade da área continua realizando pesquisas relacionadas ao tema.

### 3.4 Considerações finais

Todos os trabalhos apresentados nas seções anteriores possuem uma relação interessante com o trabalho desenvolvido. Primeiramente, a Seção 3.1 mostra a preocupação de pesquisadores e grupos de pesquisa no âmbito da alocação eficiente de memória para máquinas NUMA. Neste contexto, a premissa de que é importante e necessário permitir que o usuário controle a política de afinidade de suas aplicações é fortalecida. Além disto, os trabalhos levantados trazem fundamentações e ideias que servem para o contexto da presente pesquisa, como a utilização da biblioteca MAI para realizar a alocação eficiente de memória.

A Seção 3.2, por sua vez, elenca um conjunto de frameworks e ferramentas que disponibilizam padrões de programação paralela, os *skeletons*, para os desenvolvedores. Embora a ideia de facilitar (de alguma forma) a programação para ambientes paralelos não seja nova, esforços constantemente são realizados, como mostram os trabalhos descritos. Este fato corrobora um dos objetivos desta tese sobre a importância de fornecer um modelo



amigável, com suporte, no mínimo, aos padrões mais comuns e que apresente facilidade de conversão/integração de aplicações já existentes.

Finalmente, na Seção 3.3 diversos trabalhos interessantes são descritos, todos apresentando uma interface visual com o usuário (programador). Destes trabalhos, alguns aspectos se sobressaem, e foram de extrema importância para o desenvolvimento realizado nesta tese:

- Forma de representação dos processos: através de desenhos, geralmente indicando um grafo dirigido;
- Forma de representação da comunicação: através dos arcos que ligam os nodos dos grafos, o fluxo da comunicação é conhecido;
- Parametrização da comunicação: o usuário pode informar aos modelos os tipos de dado que deseja trabalhar;
- Inclusão de código dos processos: alguns trabalhos mostram a ideia de clicar no nodo e abrir uma janela, o que se trata de uma abordagem interessante. Entretanto, o mais importante é o modelo por baixo: é possível incluir código nos nodos do grafo;
- Abstração de primitivas de comunicação: a principal ideia da maioria das ferramentas e modelos elencados é tirar do programador a necessidade de um conhecimento maior de primitivas de comunicação entre nodos.

Conforme descrito em seções anteriores, o trabalho se apoia fortemente nos três eixos apresentados neste capítulo: alocação eficiente de memória para máquinas NUMA, padrões de programação paralela (*skeletons*) e interface com o usuário. Nas pesquisas realizadas, foi encontrado um modelo para a criação e parametrização dos padrões de programação paralela desejados que permita ao usuário definir políticas de alocação de memória. No decorrer dos capítulos restantes, o modelo desenvolvido será descrito em maiores detalhes.



## 4. MAPEAMENTO DE PROCESSOS PESADOS COM PADRÕES

Ao desenvolver aplicações paralelas, usuários (programadores<sup>1</sup>) devem definir e identificar os processos pesados a serem utilizados na sua aplicação, bem como o que cada um destes processos deve computar.

Um ambiente paralelo deve ser capaz de lidar com dois tipos de processos: processos pesados e processos leves. Um processo leve, chamado de *thread*, é um fluxo de controle que executa dentro de um programa. Pode ser vista como um pequeno programa que executa dentro de um programa principal. Um processo pesado, por sua vez, trata-se de um programa que possui uma única *thread*, ou seja, um único fluxo de execução. Quando as tarefas de um programa são executadas concorrentemente, diz-se que o programa é multithread [SGG09].

Este capítulo aborda a maneira na qual o modelo desenvolvido realiza o mapeamento dos processos pesados para as máquinas de um *cluster*, utilizando padrões de programação paralela. Este mapeamento é realizado de forma totalmente transparente para o usuário.

### 4.1 Visão Geral do Processo de Mapeamento

A Figura 4.1 ilustra a visão geral do processo de mapeamento de processos pesados com a utilização de padrões. A finalidade deste modelo é que o usuário informe as configurações desejadas para sua aplicação paralela e que cada processo definido seja automaticamente mapeado para uma máquina do *cluster*.



Figura 4.1 – Visão geral do processo de mapeamento de processos pesados com padrões.

Conforme ilustra a Figura 4.1, o processo de mapeamento deve receber basicamente três informações vindas do usuário: um grafo dirigido, as configurações de comunicação e os códigos das computações a serem realizadas por cada processo. O fluxo de funcionamento do modelo proposto para o mapeamento é descrito a seguir.

<sup>1</sup>A partir deste ponto, a palavra usuário será utilizada no texto para representar desenvolvedores de aplicações paralelas que farão uso do modelo desenvolvido.

A aplicação paralela é representada por um grafo dirigido. Uma estrutura como um grafo é propícia para o projeto de aplicações paralelas, e diversos *frameworks* e ferramentas utilizam esta abordagem. Neste sentido, os processos pesados são representados pelos nodos do grafo e as comunicações pelos arcos que os interligam. O sentido dos arcos define qual dos processos está enviando (*send*) e qual dos processos está recebendo (*recv*). As comunicações aceitas pelo modelo podem ser tanto uni quanto bidirecionais. A Figura 4.2 apresenta exemplos de comunicações.

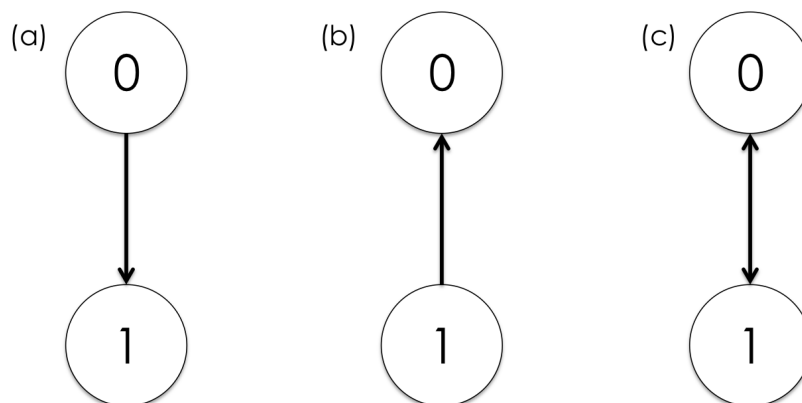


Figura 4.2 – Exemplos de fluxos de comunicação possíveis.

Na Figura 4.2a, é possível perceber que *Process 0* está realizando uma operação de envio (*send*) para *Process 1*, enquanto *Process 1*, por sua vez, está realizando uma operação de recebimento (*recv*) de *Process 0* (a seta saindo de um nodo a e chegando em um nodo b indica uma comunicação de envio no sentido de a para b). Na Figura 4.2b, o contrário acontece, quando *Process 0* recebe algum(s) dado(s) de *Process 1*. A Figura 4.2c, por sua vez, ilustra uma situação na qual ambos os processos realizam as duas operações: *Process 0* envia uma mensagem para *Process 1* e aguarda o recebimento de uma resposta (também de *Process 1*), enquanto *Process 1* aguarda o recebimento de dados de *Process 0* e envia uma mensagem para *Process 0*.

Para evitar *deadlocks* e problemas de sincronização, a ordem da comunicação sempre respeitará o processo que faz o envio primeiramente. Assim sendo, na Figura 4.2c, se o usuário definir o arco bidirecional partindo de *Process 0* e chegando em *Process 1*, *Process 0* realizará o primeiro envio e *Process 1* o primeiro recebimento.

Com isto, cada nodo do grafo informado pelo usuário tornar-se-á um processo MPI. Um nodo não necessariamente refere-se a apenas um processo. O usuário, então, pode indicar que um determinado nodo refere-se a um conjunto de processos, permitindo que comunicações coletivas sejam realizadas na aplicação paralela. A Figura 4.3 ilustra um exemplo onde um nodo do grafo trata-se de um processo, enquanto outro nodo do grafo trata-se de um grupo de processos.

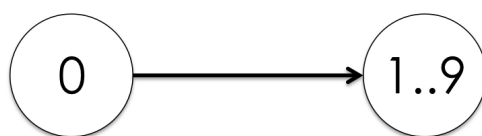


Figura 4.3 – Comunicação entre um processo e um grupo de processos.

No exemplo, o processo MPI de *rank* 0 envia uma mensagem para os processos de rank 1 até 9, que respondem ao processo 0 quando finalizarem sua computação. Os grafos que representam as aplicações estão diretamente relacionados com os padrões paralelos escolhidos, que serão detalhados na Seção 4.3.

Após definido o grafo da aplicação paralela, as comunicações (ou seja, os arcos) devem ser parametrizadas. Isto significa que o usuário pode informar, para cada envio a ser realizado, diferentes configurações, tais como:

- Quantidade de dados: a cada comunicação, é possível enviar mais de um dado. Por exemplo, ao definir que um envio deve acontecer do processo 0 para o processo 1, o usuário pode dizer que deseja enviar  $n$  dados diferentes, e o modelo realizará  $n$  envios;
- Estrutura a ser enviada: para cada dado enviado, é possível indicar se trata-se de uma variável simples, um *array* unidimensional ou um *array* bidimensional. Adicionalmente, foi incorporado no modelo desenvolvido a possibilidade de envio de `std::string` e `std::vector` (tipos de dados em C++);
- Tipo de dados: para cada estrutura a ser enviada, podem ser definidos os tipos de dado disponíveis, de acordo com os tipos do MPI, sendo os principais: `char`, `int`, `long int`, `double`, `short` e `float`.

Caso a estrutura de dado a ser enviada seja um *array* unidimensional, um *array* bidimensional, um `std::vector` (de qualquer tipo) ou `std::string`, o modelo permite que os dados sejam divididos entre os processos de diferentes formas:

- enviar a estrutura inteira para todos os processos destino: no caso de envio para um grupo de processos, a estrutura será enviada para todos os processos do grupo em *broadcast*. Para isto, o modelo utiliza a primitiva *Bcast* do MPI<sup>2</sup>;
- dividir a estrutura igualmente entre os processos destino: no caso de envio para um grupo de processos, a estrutura será dividida igualmente para todos os processos do grupo. Para isto, o modelo utiliza a primitiva *Scatter* do MPI;

<sup>2</sup>Cabe ressaltar que a utilização das primitivas é realizada de maneira transparente ao usuário, não importando ao usuário como os envios e recebimentos são implementados.

- dividir a estrutura em tarefas: para este tipo de divisão, são utilizadas as primitivas *Send* e *Recv* do MPI. O usuário informa a quantidade  $T$  de tarefas a serem criadas e a estrutura será quebrada em  $T$  partes. Esta divisão só pode ser realizada quando a comunicação for do tipo bidirecional. Seja  $P$  o número de processos do grupo destino. A divisão, neste caso, ocorre da seguinte maneira:

- se  $T \leq P$ , cada processo do grupo destino receberá uma tarefa e enviará o resultado de sua computação para o processo origem;
- se  $T > P$ , o processo origem divide as primeiras  $n$  tarefas ( $n \in T$ ) aos  $P$  processos do grupo de destino e aguarda alguma resposta. Quando um processo  $p$  ( $p \in P$ ) retorna ao processo origem a resposta do seu processamento, o processo origem recebe a resposta e verifica se ainda existe alguma tarefa em  $T$  para ser enviada. Caso exista, envia mais uma tarefa para o processo  $p$ . Caso não existam mais tarefas, envia uma mensagem de finalização ao processo  $p$ .

Outro aspecto importante para o usuário é a possibilidade de definir uma redundância no envio dos dados. Isto significa que ao enviar *arrays* unidimensionais, *arrays* bidimensionais, `std::vector` ou `std::string`, pode ser definida uma “borda”, que será enviada para mais de um processo. Por exemplo, suponha a matriz  $A$  ilustrada na Figura 4.4a. Caso ela fosse dividida entre 4 processos, cada um deles ficaria com 3 linhas. Ao definir uma redundância, pode-se informar ao modelo que se deseja uma borda de tamanho 1, por exemplo. Assim, o dado é quebrado entre os processos e é acrescentada uma linha acima e uma linha abaixo para cada um deles (Figura 4.4b).

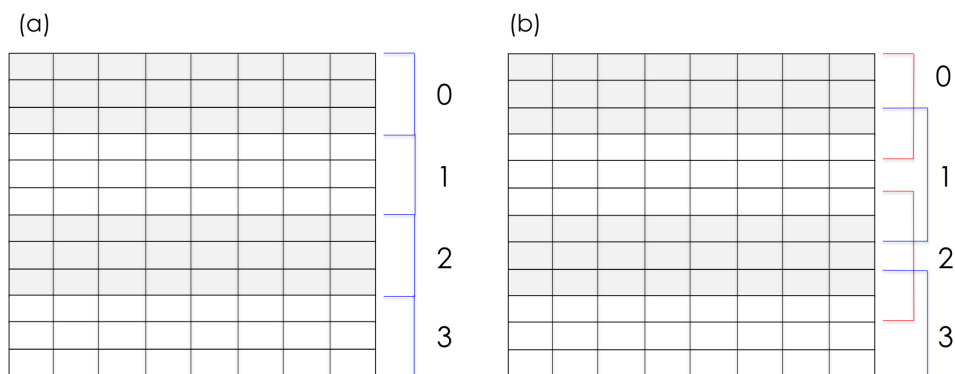


Figura 4.4 – Exemplo de redundância de dados (“borda”).

Tal abordagem pode ser bastante útil em aplicações de busca em textos ou aplicações de manipulação de imagens, por exemplo.

Através da combinação do grafo da aplicação paralela com as configurações das comunicações informadas, é criado um arquivo com extensão `.cpp` (C++) para cada nodo do grafo. Neste ponto, é importante ressaltar a existência de um gerador de código no

modelo apresentado, que cria este arquivo fonte. O conteúdo deste arquivo é um template, com algumas partes a serem preenchidas pelo usuário.

Estes arquivos estarão ligados com uma classe chamada `mpiMapping`, através do mecanismo de herança em Programação Orientada a Objetos. A classe `mpiMapping`, na realidade, serve como uma interface entre o código sequencial do usuário e as rotinas de comunicação do MPI, além de permitir que algumas variáveis globais sejam acessadas (Seção 4.4). A Figura 4.5 ilustra a ideia dos arquivos criados, de acordo com um exemplo simples de modelo.

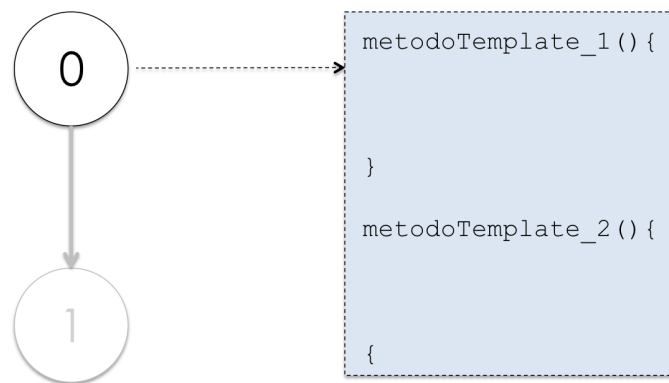


Figura 4.5 – Exemplo de estrutura de arquivo template gerado para cada processo.

Cada nodo (processo) do grafo, então, tornar-se-á um arquivo fonte, automaticamente gerado pelo modelo de mapeamento, e que herda `mpiMapping`. Pode-se visualizar na Figura 4.5 que os arquivos fonte dos processos possuem lacunas a serem preenchidas pelo usuário. Estas lacunas referem-se a códigos sequenciais que o usuário vai inserir. É interessante notar que o modelo desenvolvido permite que o usuário possa inserir códigos que ele já possui implementado. Rotinas sequenciais que serão agora realizadas em paralelo podem ser simplesmente copiadas para os arquivos *template*. De maneira análoga, o usuário pode simplesmente inserir no *template* a chamada a outros códigos fonte existentes, bastando, para isto, inserir os *#includes* dos arquivos necessários.

O *template* do código fonte gerado depende do modelo criado para a aplicação. A Seção 4.3 descreve os modelos previamente implementados, detalhando o conteúdo dos arquivos *template* para cada caso.

## 4.2 Estrutura e Controle

Para realizar o controle do mapeamento e a sincronização das comunicações, a estrutura do modelo desenvolvido (em forma de um diagrama simplificado de classes) é apresentada na Figura 4.6.

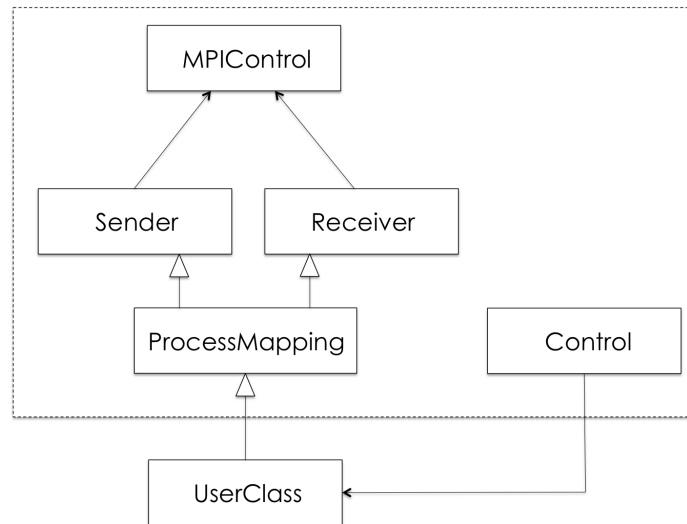


Figura 4.6 – Estrutura interna do mapeamento de processos pesados.

A classe `MPIControl` é responsável por fornecer aos demais módulos informações como *rank* de cada processo e quantidade de processos total na execução, por exemplo.

A classe `Sender` é responsável por encapsular todas as primitivas de envio do MPI necessárias para o funcionamento do modelo de mapeamento. Ainda na classe `Sender`, encontram-se os envios em *broadcast* e a chamada ao *Scatter*. A classe `Receiver`, por sua vez, é responsável por todas as primitivas de recebimento do MPI.

A classe `ProcessMapping` é responsável por fazer a interface do usuário com as classes do modelo de mapeamento. Assim sendo, como pode ser visto, as classes do usuário (`UserClass`, que representam os processos MPI) vão herdar a classe `ProcessMapping`. Como os processos podem tanto enviar quanto recebem mensagens de outros processos, `ProcessMapping` utiliza o mecanismo de herança múltipla do C++, para prover primitivas de envio e recebimento de dados, conforme necessário.

Finalmente, a classe `Control` é responsável por inicializar o ambiente MPI e controlar o fluxo de execução da aplicação paralela. De acordo com o grafo informado e as configurações de comunicações, esta classe chama os métodos necessários da `UserClass`.

### 4.3 Padrões implementados (*skeletons*)

Conforme descrito anteriormente, para que os *templates* dos programas paralelos sejam criados, é necessário que o usuário informe seu grafo e configure suas comunicações. Entretanto, muitas aplicações paralelas tradicionalmente utilizam alguns padrões de programação bem conhecidos.



Neste contexto, o modelo de mapeamento de processos pesados provê alguns destes padrões (ou *skeletons*) já implementados. Isto indica que o programador, ao escolher algum destes, precisa apenas configurar as comunicações.

No decorrer desta seção, os modelos implementados *Master/Slave*, *Pipe* e *D&C* são apresentados, além de um modelo genérico, no qual o usuário pode criar seu próprio grafo.

#### 4.3.1 Master/Slave

Nesta seção, será detalhado como o padrão *Master/Slave* foi implementado no modelo proposto. A Figura 4.7 ilustra um grafo para o modelo *Farm* com 11 processos.

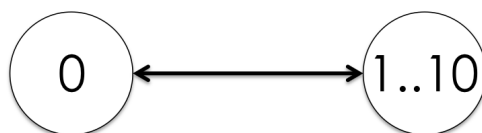


Figura 4.7 – Comunicação bidirecional entre um processo e um grupo de processos.

Conforme pode ser visto, o modelo *Master/Slave*, para 11 processos, gera um grafo com dois nodos: *Process 0* e *Processes 1 to 10* (um grupo com 10 processos). A comunicação entre eles é bidirecional, ou seja, *Process 0* envia os dados para *Processes 1 to 10* e *Processes 1 to 10* e vice-versa, começando pelo envio de *Process 0*.

Ao usuário, então, basta definir as configurações da comunicação e gerar o *template*. De acordo com o que já foi descrito, o modelo gera um arquivo `.cpp` para cada nodo do grafo. Isto indica que o modelo *Master/Slave* gera dois arquivos (neste caso, `Process0.cpp` e `Processes1to10.cpp`). Com os *templates* criados, o usuário completa o código de cada arquivo.

Quando um grupo de processos é criado, como *Processes 1 to 10*, por exemplo, significa que todos estes processos recebem os mesmos tipos de dados, retornam os mesmos tipos de dados e realizam a mesma computação sobre os dados recebidos. O que difere um nodo do outro é o valor dos dados recebidos, que serão enviados pelo *Process 0*.

No arquivo *template* de um processo  $p$  qualquer, os métodos `beforeCommunication_n` e `afterCommunication_n` são criados automaticamente para cada comunicação  $n$  realizada pelo processo  $p$ . Estes métodos possuem parâmetros de entrada e de saída, de acordo com o que foi definido no grafo da aplicação. Todos os parâmetros anteceditos pelo símbolo `&` (“e” comercial, caracterizando uma passagem por referência no C++) indicam que serão parâmetros de saída, e os dados a serem enviados devem ser atribuídos a estas variáveis.

A classe `Control` define o fluxo do modelo Master/Slave. O processo de *rank* 0 do MPI é mapeado para *Process 0* (`Process0.cpp`), assim como os processos de *ranks* 1 até 10 do MPI são mapeados para *Processes 1 to 10* (`Processes1to10.cpp`). A sequência de passos de *Process 0* é listada a seguir:

- O método `beforeCommunication_1` de *Process 0* é chamado, indicando os dados necessários para o envio por parâmetro;
- É realizado o envio dos dados aos processos de 1 até 10, de acordo com o definido nas configurações do mapeamento;
- O método `afterCommunication_1` de *Process 0* é chamado, com os dados correspondentes;
- O método `beforeCommunication_2` de *Process 0* é chamado, com os dados correspondentes;
- É realizado o recebimento dos dados dos processos de 1 até 10, de acordo com o que foi definido nas configurações do mapeamento;
- O método `afterCommunication_2` de *Process 0* é chamado, com os dados recebidos dos processos de 1 até 10.

Em paralelo à execução do processo 0, a sequência de passos realizada pelos processos de 1 até 10 é listada a seguir:

- O método `beforeCommunication_1` de *Process 1 to 10* é chamado, com os parâmetros correspondentes;
- É realizado o recebimento dos dados do processo 0, de acordo com o definido nas configurações do mapeamento;
- O método `afterCommunication_1` de *Process 1 to 10* é chamado, com os dados recebidos do processo 0;
- O método `beforeCommunication_2` de *Process 1 to 10* é chamado, com os dados correspondentes;
- É realizado o envio dos dados ao processo 0, de acordo com o que foi definido nas configurações do mapeamento;
- O método `afterCommunication_2` de *Process 1 to 10* é chamado, com os dados correspondentes.

O fluxo da comunicação quando os dados a serem enviados forem divididos em tarefas apresenta algumas pequenas modificações. A principal delas é que o processo *Master* fica em um *loop*, enviando e recebendo tarefas, até que estas acabem. Só então, quando não houver mais tarefas e receber todas as respostas, o processo *Master* finaliza a computação, utilizando os resultados recebidos pelos *Slaves*.

#### 4.3.2 Pipe

A implementação do padrão *Pipe* em relação ao modelo de mapeamento de processos pesados é descrita a seguir. A Figura 4.8 ilustra um exemplo de grafo para o padrão *Pipe*.

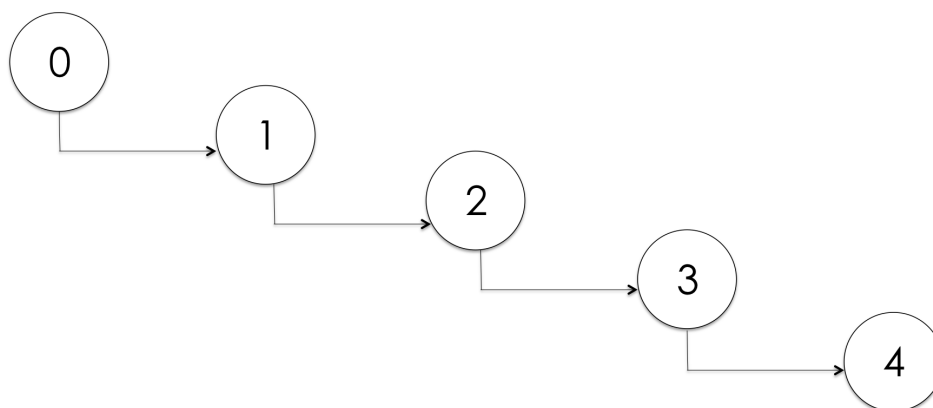


Figura 4.8 – Exemplo de *Pipe* com 5 processos.

O modelo *Pipe* indica que a saída de um processo é a entrada do seu sucessor imediato, e cada processo deve realizar sua computação específica. No modelo desenvolvido, então, será necessário que o usuário informe o código de cada processo, determinando a computação sequencial e os dados que devem ser repassados. Isto indica que o modelo de mapeamento criará um arquivo fonte para cada um dos processos do *Pipe*. No exemplo da Figura 4.8, seriam criados os arquivos `Process0.cpp`, `Process1.cpp`, `Process2.cpp`, `Process3.cpp` e `Process4.cpp`.

Além disto, o tipo de dado recebido e enviado pelos processos pode ser diferente. Isto significa, por exemplo, que *Process 2* pode ter como resultado de sua computação um *array* unidimensional de inteiros (que será a entrada de *Process 3*), enquanto *Process 3* pode enviar ao próximo processo um simples caractere (`char`). O modelo foi desenvolvido desta maneira para dar maior flexibilidade aos programas gerados com este padrão de programação. Para editar as configurações de cada comunicação, o usuário seleciona o *incoming* desejado e utiliza a *Configuration Area*.

Após a definição dos dados a serem comunicados, as computações sequenciais, então, são inseridas em cada processo. Existem 3 tipos diferentes de processos no modelo Pipe: *InitialProcess*, *IntermediateProcesses* e *FinalProcess*. A nível de exemplo, para o grafo da Figura 4.8 teríamos:

*InitialProcess*: *Process 0*

*IntermediateProcesses*: *Process 1, 2 e 3*

*FinalProcess*: *Process 4*

Todos estes processos possuem, em seu *template*, apenas um método a ser definido pelo usuário: `functionToApply`. Este método, de acordo com a posição no Pipe, possuirá parâmetros específicos, conforme especificado nas configurações do modelo. A diferença básica entre os tipos de processos é que *InitialProcess* tem como função inicializar os dados antes de realizar sua computação, *IntermediateProcesses* realizam uma computação de acordo com o que recebem do processo anterior e enviam uma resposta ao próximo processo e, finalmente, *FinalProcess* realiza sua computação com o que for recebido do processo anterior e finaliza a computação. A Figura 4.9 mostra um exemplo do *template* gerado para cada tipo de processo.

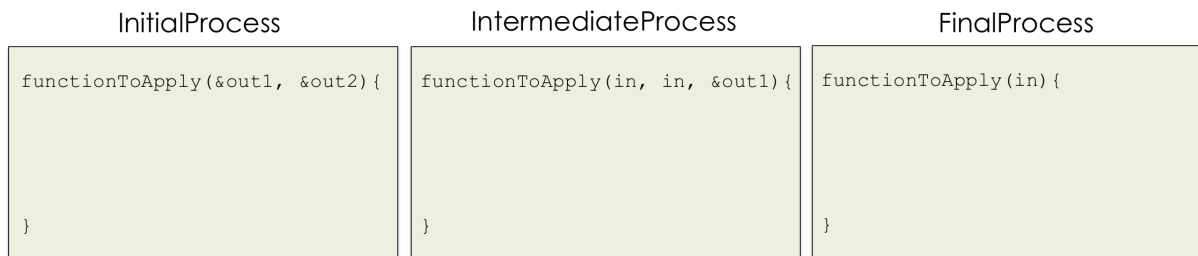


Figura 4.9 – Exemplo de template gerado.

É possível perceber que a diferença principal entre os arquivos de acordo com o tipo de processo são os parâmetros. O método do *InitialProcess* possui apenas parâmetros de saída; o método do *IntermediateProcesses* possui parâmetros de entrada e parâmetros de saída, e; o método do *FinalProcess* possui apenas parâmetros de entrada.

Conforme descrito anteriormente, a utilização do modelo Pipe é bastante útil principalmente quando diversos dados devem ser computados e cada processo do Pipe responsabiliza-se por determinada tarefa. Assim, o Pipe apresentará ganho no desempenho em relação a versões sequenciais quando estiver sendo totalmente utilizado. Assim, é previsto no modelo de mapeamento a definição de uma configuração adicional, chamada *Rounds*. A quantidade de dados a serem processados na aplicação paralela do usuário é definida pelo valor de *Rounds*. Assim sendo, cada *Round* é um dado novo a ser computado pela aplicação.

### 4.3.3 Divide and Conquer (D&C)

Esta seção apresenta o modelo D&C implementado no modelo de mapeamento em questão. Este modelo foi implementado de duas maneiras distintas. A primeira delas trata-se de uma versão estática do modelo, na qual o usuário deve informar a quantidade de processos desejada, o modelo cria o grafo de acordo com o valor informado. A segunda maneira utiliza a primitiva `Spawn`, presente a partir da versão 2 do MPI. Com esta abordagem, o modelo de mapeamento desenvolvido cria processos dinamicamente, conforme for necessário. Ou seja, de acordo com a condição de parada do modelo, mais um processo será criado.

A principal diferença entre as duas versões é em relação à eficiência da execução da aplicação final do usuário. Por exemplo, uma aplicação que utiliza o modelo D&C com 16 processos, mas precisa apenas de 11 pela condição de parada informada, acarretará na subutilização de recursos, pois 5 processos não realizariam computação alguma. Obviamente, o usuário pode informar quantos processos a aplicação precisa, e poderiam ser definidos apenas os 11 processos necessários. Entretanto, o usuário precisaria verificar a condição de parada e realizar cálculos antes de cada execução, para alocar a quantidade de processos correta.

O modelo criado, então, utiliza apenas 1 nó no grafo do modelo para definir os métodos *template* e a comunicação a ser realizada por cada processo. Isto é possível em virtude de que, no padrão D&C, a computação de todos os processos é a mesma, bem como o critério de parada. Assim sendo, o usuário deve informar a condição de parada e as computações de cada nó, deixando que o controle cuide do restante (como a criação de novos processos quando necessário, por exemplo).

Os métodos definidos para o modelo D&C são os seguintes:

**initialProcessing** método criado apenas no processo 0, raiz da árvore do padrão D&C.

Neste método, são definidos os valores a serem divididos entre os demais processos;

**split** este método é comum a todos os processos, e retorna um valor booleano, que deve ser verdadeiro quando a condição de parada ainda não foi atingida (ou seja, deve continuar dividindo) e falso caso contrário;

**conquer** este método é usado para realizar alguma computação sobre os dados antes da conquista. Este método está presente em todos os processos, tratando-se da conquista. A saída deste método será enviada ao processo pai;

**finalProcessing** este método é criado apenas no processo 0, raiz da árvore. Ele é chamado pelo controle para finalizar a computação com as conquistas recebidas.

Os métodos listados devem ser completados pelo usuário com seus códigos. Objetivando fornecer maior facilidade e agilidade para o programador, definiu-se que todos os métodos são implementados em um único arquivo. A classe `Control`, então, será responsável por atribuir a cada processo os respectivos métodos e controlar toda a criação de novos processos.

#### 4.3.4 Modelo genérico

O desenvolvimento de modelos específicos, como `Master/Slave`, `Pipe` e `D&C`, bem como a percepção sobre o comportamento de cada um deles, culminou na possibilidade de criação de um modelo genérico para as aplicações paralelas. Neste modelo, o usuário é quem informa os nodos que deseja em seu modelo e indica o fluxo das comunicações a serem realizadas, ou seja, é o usuário quem cria o grafo da aplicação.

Seguindo a mesma linha dos demais modelos, cada nodo do grafo criado será mapeado para um processo MPI. Em cada processo do grafo, então, existem 2 métodos a serem implementados, analogamente ao modelo `Master/Slave`: toda a comunicação definida no grafo informado possuirá um método que se refere a um processamento a ser realizado antes da comunicação e um método que refere-se a um processamento a ser realizado depois da comunicação. Respectivamente, estes métodos são chamados de `beforeCommunication_X` e `afterCommunication_X`, onde `X` representa o número da comunicação. O número da comunicação segue a ordem de cada comunicação definida para um processo<sup>3</sup>.

Suponha um grafo com 2 nodos. O primeiro nodo trata-se do processo de *rank* 0 e o outro nodo trata-se de um grupo de processos, de 1 a 10, por exemplo. O processo 0 deve realizar o envio de um array unidimensional de *double* para os demais processos. Assim, através da classe `Control` (responsável pelo controle de toda a execução paralela), serão realizadas as seguintes chamadas da classe `Process0.cpp` (nesta ordem): `beforeCommunication_1` e `afterCommunication_1`. A função de cada um destes métodos é a seguinte:

**beforeCommunication\_1** a função do método `beforeCommunication_1` é, principalmente, inicializar os dados a serem enviados na comunicação `X`. No exemplo, no qual *Process 0* deve enviar um *array* unidimensional de *double* para *Process 1*, os parâmetros do método `beforeCommunication_1` são, respectivamente, o *array* a ser enviado, o tamanho do *array* a ser enviado e o *Round*<sup>4</sup> (descrito anteriormente). Através dos

<sup>3</sup>Por exemplo, se foi definido que ele possui um envio e um recebimento, então, ele possuirá 2 comunicações (1 e 2) e, conseqüentemente, 4 métodos (`beforeCommunication_1`, `afterCommunication_1`, `beforeCommunication_2` e `afterCommunication_2`)

<sup>4</sup>Assim como no modelo `Pipe`, o *Round* trata-se de quantas vezes a aplicação receberá uma nova entrada

parâmetros, o modelo de mapeamento sabe os dados a serem enviados e realizará a comunicação destes de acordo com o que foi definido nas configurações;

**afterCommunication\_1** sempre após uma comunicação, seja ela um envio ou um recebimento, é possível que o processo necessite realizar algum processamento antes da próxima comunicação (caso exista outra comunicação). No exemplo em questão, por tratar-se de um envio, o método `afterCommunication_1` recebe por parâmetro os dados enviados, pois podem ser utilizados pelo próprio processo.

Analogamente, os métodos dos processos de *rank* 1 a 10 possuem os mesmos nomes (`beforeCommunication_1` e `afterCommunication_1`). O que difere os métodos são os parâmetros de entrada e suas funções na aplicação como um todo. Por exemplo: enquanto o método `beforeCommunication_1` do processo de *rank* 0 não pode ficar sem ser preenchido, pois realiza uma computação e armazena os dados a serem enviados, o método `beforeCommunication_1` dos demais processos talvez sirva para alguma inicialização necessária pela aplicação antes do recebimento, podendo este não possuir linha de código alguma.

Os parâmetros dos métodos `beforeCommunication_X` e `afterCommunication_X` dependem do tipo da comunicação *X*:

- Caso *X* seja um envio:

**beforeCommunication\_X** sempre recebe por parâmetro as variáveis da comunicação *X* – 1 (caso exista), as variáveis a serem enviadas na comunicação *X* e o valor do *round*;

**afterCommunication\_X** sempre recebe por parâmetro o que foi enviado na comunicação *X* e o valor do *round*;

- Caso *X* seja um recebimento:

**beforeCommunication\_X** sempre recebe por parâmetro as variáveis da comunicação *X* – 1 (caso exista) e o valor do *round*;

**afterCommunication\_X** sempre recebe por parâmetro as variáveis recebidas na comunicação *X* e o valor do *round*;

Para um processo com *C* comunicações, a classe `Control` sempre invoca os métodos de cada processo na ordem `beforeCommunication_1`, `afterCommunication_1`, `beforeCommunication_2`, `afterCommunication_2` e assim sucessivamente, até `beforeCommunication_X` e `afterCommunication_X`.

Com isto, o usuário pode incluir seu código no local desejado, tendo em vista o que será enviado e recebido por cada processo a cada momento. A Figura 4.10 ilustra o

código de um processo que recebe um *array* bidimensional de inteiros, juntamente com sua quantidade de linhas e de colunas. O código dos métodos foi incluído apenas para exemplificar sua utilização, e apenas imprime o *array* recebido. Naturalmente, qualquer outra operação poderia ser realizada com este dado.

```
void Process0::beforeCommunication_1(int round){
    cout << "Vou receber..." << endl;
}

void Process0::afterCommunication_1(int** input_1, int rowsInput_1, int colsInput_1, int round){
    for(int i=0; i<rowsInput_1; i++){
        for(int j=0; j<colsInput_1; j++){
            cout << input_1[i][j];
        }
        cout << endl;
    }
}
```

Figura 4.10 – *Exemplo de código: recebimento de array bidimensional de inteiros.*

Os dados são, então, enviados e recebidos de forma transparente para o usuário, sem que ele preocupe-se em conhecer primitivas de envio e recebimento, tampouco com sincronizações, uma vez que é o modelo de mapeamento que controla toda a execução. A vantagem em utilizar padrões predefinidos é de que a estrutura e funcionamento já são previamente conhecidos pelo modelo, permitindo que o controle seja realizado de maneira mais específica para cada caso. Com os modelos conhecidos, a programação é mais intuitiva para o usuário e modelo é menos intrusivo no desempenho final da aplicação.

#### 4.3.5 Comparativo entre os padrões

A Tabela 4.1 apresenta um comparativo entre os padrões implementados no modelo proposto. Na referida tabela foi realizada uma síntese dos principais aspectos de cada modelo.

É possível verificar algumas diferenças importantes entre os padrões. A primeira refere-se ao número de nodos no grafo necessários para cada um deles. O padrão Master/Slave possui apenas 2 nodos, um para cada tipo de processo (mestre e escravo), enquanto o padrão D&C, por exemplo, possui apenas 1 nodo no grafo. Isto se deve ao fato de que é utilizada a primitiva Spawn do MPI-2, que cria os processos dinamicamente, conforme a necessidade da aplicação. Os padrões Pipe e Genérico precisam de um nodo para cada processo que o usuário desejar.

Em relação à comunicação, duas informações são pertinentes: se existe comunicação bidirecional (ou seja, o processo A envia mensagens para o processo B e vice-versa) e se a comunicação é explícita ou implícita (ou seja, se o usuário necessita informar as arestas de comunicação entre os processos). Conforme pode ser visualizado na Tabela



Tabela 4.1 – Comparativo entre os padrões implementados

	Master/Slave	Pipe	D&C	Genérico
Número de nodos no grafo	2	Número de processos informados	1	Número de processos informados
Comunicação bidirecional	Sim	Não	Sim	Sim
Comunicação implícita ou explícita	Explícita	Explícita	Implícita	Explícita
Arquivos template criados	2	1 por processo	1	1 por processo

4.1, apenas o padrão D&C possui a comunicação implícita, enquanto apenas o padrão Pipe não possui comunicação bidirecional entre os processos.

Por fim, ainda pode ser comparada a quantidade de arquivos template gerados por cada um dos padrões utilizados. O padrão Master/Slave cria um arquivo template para o mestre e outro para os escravos (um para todos os escravos, uma vez que eles farão a mesma computação). Já o padrão D&C cria apenas 1 arquivo template, no qual são definidas todos os códigos a serem executados pelos processos. Os padrões Pipe e Genérico, por sua vez, criam um arquivo template para cada nodo do grafo, indicando que cada processo fará uma computação diferente a ser indicada pelo usuário.

#### 4.4 Obtendo informações globais

O modelo desenvolvido possui algumas informações que podem ser obtidas e utilizadas pelo usuário em seus programas paralelos. São informações bastante úteis, e que em diversas vezes são necessárias para o correto funcionamento da aplicação. São elas:

**Rank do processo** o *rank* de um processo é muito utilizado no desenvolvimento paralelo, tanto por questões de *debug* quanto para criação de *offsets* ou definições de limites destinados a cada processo. Logo, o *rank* de cada processo pode ser acessado através da variável global *MyRank*;

**Quantidade de processos** assim como o *rank* de cada processo, saber a quantidade de processos participando da comunicação é uma informação bastante útil em diversas ocasiões. Logo, a quantidade de processos pode ser obtida através da variável global *Size*;

**argc e argv** os argumentos de entrada podem ser utilizados para definir valores sem a necessidade de recompilar o código, informando-os no momento da execução do programa. Assim sendo, para obter o valor do `argc` (quantidade de argumentos) e do `argv` (*array* com todos os argumentos informados para a aplicação), o modelo possui variáveis globais chamadas de `Argc` e `Argv`. Desta forma, os argumentos de entrada do programa paralelo são informados da mesma forma como no programa sequencial. Entretanto, para utilizá-los no template gerado, é necessário utilizar as variáveis globais supracitadas, que são responsáveis por armazenar os argumentos e disponibilizá-los em qualquer parte do código.

#### 4.5 Considerações finais

O modelo apresentado neste capítulo objetiva automatizar o máximo possível a tarefa de desenvolver programas paralelos para clusters. Isto significa que o programador não precisará utilizar primitivas MPI para o envio e recebimento de mensagens, bastando explicitar o esquema de comunicação entre as máquinas. A principal ideia do modelo supracitado é abstrair a sintaxe do comando de baixo nível do MPI.

Para informar o esquema de comunicação, o usuário deve criar um grafo dirigido. Cada nodo do grafo representa uma máquina e cada seta representa um fluxo de comunicação. Em seguida, o usuário configura os nodos do grafo definindo o tipo dos dados que serão trocados entre as máquinas. Depois de toda a comunicação configurada, arquivos serão criados para cada máquina, e basta que o usuário preencha os respectivos métodos com seus códigos sequenciais para que a paralelização automática seja realizada<sup>5</sup>.

O modelo apresentado neste capítulo trata da comunicação entre diferentes máquinas de um cluster. Entretanto, o usuário ainda poderá realizar mais um nível de mapeamento no seu código sequencial, desta vez utilizando os diferentes processadores de cada uma das máquinas descritas no grafo do seu modelo paralelo. Este tipo de mapeamento também poderá ser realizado de forma automática, e é apresentado no Capítulo 5.

---

<sup>5</sup>O Capítulo 7 apresenta a forma de criação do grafo pelo usuário, bem como a forma de configurar cada máquina e informar seus códigos sequenciais.

## 5. MAPEAMENTO DE MEMÓRIA

O presente capítulo apresenta os detalhes do mapeamento realizado para alocar a memória em arquiteturas NUMA de maneira transparente ao usuário. Está organizado como segue: a Seção 5.1 apresenta a visão geral e o cenário em que a aplicação se enquadra. Os detalhes do modelo e algumas considerações estão descritos na Seção 5.2.

### 5.1 Visão geral do processo de mapeamento

Esta seção aborda o modelo geral do processo de mapeamento. Este modelo foi desenvolvido para usuários que desejam explorar melhor as características de arquiteturas NUMA, sem que o usuário possua conhecimento sobre primitivas de mais baixo nível para a alocação ser realizada, bastando definir os parâmetros desejados.

O modelo desenvolvido utiliza a biblioteca MAI para auxiliar nas definições de políticas de alocação de memória. Conforme descrito anteriormente, a utilização da MAI pressupõe a utilização de *threads* na aplicação. Neste sentido, o processo de mapeamento foi desenvolvido para suportar aplicações implementadas com o padrão OpenMP, devido ao crescimento, disseminação, facilidade de uso e grande gama de aplicações que o utilizam. As aplicações de entrada, então, devem ser implementadas utilizando C ou C++ com OpenMP.

Aplicações que possuem operações com *arrays*, sejam eles de qualquer tipo ou dimensão, são interessantes candidatas neste contexto. A visão geral do funcionamento do sistema é apresentada na Figura 5.1.

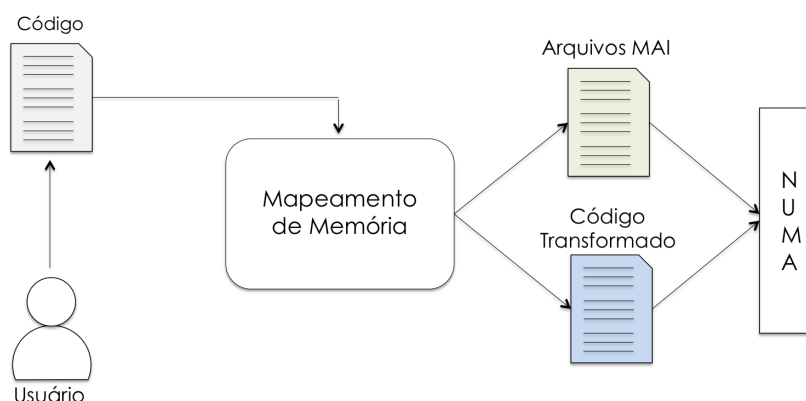


Figura 5.1 – Visão geral do mapeamento de memória.

Acompanhando a Figura 5.1, o fluxo de funcionamento da aplicação é dado por:

- o usuário escreve seu código fonte normalmente (em C ou C++) e salva onde desejar;

- o usuário indica ao modelo o(s) arquivo(s) no(s) qual(is) deseja aplicar alguma política de alocação de memória;
- são criados automaticamente, então, os seguintes arquivos:
  - (i) dois arquivos de alocação e configuração do ambiente;
  - (ii) os arquivos necessários para a execução da aplicação em um ambiente NUMA, incluindo a interface MAI;
  - (iii) o arquivo de configuração da MAI, de acordo com a política, número de nodos NUMA e número de *threads* informados;
  - (iv) código fonte do usuário transformado, através de um mapeamento que leva em consideração as configurações do usuário;
  - (v) um arquivo *README* com informações sobre a compilação do código final.
- o usuário pode acessar os arquivos criados, compilar e executar o código gerado.

Um dos pontos principais dos passos listados é a criação do arquivo de configuração da MAI. De acordo com a quantidade de *threads*, nodos NUMA e da política de afinidade de memória escolhida, é criado o arquivo *MAI.cf*, responsável pela configuração da máquina NUMA. Um exemplo de arquivo de configuração pode ser observado na Figura 5.2.

```
2 #number of nodes
0
1
5 #number of threads
0
1
2
3
4
```

Figura 5.2 – Exemplo de arquivo de configuração da MAI.

Para que o usuário possa interagir com o modelo, deve haver uma interface, permitindo-o informar os detalhes inerentes às estruturas de dados alvo, política de afinidade de memória escolhida, quantidade de nodos NUMA e quantidade de *threads*. No presente trabalho, foi desenvolvido um protótipo que disponibiliza uma interface gráfica para o usuário informar estas configurações. Detalhes do protótipo desenvolvido são descritos no Capítulo 7.

## 5.2 Estrutura e Controle

Os arquivos criados automaticamente pelo modelo incorporam as chamadas da biblioteca MAI. O código fonte do usuário transformado, então, inclui a biblioteca do modelo. O arquivo *memory\_management.c* (que tem como *header* o arquivo *memory\_management.h*) é gerado de acordo com o informado pelo usuário. Nele, é inicializado o ambiente da MAI, além de ser o local onde define-se como a arquitetura deve alocar as estruturas indicadas. A Figura 5.3 mostra um exemplo da utilização do arquivo *memory\_management.h* em um código transformado.

```
#include <stdio.h>
#include "memory_management.h"

#define LINHAS_A 500
#define COLUNAS_A 1000
#define LINHAS_B 1000
#define COLUNAS_B 500

int main() {
    init();
    int **m1 = allocate_structures(m1, LINHAS_A, COLUNAS_A, BIND_ALL);
    int **m2 = allocate_structures(m2, LINHAS_B, COLUNAS_B, BIND_ALL);
    int **m3 = allocate_structures(m3, COLUNAS_A, LINHAS_B, BIND_ALL);
    ...
}
```

Figura 5.3 – Exemplo de código transformado, utilizando o arquivo *memory\_management.h*.

É possível notar a inclusão do arquivo *memory\_management.h* no cabeçalho do código fonte. Este arquivo possui as chamadas para as alocações de memória e demais aspectos relacionados ao ambiente NUMA. O método *init* é responsável por ler o arquivo de configuração *MAI.cf* e inicializar o ambiente da MAI, passando a quantidade de nodos NUMA e a quantidade de *threads* a ser utilizada. O método *allocate\_structures* recebe por parâmetro o tipo do dado (no caso, indicado pela própria variável declarada), a quantidade de elementos e a política a ser utilizada. Como trata-se de um *array* bidimensional, é informado a quantidade de linhas e de colunas.

Dentro da implementação de *memory\_management.h* (ou seja, em *memory\_management.c*) são incluídas todas as inicializações da MAI, chamadas de *bind* e dos métodos de alocação (como *alloc\_2D* por exemplo). Toda esta transformação de código, bem como a geração dos arquivos *memory\_management.h* e *memory\_management.c* são realizadas automaticamente pelo modelo de mapeamento de memória desenvolvido.

Nota-se que o usuário não necessita criar arquivos de configuração, nem alocar estruturas seguindo o padrão da biblioteca MAI (e de nenhuma outra), tornando transparente a tarefa de alocação eficiente de memória.



## 6. MAPEAMENTO HÍBRIDO

O desenvolvimento dos modelos de mapeamento de processos pesados e de mapeamento de memória foram desenvolvidos de forma independente por questões de testes e modularidade. Além disto, com os modelos separados foi possível identificar os aspectos que eram estritamente relacionados a cada um, permitindo a geração automática de código para diferentes arquiteturas.

A integração destes dois mapeamentos culmina no objetivo principal do trabalho: geração automática de código paralelo para arquiteturas híbridas, utilizando afinidade de memória. Este capítulo descreve o desenvolvimento do mapeamento híbrido, desenvolvido com o intuito de permitir ao programador que utilize melhor os recursos disponíveis em *clusters* de máquinas NUMA. Neste sentido, os modelos de mapeamento de processos pesados (Capítulo 4) e de mapeamento de memória (Capítulo 5) foram de extrema importância.

Este capítulo apresenta a integração entre os mapeamentos de processos pesados e de memória, bem como um exemplo de utilização.

### 6.1 Visão geral do mapeamento híbrido

Conforme descrito anteriormente, arquiteturas do tipo *cluster* de máquinas NUMA possuem dois níveis de paralelismo: inter e intranodo. Para o paralelismo internodo, o mapeamento híbrido permite que o usuário crie o modelo do programa paralelo através da definição de um grafo dirigido. Neste grafo, os nodos representam processos (geralmente mapeados para diferentes máquinas do *cluster*) e os arcos representam as comunicações entre eles. O mapeamento de memória, por sua vez, permite que o usuário aplique políticas de afinidade de memória no nível intranodo. Neste contexto, o usuário define a política, quantidade de *threads* e quais estruturas devem ser alocadas e o modelo de mapeamento gera o código final. A Figura 6.1 apresenta os dois mapeamentos lado a lado.

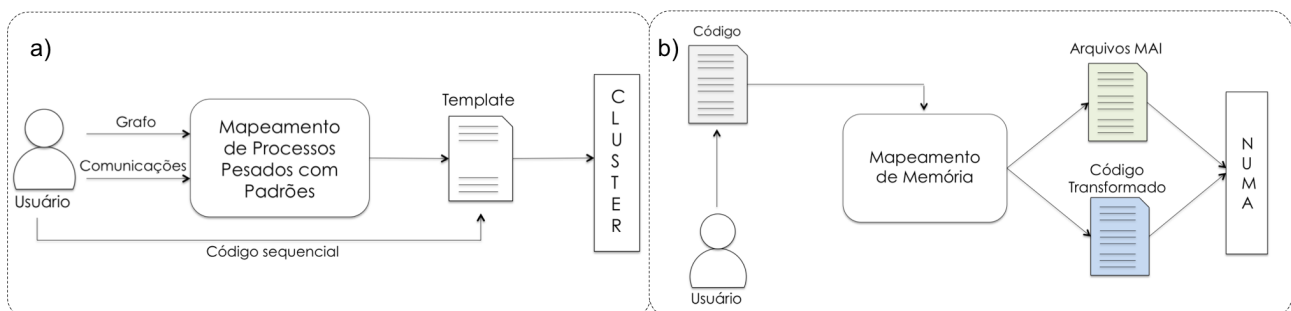


Figura 6.1 – Tipos de mapeamento: (a) Processos Pesados; (b) Mapeamento de Memória.

Como pode ser visto, o mapeamento de processos pesados (Figura 6.1a) gera arquivos *template*, que o usuário deve completar com seu código sequencial. Estes códigos, então, poderão ser executados em um cluster. O mapeamento de memória (Figura 6.1b) permite que um usuário aplique uma determinada política de afinidade de memória no seu código sequencial, para que ele tenha um desempenho mais satisfatório. O código com as políticas aplicadas está pronto para ser executado em uma máquina com arquitetura NUMA.

Contudo, as máquinas de um cluster podem possuir arquitetura NUMA. Neste cenário, os dois mapeamentos separadamente não conseguem atingir todas as funcionalidades da arquitetura disponível. A criação do modelo de mapeamento híbrido objetiva explorar o desenvolvimento de aplicações híbridas, que utilizem os dois níveis de paralelismo, e baseou-se na integração entre dos dois mapeamentos supracitados. Assim sendo, para a programação do nível internodo, utiliza-se um grafo dirigido. Entretanto, como cada nodo deste grafo representa um processo que pode ser associado a uma máquina do tipo NUMA, em cada nodo é permitido ao usuário definir a política de memória que deseja aplicar. A Figura 6.2 apresenta um diagrama ilustrando a visão geral do mapeamento híbrido.

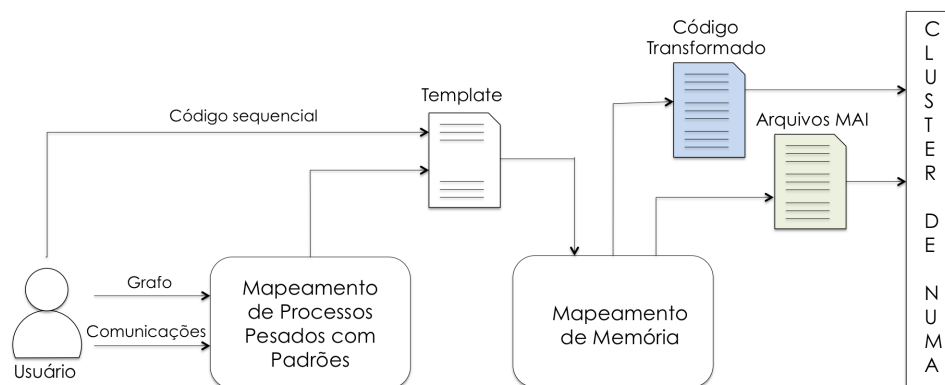


Figura 6.2 – Visão geral do mapeamento híbrido.

Para integrar os mapeamentos, foi necessário definir uma forma de “comunicação” entre a parte responsável pelo desenvolvimento intranodo com aquela responsável pelo desenvolvimento internodo. Isto se deve ao fato de que, neste mapeamento, cada nodo do grafo deixa de ser uma máquina comum e passa a ser uma máquina NUMA.

De acordo com a Figura 6.2, tem-se que o usuário interage normalmente com o mapeamento de processos pesados com padrões, indicando o grafo da aplicação e as configurações de comunicação. Em seguida, é gerado um arquivo *template*. Este arquivo *template* possui métodos que devem ser preenchidos com código sequencial do usuário.

Neste momento, entra o mapeamento de memória. O usuário então, deve interagir novamente, desta vez informando tudo o que é necessário para que as estruturas sejam alocadas corretamente. Assim sendo, o usuário deve definir a política de afinidade de memória a ser utilizada, a quantidade de nodos NUMA e a quantidade de *threads*. Com isto, é gerado o código transformado, que serve tanto para comunicação entre as máqui-



nas do *cluster* quanto para a comunicação interna de cada nodo, caracterizando, assim, a aplicação híbrida.

Cabe ressaltar que todos os aspectos detalhados no Capítulo 5 sobre a geração automática de código para máquinas NUMA, bem como todas as funcionalidades e os padrões de programação (*skeletons*) descritos no Capítulo 4 são utilizados da mesma maneira neste mapeamento. A integração dos modelos, no entanto, exige algumas mudanças na forma com que o usuário interage com o mapeamento. A Seção 6.2 apresenta um exemplo de utilização híbrido.

## 6.2 Exemplo simples de utilização

Para utilizar o mapeamento híbrido, conforme descrito anteriormente, deve-se informar um grafo para a aplicação paralela. Considere uma simples aplicação que apenas envia um *array* unidimensional para outra máquina, que processa o *array* e retorna um resultado. Será utilizado, para este exemplo, o padrão de programação *Master/Slave*. A Figura 6.3 apresenta um possível grafo para esta aplicação.

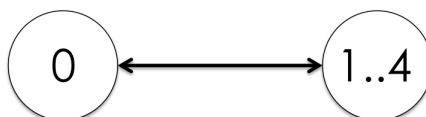


Figura 6.3 – Exemplo de um grafo com 5 nodos.

No exemplo apresentado, existe a figura de um mestre que deve comunicar-se com 4 escravos. Após este passo, deve ser definido o tipo da comunicação intranodo. Conforme supracitado, será um *array* unidimensional, e será definido que ele é do tipo *double*. Com isto, o *template* gerado para a aplicação teria um formato semelhante ao ilustrado na Figura 6.4.

Os passos realizados até então utilizam o mapeamento de processos pesados com padrões. Quando o usuário informar o código a ser incluído no *template*, ele poderá informar uma política de alocação de memória para os dados a serem alocados. Assim, supondo a escolha da política *bind\_block*, um exemplo de código transformado automaticamente pelo mapeamento de memória pode ser visto na Figura 6.5.

É importante notar que não deve ser utilizado *malloc* ou qualquer outra forma de alocação dos *arrays*, pois não é garantido que as páginas serão *untouched*. Por isto, as chamadas a *allocate\_structures* são realizadas, pois alocarão a memória com as funções específicas da MAI.

```

void Process0::beforeCommunication_1(double* (&out_1), int (&sizeOut_1), int round){

    sizeOut_1 = 100;
    out_1 = (double*) malloc(sizeof(double)*sizeOut_1);

}

void Process0::afterCommunication_1(double* out_1, int sizeOut_1, int round){

}

```

Figura 6.4 – *Template gerado pelo mapeamento de processos pesados com padrões.*

O mesmo procedimento deve ser realizado para o outro nodo do grafo: incluir o código sequencial e escolher as configurações de memória desejadas. Ao final, os arquivos de cada nodo serão mapeados para um processo MPI (controlado pela classe *Control*), e nos nodos do *array* serão alocados utilizando a política *bind\_block*.

```

void Process0::beforeCommunication_1(double* (&out_1), int (&sizeOut_1), int round){

    sizeOut_1 = 100;
    out_1 = allocate_structures(out_1, sizeOut_1, BIND_BLOCK);

}

void Process0::afterCommunication_1(double* out_1, int sizeOut_1, int round){

}

```

Figura 6.5 – *Template gerado pelo mapeamento de memória.*

### 6.3 Considerações finais

Analisando os dois exemplos apresentados na Seção 6.2, é possível notar que a afinidade de memória não precisa obrigatoriamente ser utilizada, sendo possível escolher apenas o mapeamento entre os processos pesados, por exemplo.

Da mesma forma, é possível a criação de um único nodo no grafo de entrada. Desta forma, haverá apenas uma máquina no modelo, e é possível utilizar apenas o mapeamento de memória para máquinas NUMA, escolhendo a política de afinidade desejada.

Estas são decisões do usuário, cabendo a ele escolher o que necessita utilizar da ferramenta. A forma de utilização de cada um dos mapeamentos através de uma interface gráfica (desenvolvida no presente estudo) é detalhada no Capítulo 7.

## 7. PROTÓTIPO DESENVOLVIDO

Os mapeamentos definidos nos Capítulos 4, 5 e 6 apresentam um modelo no qual o usuário deve informar algumas informações para parametrizá-los. Neste sentido, foi desenvolvido um protótipo que apresenta um modelo visual para o desenvolvimento de programas paralelos, permitindo ao usuário interagir com os modelos de mapeamentos desenvolvidos. A ideia central desta ferramenta é que usuários definam seus modelos de programação paralela através de uma interface gráfica. Programadores que já possuem suas implementações podem adaptá-las diretamente na ferramenta, que gera uma versão paralela de maneira totalmente transparente para o usuário. As linguagens de programação aceitas na ferramenta são C++ e C.

O protótipo foi implementado utilizando a linguagem de programação Java. O código gerado para a aplicação paralela final, no entanto, é C++. Para o paradigma de troca de mensagens, é utilizada a versão MPICH-2 do MPI. A escolha do MPI para a implementação da comunicação paralela deu-se, principalmente, pelo fato de que esta ferramenta é amplamente utilizada pela comunidade científica, além de possuir uma grande quantidade de referências e implementações disponíveis. Outro ponto importante na decisão pela escolha do MPI refere-se à familiaridade e experiência do autor com a biblioteca em questão.

As próximas seções descrevem maiores detalhes sobre a interface gráfica, sobre a criação do código fonte da aplicação paralela, sobre os modelos disponíveis na ferramenta e demais funcionalidades.

### 7.1 Interface gráfica com o usuário

A tela inicial da ferramenta é apresentada na Figura 7.1. Nela, podem ser percebidas três áreas distintas: a primeira delas trata-se da *Action Buttons* (área dos botões), *Model Configuration* (área das configurações) e *Model Design* (área de desenho).

A *Model Design* é onde o desenho dos modelos (grafo) será apresentado ao usuário. Na *Action Buttons*, o usuário pode escolher dentre as opções apresentadas. As primeiras opções na área dos botões referem-se aos modelos previamente implementados pela ferramenta, enquanto as demais opções são utilizadas na criação do modelo *Generic*. Ao clicar em uma destas opções, o usuário (com a utilização do mouse) desenha o grafo da aplicação correspondente na *Model Design*. A Figura 7.3 apresenta um grafo simples de exemplo. O “quadrado preto” próximo de um nodo, chamado de *incoming*, indica uma mensagem sendo recebida por aquele nodo.

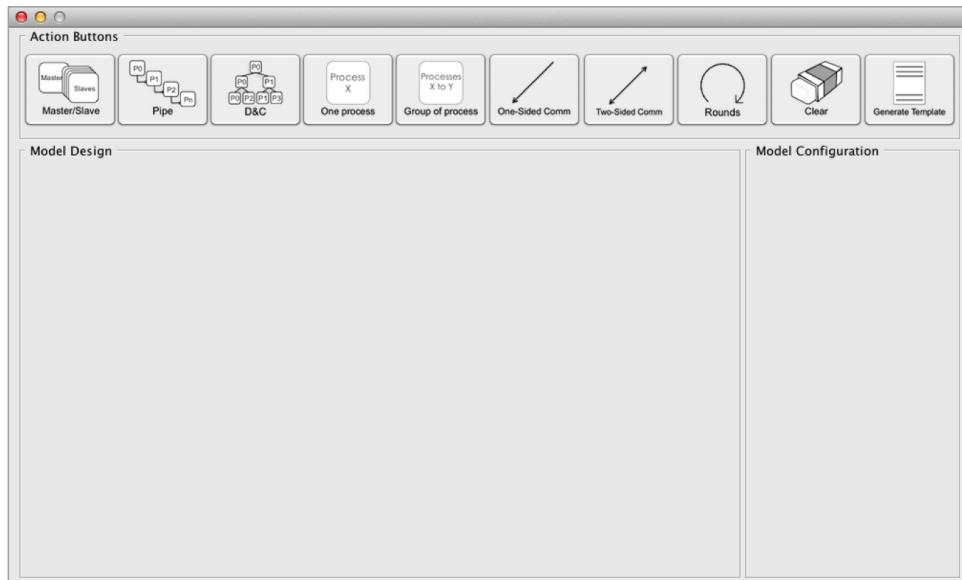


Figura 7.1 – Tela inicial do protótipo desenvolvido.

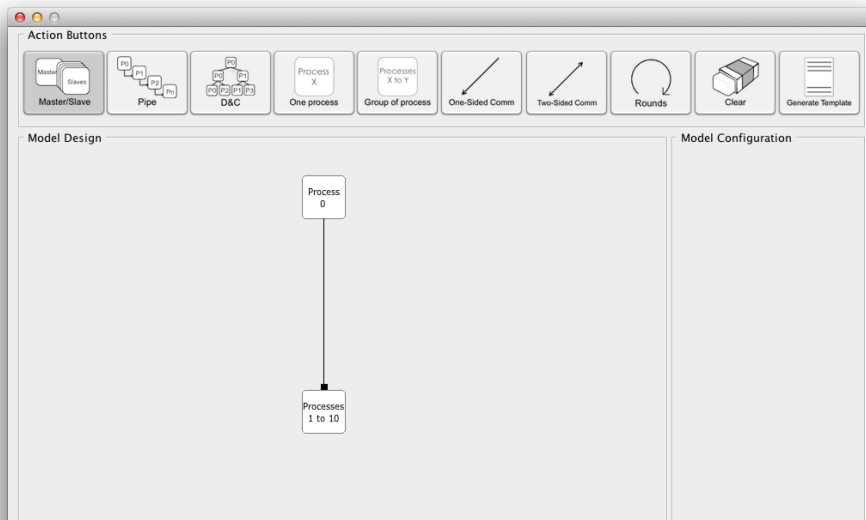


Figura 7.2 – Exemplo de grafo no protótipo desenvolvido.

Para desenhar os grafos, o usuário pode selecionar o padrão paralelo desejado e clicar na *Model Design*. A quantidade de processos é solicitada e, então, o grafo é automaticamente desenhado, de acordo com o padrão escolhido.

Caso o usuário deseje criar o seu próprio grafo (ou seja, utilizar um modelo genérico), basta selecionar na ferramenta os botões *One Process* ou *Group of Process*. Quando selecionados, um clique na *Model Design* desenhará o nodo, sempre incrementando o *rank* dos processos de maneira sequencial. Depois disto, é possível inserir os arcos entre os processos, indicando o fluxo de comunicações a serem realizadas. O botão *One Sided Comm* indica uma comunicação unilateral, enquanto o botão *Two Sided Comm* indica uma comunicação bilateral.

A última parte da interface, a *Model Configuration*, é onde o usuário vai inserir os parâmetros das comunicações do seu modelo. Nesta área, o usuário pode alterar a quantidade de dados enviados, o tipo do dado enviado e a maneira na qual os dados são divididos entre os nodos (dependendo do tipo de dado).

Quando o usuário clica duas vezes em algum *incoming*, a área de configuração da comunicação é alterada. Assim, uma configuração diferente é definida para cada comunicação do modelo. As configurações possíveis são de acordo com o descrito no Capítulo 4.

Como cada comunicação terá uma configuração diferente, o usuário deve selecionar um botão chamado *Save Configuration*, que faz com que as configurações referentes àquela comunicação sejam armazenadas. Após a criação do grafo que representa o modelo paralelo da aplicação e da definição dos parâmetros das comunicações, está tudo pronto para que o usuário complete o modelo com o código das computações.

Nas próximas seções serão detalhados o gerador de código, a maneira pela qual o usuário pode informar seu código sequencial na ferramenta e alguns exemplos de utilização.

## 7.2 Gerador de código

Depois de o usuário definir todo o seu modelo na ferramenta, ele deve escolher a opção *Generate Template*, um botão na interface da ferramenta que gera alguns arquivos para que o usuário inclua o código sequencial da aplicação. Neste momento, o protótipo (desenvolvido em Java) criará os arquivos com extensão `.cpp` (C++) para cada nodo do grafo. O conteúdo deste arquivo é um *template*, com algumas partes a serem preenchidas pelo usuário, sempre de acordo com o que foi definido no grafo do modelo e nas configurações das comunicações.

Para a geração deste arquivo fonte (template), foi desenvolvido um gerador, utilizando a linguagem de programação Java. Assim sendo, para a geração do arquivo em questão, o gerador:

- analisa o grafo informado pelo usuário (informado na área *Model Design* da interface gráfica), que possibilita o entendimento do modelo e fluxo de comunicação desejado (ou seja, o gerador sabe qual processo se comunica com outro e a ordem desta comunicação). Em linhas gerais:
  - quando uma seta sai de um processo é indicativo de um envio, criando uma primitiva `MPI_Send`;
  - quando uma seta chega em um processo é indicativo de um recebimento, criando uma primitiva `MPI_Recv`;

- quando um arco é identificado na interface, o gerador sabe que a mesma comunicação será realizada N vezes (valor informado no arco Rounds na interface gráfica), e um loop é criado no método principal do programa para garantir esta repetição.
- analisa as configurações de envio (informadas na área Model Configuration da interface gráfica), o que permite saber os dados a serem enviados (e recebidos), bem como a quantidade e a forma de divisão destes dados entre os processos;
- cria um arquivo fonte para cada processo do grafo, com os métodos template prontos para que o usuário apenas informe seu código sequencial;
- cria um método principal, que executa as importações de bibliotecas paralelas e chamadas necessárias para a correta execução do programa.

Desta maneira, o usuário estará de posse de códigos paralelos com espaços para edição. As formas de editar o código são descritas na Seção 7.3

### 7.3 Editando o código de cada processo

De posse dos arquivos *template*, o usuário tem duas opções: implementar seu programa utilizando a IDE (*Integrated Development Environment*) desejada ou, ainda, utilizar a própria ferramenta para este fim. Para utilizar a ferramenta desenvolvida, após a geração do *template*, o usuário clica duas vezes no nodo desejado e uma nova janela abrirá. Nesta janela, deve ser incluído o código sequencial de cada processo. A Figura 7.3 ilustra um exemplo de janela de edição para um nodo qualquer.

Nesta parte da ferramenta, o usuário pode visualizar o *template* criado para cada processo, editando e visualizando seu código. Além disto, pode-se perceber que esta janela permite ao usuário que ele crie aplicações híbridas, definindo políticas de afinidade de memória para um processo que executará em uma máquina NUMA. É possível, ainda, definir o número de *threads* e as estruturas alvo da política.

Conforme previamente descrito, para criar aplicações híbridas, as aplicações de entrada devem utilizar o padrão OpenMP com C ou C++. Visando ampliar a quantidade de aplicações que podem ser utilizadas na ferramenta, além de prover uma funcionalidade importante a mais para os usuários, foi incorporada no protótipo a ferramenta CETUS [DBM+09]. CETUS é utilizada, neste contexto, para a paralelização automática de laços, através da inserção de *pragmas* OpenMP e CETUS no código, criando uma região paralela, através do botão *Generate Omp Calls*. Entretanto, cabe ressaltar que a paralelização automática de laços não é foco deste trabalho, tratando-se apenas de um ponto adicional na implementação.

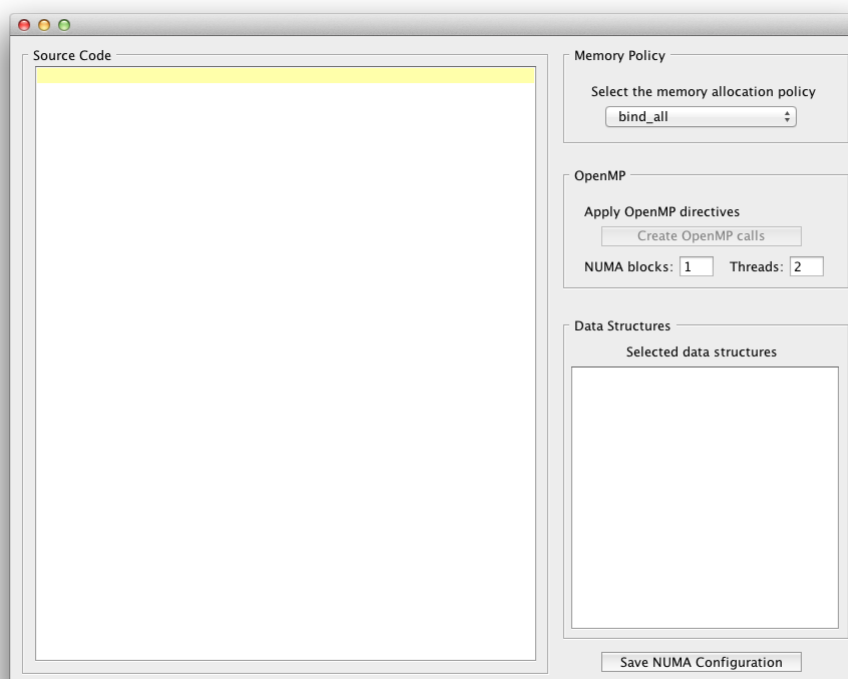


Figura 7.3 – Janela de edição de código de cada processo.

Após a inclusão de seus códigos, a ferramenta salva os arquivos automaticamente quando a janela de edição de código de cada processo é fechada ou quando o botão *Save NUMA Configuration* é selecionado. A aplicação paralela vai sendo criada na medida que as informações vão sendo inseridas nos nodos do grafo. Depois disto, o programa paralelo, então, está pronto para ser compilado e executado.

## 7.4 Exemplos de utilização

Todos os grafos criados são ligados ao modelo de mapeamento de processos pesados, enquanto todas as definições de códigos de cada nodo do grafo são ligadas ao modelo de mapeamento de memória. De fato, não é necessário criar sempre um grafo com mais de um nodo, pois algumas aplicações não necessitam necessariamente de comunicação. Logo, podem ser criadas aplicações apenas para máquinas NUMA, aplicações para *clusters* sem máquinas NUMA e aplicações para arquiteturas híbridas e com afinidade de memória.

Um exemplo tradicionalmente utilizado quando se aprende a desenvolver aplicações paralelas é o *HelloWorld*. No modelo de mapeamento de processos pesados (utilizado pelo protótipo), este simples programa pode ser descrito como um grafo de um só nodo. Este nodo, no entanto, representará mais de um nodo, e será criado a partir do bo-

tão *Group of Process*, definindo a quantidade de processos a serem utilizados. A Figura 7.4 mostra o grafo correspondente à execução da aplicação *HelloWorld* com 10 processos, juntamente com o código necessário para sua execução.

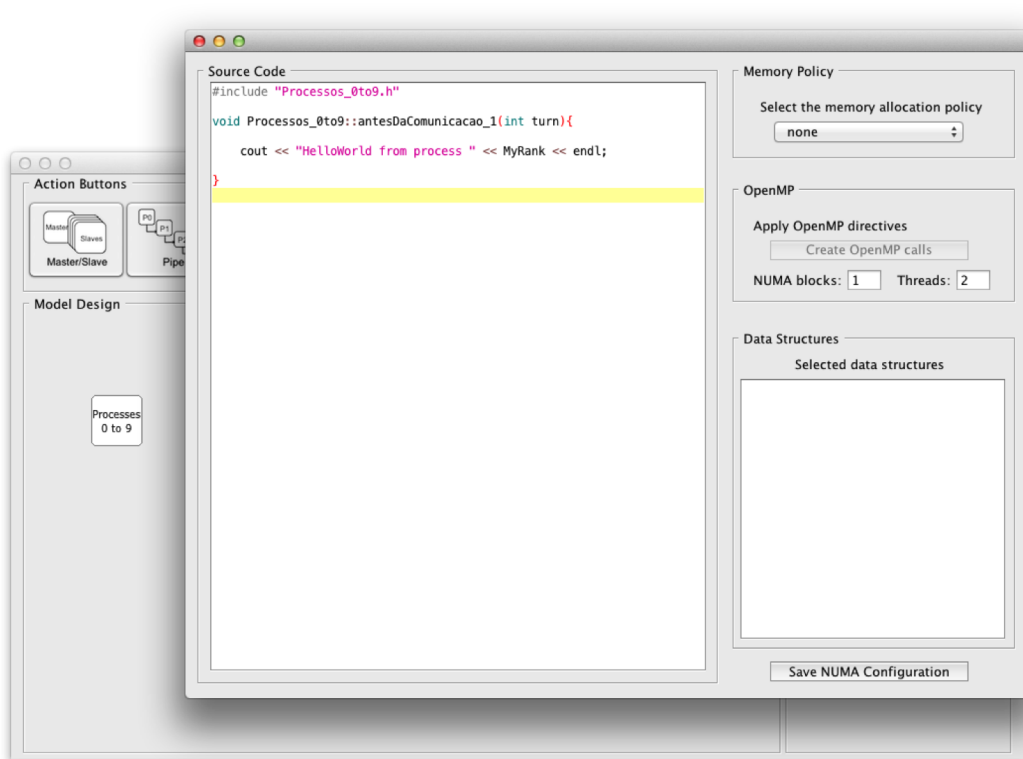


Figura 7.4 – Exemplo de HelloWorld.

No exemplo da aplicação *HelloWorld*, percebe-se que não existe comunicação entre os processos, logo, nenhum *incoming* é criado, e nenhuma área de configuração aparecerá (Model Configuration). Aplicações apenas para máquinas NUMA podem ser criadas seguindo a mesma lógica.

Seguindo o descrito, qualquer modelo pode ser criado com o protótipo, utilizando os botões de criação de um processo ou de um grupo de processos. A Figura 7.5a mostra um exemplo de um grafo para uma aplicação com uma topologia de comunicação em forma de anel, enquanto a Figura 7.5b apresenta um grafo de uma aplicação que utiliza 7 nodos, sendo que apenas um deles (*Process 0*) envia dados a serem computados.

No momento da execução, a classe de controle do mapeamento de processos pesados é a responsável pelo controle destes mapeamentos.



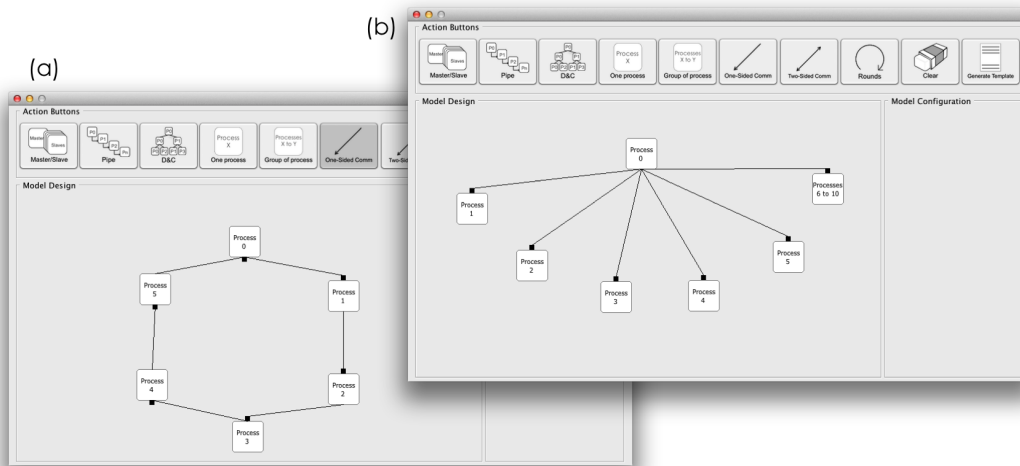


Figura 7.5 – Exemplo de grafos de diferentes aplicações.

Caso nenhuma política de alocação de memória seja escolhida, a aplicação gerada poderia ser utilizada em qualquer tipo de arquitetura, sem afinidade de memória.



## 8. EXPERIMENTOS E RESULTADOS OBTIDOS

Este capítulo apresenta alguns dos experimentos realizados para a avaliação do modelo proposto, além de uma análise sobre os resultados obtidos.

O capítulo está estruturado como segue: a Seção 8.1 apresenta o ambiente de testes utilizado; a Seção 8.2 descreve a metodologia empregada para a realização dos testes, e; a Seção 8.3 define as aplicações implementadas e os resultados obtidos.

### 8.1 Ambientes de teste

O ambiente utilizado para testes trata-se de um cluster com 16 Blades Dell PowerEdge M610. As máquinas são compostas por dois processadores Intel Xeon Six-Core E5645 2.4GHz Hyper-Threading. Cada um dos processadores possui 24GB de memória RAM e 6 núcleos, totalizando 12 núcleos em cada nó (24 threads devido ao Hyper-Threading) e 192 núcleos no cluster. Os nós estão interligados por 2 redes Gigabit-Ethernet chaveadas e 2 redes InfiniBand. Cada máquina deste cluster possui 2 nodos NUMA, que foram utilizados para realizar os testes de afinidade de memória do modelo desenvolvido neste estudo. O Sistema Operacional em execução no ambiente é Linux Ubuntu, versão 10.04. Para o armazenamento dos dados das execuções dos usuários, é utilizado o sistema de arquivos NFS (Network File System).

### 8.2 Metodologia

Os testes foram organizados para avaliar os três tipos de mapeamento oferecidos pelo modelo desenvolvido: processos pesados com padrões, memória e híbrido. As principais etapas da metodologia empregada para os testes são listadas a seguir:

- primeiramente, foram escolhidas algumas aplicações específicas para cada mapeamento, que visam abranger os principais aspectos do modelo desenvolvido;
- as aplicações escolhidas foram implementadas sequencialmente, para que comparações de desempenho pudessem ser realizadas;
- as aplicações escolhidas foram implementadas utilizando o respectivo modelo de mapeamento;
- de acordo com o mapeamento, algumas implementações adicionais foram realizadas:

- **Mapeamento de Processos Pesados com Padrões:** foram implementadas aplicações para cada padrão de programação disponível, mostrando o envio de diferentes estruturas de dados, de diferentes tipos de dados, com diferentes configurações de divisão dos dados (por tarefas, divisão igualitária entre os processos, utilizando redundância etc.). Além disto, as aplicações deste mapeamento também foram desenvolvidas sem a utilização do modelo (i.e., manualmente);
  - **Mapeamento de Memória:** as aplicações escolhidas para este tipo de mapeamento utilizam *arrays* unidimensionais e *arrays* bidimensionais, possibilitando a utilização das políticas de afinidade de memória. Além disto, as aplicações implementadas utilizam OpenMP para o paralelismo de laços, e testes com diferentes quantidades de *threads* foram realizados;
  - **Mapeamento Híbrido:** as aplicações para este mapeamento englobam as funcionalidades dos dois mapeamentos anteriores. Assim sendo, as aplicações escolhidas utilizam algum padrão de programação paralela, utilizam *threads* e foram testadas com diferentes configurações. Para as implementações, foi desenvolvida uma aplicação híbrida sem afinidade de memória, para comparar com os resultados obtidos com a política aplicada pelo modelo.
- os resultados foram tabulados e avaliados de acordo com a proposta de cada mapeamento;
  - uma análise estatística foi realizada nos resultados obtidos, com o objetivo de mostrar que os resultados com a geração de código automática proposta no presente trabalho não apresentam diferenças significativas daqueles obtidos manualmente (ou seja, sem a utilização do modelo proposto).

Estes passos possibilitam avaliações de desempenho de aplicações criadas com o modelo desenvolvido (tempo de execução e *speedup*), bem como avaliar as funcionalidades providas pelo modelo. Além disto, os tempos obtidos através das implementações com e sem o modelo (no caso do Mapeamento de Processos Pesados com Padrões), possibilitam analisar a diferença entre estas abordagens, apresentando um indicativo de quão intrusiva é a geração automática de código criada.

A análise estatística utilizada foi o teste *t de Student* [SSS13]. Este teste é utilizado para avaliar se as médias de duas amostras são ou não significativamente diferentes. As hipóteses levantadas pelo teste *t de Student* foram as seguintes:

$H_0$ :  $\text{médiaAutomático} == \text{médiaManual}$

$H_A$ :  $\text{médiaAutomático} != \text{médiaManual}$

Em  $H_0$ , a hipótese é de que as médias são iguais, sem diferenças significativas. Em  $H_A$  (Hipótese Alternativa), a hipótese é que os valores obtidos são estatisticamente

diferentes. Os testes realizados foram bi-caudais (pois deseja-se saber se os valores são significativamente diferentes ou não). O nível de confiança aplicado nos testes foi de 95%. Nestas condições, o teste *t de Student* retorna um valor chamado *p-value*: caso *p-value* seja menor do que 0,05, tem-se um resultado a favor de  $H_A$ , ou seja, os valores são diferentes; caso contrário, aceita-se  $H_0$  e não existe diferença significativa nos valores obtidos. Para maiores detalhes sobre o teste *t de Student*, sugere-se a leitura de [SSS13].

Os dados foram analisados utilizando o ambiente R [HE14], que se trata de uma ferramenta específica para a realização de testes estatísticos. Para tanto, utilizou-se a seguinte sintaxe na ferramenta:

```
dadosAutomaticos = c(dado1, dado2, dado3, ..., dadoN)
dadosManuais = c(dado1, dado2, dado3, ..., dadoN)
t.test(dadosAutomaticos, dadosManuais, alternative='two.sided')
```

`dadosAutomaticos` refere-se ao conjunto de resultados coletados com a utilização do modelo proposto, enquanto `dadosManuais` referem-se aos resultados coletados com o desenvolvimento manual das aplicações. Em seguida, o comando `t.test` foi executado com estes dados como parâmetro, e com a indicação do tipo de teste a ser realizado (`two.sided`, ou bi-caudal). A intenção destas análises estatísticas é mostrar se houve ou não perda significativa de desempenho nas aplicações geradas automaticamente através do modelo proposto neste trabalho.

Algumas das aplicações utilizadas para testes foram implementadas pelo autor, enquanto outras referem-se a implementações previamente realizadas por outros autores, objetivando demonstrar que o modelo possibilita a adaptação de códigos já existentes. Os valores apresentados foram obtidos através da média de 30 execuções de cada versão. As aplicações e a avaliação de cada mapeamento são contempladas na Seção 8.3.

## 8.3 Avaliação dos resultados

No decorrer desta seção, são apresentados os resultados obtidos a partir das implementações realizadas, separadas por tipo de mapeamento.

### 8.3.1 Processos pesados

Para cada padrão de programação paralela disponível foram implementadas algumas aplicações específicas, que serão descritas no decorrer das próximas subseções.

## Master/Slave

Para o padrão *Master/Slave*, quatro aplicações foram implementadas: o problema das  $N$  – Rainhas ( $N$  – *Queens*), a contagem sequencial de elementos em um *array* de inteiros (*CountElements*), a busca de um determinado sub-texto em um texto (*FindText*) e a multiplicação de matrizes (*MatrixMult*).

### N-Queens

O problema das N-Rainhas, ou *N-Queens*, objetiva encontrar todas as combinações de N rainhas em um tabuleiro com dimensões NxN, sendo que nenhuma destas rainhas pode conseguir atacar a outra, de acordo com as regras do jogo de xadrez. A implementação original foi obtida em [Rol08]. Assim sendo, a aplicação foi inserida no contexto dos mapeamentos, sem alterações na lógica.

A Figura 8.1 apresenta o gráfico obtido com as execuções da aplicação *N – Queens*. As curvas indicam os tempos obtidos com uma implementação utilizando o mapeamento de processos pesados com o padrão *Master/Slave* (geração automática do código), comparados com aqueles obtidos com a implementação da aplicação de forma manual. Os resultados apresentados são para uma implementação com um  $N = 16$ .

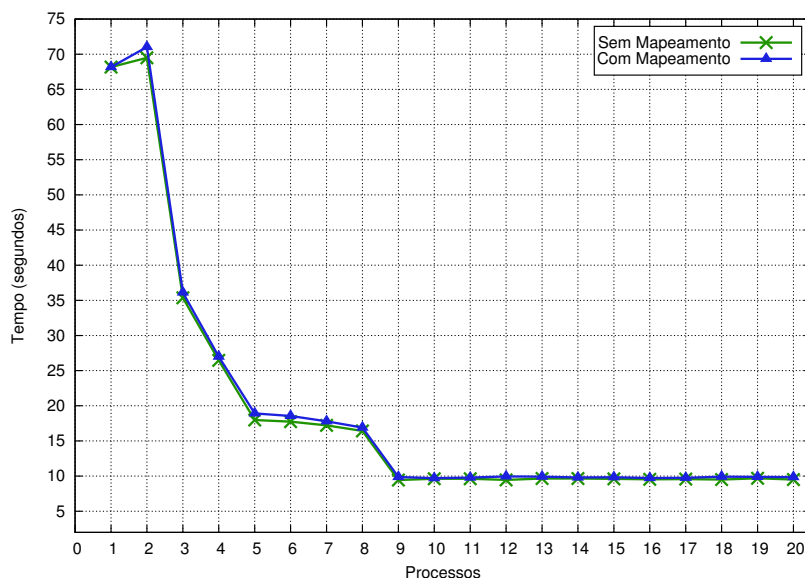


Figura 8.1 – *N – Queens* (16x16)

O tempo sequencial da aplicação *N – Queens* para um N com valor 16 foi de 68,1535 segundos. A Tabela 8.1 lista os tempos correspondentes ao gráfico da Figura 8.1, acompanhados de seus respectivos desvios padrão. Através dos tempos, é possível perceber que a aplicação gerada automaticamente apresentou um desempenho menor do que aquela realizada manualmente. Este comportamento era o esperado, uma vez que não

se desejava obter um desempenho maior do que implementações tradicionais, em virtude da abstração acrescentada na solução.

Tabela 8.1 – Tempos obtidos com a aplicação *N – Queens* para  $N=16$

Processos	Manual		Automático	
	Média	Desvio Padrão	Média	Desvio Padrão
2	69,4804	1,9068	71,0168	0,4154
3	35,3499	0,8233	36,1096	0,2493
4	26,4627	0,5538	26,9879	0,3293
5	17,9771	0,1702	18,9134	0,4584
6	17,7438	0,1099	18,5483	0,4458
7	17,2174	0,3789	17,7921	0,2842
8	16,4306	0,3090	16,9066	0,2591
9	9,4492	0,2719	9,8746	0,3022
10	9,6251	0,3074	9,7097	0,2702
11	9,6233	0,2816	9,7835	0,2039
12	9,4647	0,2758	9,9536	0,3958
13	9,6655	0,3196	9,9183	0,2913
14	9,6619	0,2732	9,8015	0,1743
15	9,6110	0,2841	9,8437	0,1570
16	9,5451	0,2469	9,7224	0,1618
17	9,5712	0,2507	9,7539	0,1959
18	9,5146	0,2302	9,9186	0,1705
19	9,6951	0,4330	9,8926	0,2744
20	9,5148	0,3130	9,8603	0,1951

Os tempos obtidos com o modelo automático, no entanto, não foram muito distantes daqueles observados na aplicação tradicional. A maior diferença de tempo foi na execução com 2 processos, onde a diferença foi de 1,5364 segundos.

O resultado do teste t de Student no ambiente R são listados a seguir:

```
Welch Two Sample t-test
data: f1 and f2
t = -0.095645, df = 35.984, p-value = 0.9243
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
-10.172135 9.255925
sample estimates:
mean of x mean of y
16.61071 17.06881
```

O valor obtido para *p-value* foi 0.9243, indicando não haver diferença significativa nos tempos obtidos.

Além da implementação com o valor de  $N=16$ , outras implementações para esta aplicação foram realizadas. Os resultados obtidos são apresentados no Apêndice A.

## CountElements

A aplicação `CountElements` consiste em uma busca sequencial em um array de inteiros contando quantas vezes algum valor aparece no array. Esta aplicação foi implementada pelo autor, tanto na versão manual quanto na versão automática.

Os resultados apresentados na Figura 8.2 e na Tabela 8.2 são referentes à execução da aplicação `CountElements` em um *array* de tamanho 15000000000. Demais experimentos para esta aplicação podem ser encontrados no Apêndice B.

Nesta versão, a aplicação foi paralelizada dividindo o *array* entre os escravos. Assim, o mestre lê o *array* a ser computado e divide-o entre os escravos. Os escravos realizam a computação de suas partes e respondem ao mestre com o resultado obtido.

A aplicação implementada possui uma característica interessante: os tempos paralelos foram muito superiores ao tempo sequencial da aplicação. Enquanto a aplicação sequencial apresentou um tempo de 12,1363 segundos, o menor tempo encontrado com a paralelização foi de 58,1470 segundos com 15 processos (sem a utilização do mapeamento de processos pesados).

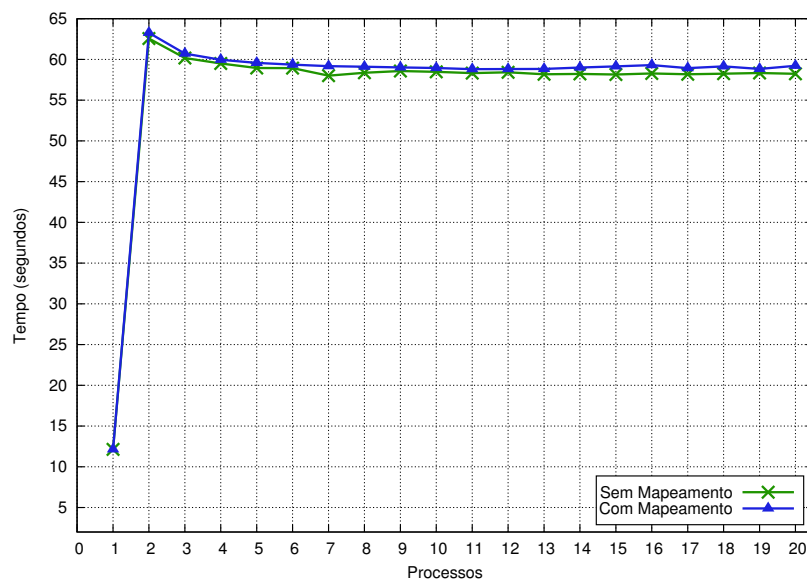


Figura 8.2 – `CountElements` para o tamanho 15000000000.

A partir dos tempos de execução (Tabela 8.2), este caso de teste mostra que a utilização da geração automática desenvolvida segue a tendência das aplicações sem o mapeamento. Assim como os resultados obtidos na aplicação *N – Queens*, os tempos obtidos com a versão automática de `CountElements` são maiores do que os obtidos com a versão manual de `CountElements`.



Tabela 8.2 – Tempos obtidos para a aplicação CountElements para o tamanho 1500000000

Processos	Manual		Automático	
	Média	Desvio Padrão	Média	Desvio Padrão
2	62,5670	0,2928	63,2589	0,0539
3	60,1713	0,2948	60,7064	0,0570
4	59,5078	0,2833	59,9560	0,1165
5	58,9534	0,2470	59,5698	0,1101
6	58,9387	0,2785	59,3583	0,1436
7	58,7126	0,2441	59,1727	0,1428
8	58,3561	0,2503	59,0980	0,1238
9	58,5742	0,3288	59,0141	0,0752
10	58,4781	0,2972	58,9405	0,0905
11	58,3208	0,2218	58,8053	0,1170
12	58,4225	0,2610	58,8209	0,1267
13	58,1809	0,2851	58,8388	0,1094
14	58,2228	0,3078	59,0083	0,4459
15	58,1470	0,2361	59,1496	0,7356
16	58,2785	0,2577	59,3076	0,8184
17	58,1793	0,4023	58,9378	0,4055
18	58,2543	0,3350	59,1481	0,6470
19	58,3356	0,1722	58,8359	0,2063
20	58,2397	0,2187	59,2231	1,3296

O resultado do teste t de Student (obtido no R) pode ser visto abaixo:

```
Welch Two Sample t-test
data: e1 and e2
t = -1.9125, df = 35.989, p-value = 0.0638
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
-1.33490392 0.03916708
sample estimates:
mean of x mean of y
58.78108 59.42895
```

O valor 0.0638 de *p-value* indica que a diferença da média dos valores não é estatisticamente significativa.

A versão descrita anteriormente divide os *arrays* entre os processos. Utilizando uma abordagem sem o envio dos *arrays*, foram obtidos os resultados apresentados na Tabela 8.3.

A saída abaixo foi obtida com a utilização do ambiente R para o teste t de Student dos valores supracitados.

Tabela 8.3 – Tempos obtidos para `CountElements` com tamanho 15000000000 sem envio do array pelo processo *Master*

Processos	Manual		Automático	
	Média	Desvio Padrão	Média	Desvio Padrão
2	12,8693	0,1605	12,7759	0,0777
3	10,3565	0,2198	10,5257	0,4033
4	9,6960	0,1412	9,9479	0,3206
5	9,3216	0,1202	9,5781	0,1021
6	9,1502	0,1158	9,2790	0,0842
7	9,0216	0,1391	9,1945	0,0767
8	8,9725	0,2090	9,0802	0,0784
9	8,8716	0,0994	9,0295	0,3249
10	8,8109	0,1039	8,9409	0,0525
11	8,7794	0,1240	8,8688	0,0736
12	8,7997	0,1207	8,8436	0,0643
13	9,6546	0,0702	8,9387	0,2753
14	9,7683	0,1170	8,7686	0,0526
15	9,9398	0,0876	8,8376	0,3033
16	9,9033	0,1546	8,8184	0,3138
17	9,8798	0,1109	8,6995	0,0398
18	9,8019	0,1431	8,8101	0,3219
19	9,8294	0,0848	8,6353	0,0654
20	9,7588	0,1116	8,6642	0,2601

Bem como nos testes anteriores, as diferenças entre os tempos obtidos não são significativas (de acordo com o valor 0.241 obtido para *p-value*):

```
Welch Two Sample t-test
data:  d1 and d2
t = 1.1923, df = 35.877, p-value = 0.241
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
-0.2564631 0.9879052
sample estimates:
mean of x mean of y
9.641326 9.275605
```

Nesta versão, todos os escravos realizam a leitura do *array*, e utilizam a sua informação de *rank* para realizar o processamento. Ao final, cada escravo informa ao mestre suas respostas, que as agrupa e finaliza a computação. As diferenças entre os tempos obtidos podem ser melhor visualizadas no gráfico da Figura 8.3.

Nos tempos obtidos, percebe-se que a diferença entre as versões foi pequena, e em alguns momentos até mesmo a versão automática apresentou um desempenho melhor do que a versão manual. Esta diferença pode ser explicada por algum detalhe na implementação da versão manual, que pode ter algum controle adicional desnecessário. Nota-se, no

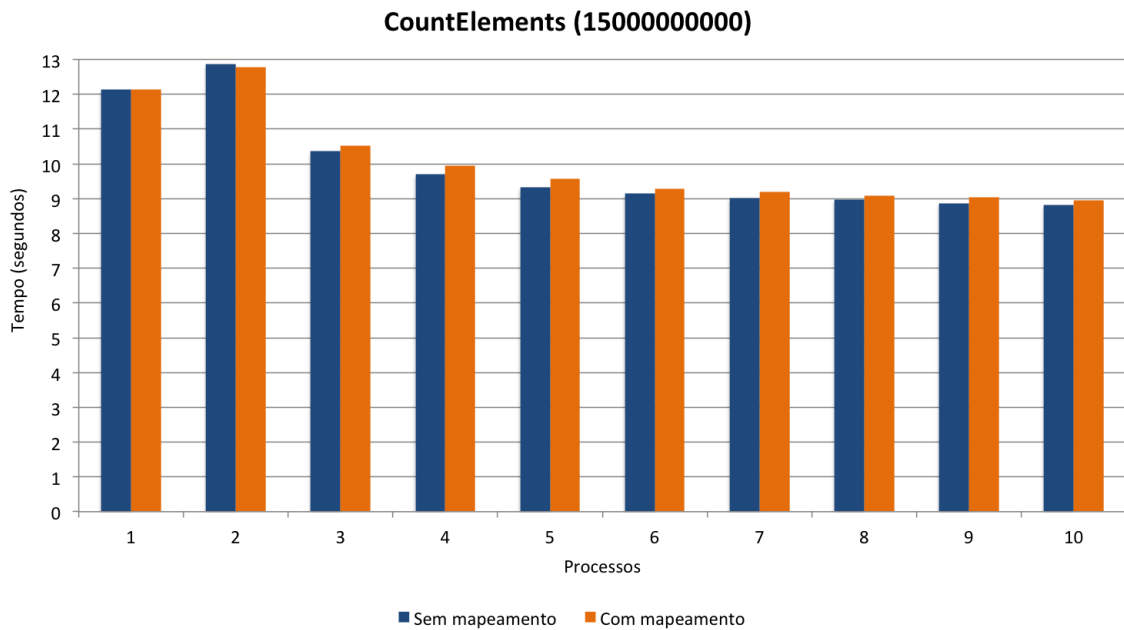


Figura 8.3 – Resultado para a aplicação `CountElements` para o tamanho 1500000000 sem envio do array pelo processo *Master*

entanto, que a diferença é sempre muito pequena, principalmente quando levado em conta o valor do desvio padrão.

## FindText

A aplicação `FindText` foi desenvolvida pelo autor, e consiste na busca de textos específicos dentro de outros textos maiores. Este tipo de aplicação também é conhecido como *StringMatching*.

No entanto, esta aplicação utiliza duas características importantes do modelo desenvolvido: primeiramente, envia o tipo de dados `std::string`; segundo, utiliza a funcionalidade de redundância de dados. A redundância acontece pois, neste caso, a simples divisão do texto entre os *Slaves* não vai ser suficiente para a aplicação funcionar, uma vez que a palavra procurada pode ser dividida, e uma parte seja atribuída a um processo e outra parte a outro processo. Assim sendo, nenhum dos dois processos vai encontrar a palavra, ocasionando um erro no resultado final.

Os resultados da aplicação `FindText` acompanham os resultados anteriores, e auxiliam na visualização do comportamento das aplicações geradas com o mapeamento desenvolvido. A Figura 8.4 apresenta um gráfico comparativo dos tempos obtidos para esta aplicação, de acordo com a quantidade de processos, para uma quantidade de 1000000000 textos de 20 caracteres cada.

É possível ver que o comportamento das duas versões é bastante similar, e que os tempos apresentados pela geração manual são, no geral, melhores do que os da aplicação

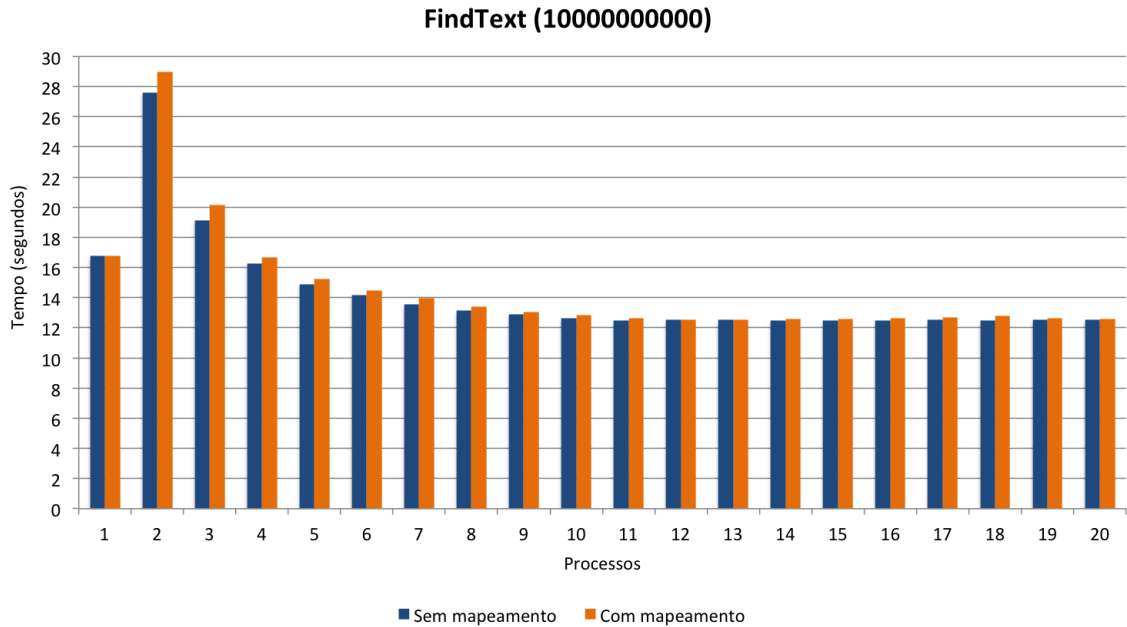


Figura 8.4 – Resultados da aplicação FindText para 1000000000 textos

automática. Esta diferença pode ser percebida, principalmente, quando os tempos são um pouco maiores, sendo a maior diferença com 2 processos. O fato de a aplicação automática abstrair o cálculo da borda pode ser um dos motivos da perda de desempenho visualizada na maioria dos casos. Os tempos de execução para esta aplicação são listados na Tabela 8.4.

Tabela 8.4 – Tempos obtidos com a aplicação FindText para 1000000000 textos

Processos	Manual		Automático	
	Média	Desvio Padrão	Média	Desvio Padrão
2	27,6131	1,3357	28,9996	1,5006
3	19,1433	0,7188	20,1371	0,9770
4	16,2494	0,0745	16,6827	0,0963
5	14,8926	0,0452	15,2607	0,0793
6	14,1943	0,2655	14,4727	0,2995
7	13,5440	0,2170	13,9658	0,3590
8	13,1321	0,0402	13,4008	0,1991
9	12,8844	0,0599	13,0600	0,0769
10	12,6457	0,1629	12,8349	0,1682
11	12,4888	0,0582	12,6479	0,2010
12	12,5157	0,0906	12,5411	0,0639
13	12,5180	0,0557	12,5534	0,1037
14	12,4843	0,0517	12,5913	0,0886
15	12,4980	0,0602	12,5974	0,1239
16	12,4806	0,0483	12,6513	0,2047
17	12,5194	0,0616	12,6752	0,2193
18	12,4999	0,0520	12,7949	0,4884
19	12,5140	0,0654	12,6406	0,1933
20	12,5410	0,0551	12,6019	0,2037

O ambiente R apresentou o resultado abaixo para o teste t de Student dos valores obtidos nesta aplicação:

```
Welch Two Sample t-test
data:  c1 and c2
t = -0.24264, df = 35.741, p-value = 0.8097
alternative hypothesis:  true difference in means is not equal to 0
95 percent confidence interval:
-2.833164 2.227827
sample estimates:
mean of x mean of y
14.17677 14.47944
```

O valor de *p-value* indica que as diferenças dos tempos não são significativas para este exemplo.

## MatrixMult

A aplicação de multiplicação de matrizes foi implementada pelo autor, e foi realizada em duas versões distintas. Na primeira versão, uma das matrizes é enviada em *broadcast*, enquanto a outra é dividida entre os escravos. A Tabela 8.5 mostra os tempos obtidos para esta versão, com a multiplicação de duas matrizes de dimensões 2000x2000.

Tabela 8.5 – Tempos obtidos para a aplicação *MultMatrix* para matrizes de dimensões 2000x2000

Processos	Manual		Automático	
	Média	Desvio Padrão	Média	Desvio Padrão
2	101,5767	0,5321	116,7701	4,1378
3	51,1029	0,1955	59,0009	1,9697
4	41,0572	2,1215	43,6892	1,3221
5	31,3350	1,2905	32,6136	0,5053
6	26,3981	0,3792	27,4985	0,5393
7	22,2368	0,2944	22,8614	0,5907
8	18,7196	0,5527	20,1991	0,5496
9	16,2913	0,6702	17,5130	0,3514
10	14,6635	0,6214	15,9022	0,2064
11	13,8080	0,4300	14,7830	0,4302
12	12,4660	0,4879	13,8408	0,7277
13	11,9872	0,3269	12,7343	0,5037
14	11,1646	0,2602	11,8299	0,2291
15	11,6743	0,1864	11,1231	0,4432
16	11,1381	0,4245	10,3784	0,2015
17	10,5917	0,2548	9,9696	0,1775
18	10,0791	0,1393	9,4435	0,0742
19	9,6779	0,1246	9,4108	0,2673
20	9,6767	0,3005	10,1150	0,3838

Para os valores da Tabela 8.5, o resultado do teste estatístico realizado no ambiente R foi o seguinte:

```
Welch Two Sample t-test
data:  b1 and b2
t = -0.22883, df = 35.208, p-value = 0.8203
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
-17.67839 14.09611
sample estimates:
mean of x mean of y
22.92867 24.71981
```

Mais uma vez, as diferenças entre os tempos obtidos com a versão manual e a versão automática não são significativas (de acordo com o valor 0.8203 de *p-value*).

Esta aplicação mostra uma utilização interessante permitida pelo modelo de mapeamento desenvolvido: o envio de mais de um tipo de dado na comunicação, e cada um deles podendo ser enviado de uma maneira específica. Neste caso específico, uma matriz é enviada inteira para todos os escravos, enquanto outra matriz é quebrada entre os escravos. Percebe-se, através dos resultados, que a geração automática lida bem com este tipo de situação, apresentando resultados bem similares aos encontrados na versão manual.

Em outra versão da mesma aplicação, utiliza-se a funcionalidade de dividir os dados em um número específico de tarefas. Assim sendo, o mestre cria as tarefas determinadas pelo usuário no modelo e realiza uma espécie de saco de tarefas, recebendo as respostas de cada escravo e encaminhando uma nova tarefa caso ainda existam. O comportamento da divisão do trabalho em 10, 20, 50 e 100 tarefas é visualizado na Figura 8.5.

Pode-se perceber que com 2 processos, o tempo de execução é maior do que o sequencial. Isto ocorre pois a aplicação com 2 processos possui 1 mestre e 1 escravo, tornando a comunicação entre eles um gargalo. A partir de 3 processos, o tempo de execução reduz consideravelmente. As versões com 50 e 100 tarefas foram as que apresentaram melhores resultados. Isto ocorre, possivelmente, devido à comunicação ser menos intensa nestes casos.

## Pipe

Duas implementações foram realizadas para o padrão *Pipe*: filtro de imagens (*Filter*) e criptografia de textos (*Crypt*). Os detalhes e resultados de cada aplicação são descritos a seguir.

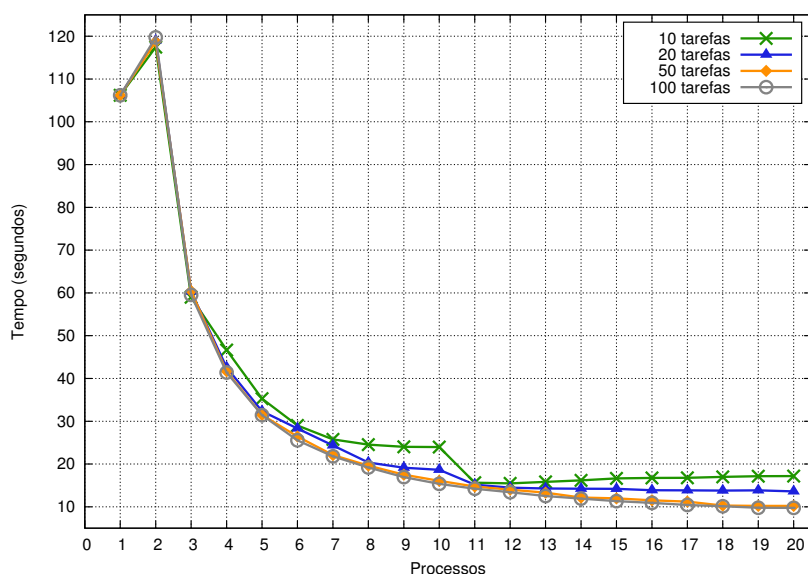


Figura 8.5 – Resultado da aplicação MultMatrix para matrizes de dimensões 2000x2000, divididas em tarefas

## Filter

A aplicação de filtro de imagem foi desenvolvida com 4 processos no Pipe, cada um realizando um filtro diferente nas imagens que chegavam. A aplicação de filtro de imagens resultou nas curvas ilustradas na Figura 8.6, que variam a quantidade de imagens sendo processadas.

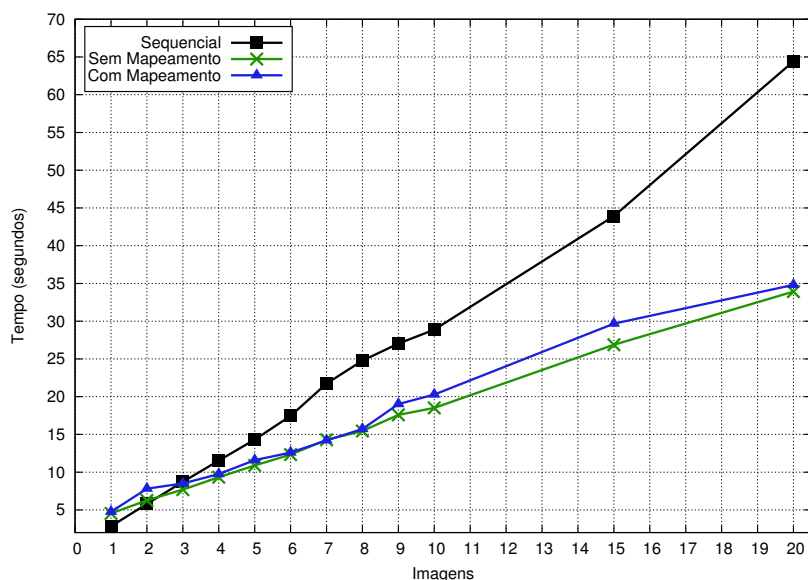


Figura 8.6 – Resultados obtidos para a aplicação de Filtro de Imagens.

Em geral, execuções com mais imagens apresentam uma variação maior entre os resultados da versão automática e os da versão manual. Entretanto, o ponto mais interessante dos tempos obtidos é o comportamento da aplicação sequencial em comparação ao comportamento das demais versões. Conforme o pipe vai sendo preenchido, as versões

paralelas vão aumentando seus tempos de execução de uma forma mais suave do que a versão sequencial. Assim, é possível perceber claramente no gráfico que o primeiro ganho de desempenho que pode ser percebido de ambas as versões sequenciais trata-se da aplicação de filtro em 4 imagens, ou seja, a mesma quantidade de processos no Pipe.

## Crypt

Para a aplicação de criptografia (chamada de *Crypt*), 4 algoritmos de criptografia foram utilizados em 6 processos. O primeiro processo do pipe era responsável apenas por ler a mensagem inicial. Para os próximos 4 processos, cada um deles realizava um algoritmo de criptografia diferente. Finalmente, o último processo recebia a mensagem e finalizava a computação. Os tempos obtidos para a criptografia de 1000000000 textos de 20 caracteres cada são ilustrados na Tabela 8.6.

Tabela 8.6 – Tempos obtidos para a aplicação *Crypt* para 1000000000 textos

Textos	Sequencial		Manual		Automático	
	Média	Des. Pad.	Média	Des. Pad.	Média	Des. Pad.
1	51,7144	0,3990	76,4127	0,2898	76,5778	0,8130
2	102,6774	0,0568	121,9141	0,2402	121,8651	0,4283
3	154,6936	1,4028	167,4281	0,2980	167,8804	1,4961
4	205,0226	0,6457	215,2508	4,1910	214,1176	3,3054
5	257,5716	0,4632	261,7411	5,2779	258,1496	0,2943
6	309,2422	0,7297	304,7639	1,0226	303,8307	0,3758
7	362,0000	0,8700	350,9640	0,9512	349,9513	0,6819
8	416,0276	2,4702	398,5038	6,7403	396,5190	1,4587
9	479,3110	3,7742	444,9390	6,8347	441,2769	0,6556
10	545,9880	2,3280	494,2248	11,0736	488,8717	1,3426
11	616,8034	1,9908	538,2792	8,6885	538,0854	9,2335
12	681,2388	4,2714	580,8401	0,9664	583,8734	9,9343
13	749,4086	2,1343	632,8762	1,1292	631,5780	0,4086
14	825,1944	7,0764	689,0984	11,4343	684,5070	1,3771
15	891,0724	5,9722	739,6012	3,9184	742,1001	12,7409
16	965,8830	6,3165	791,4861	2,8755	790,9546	2,9520
17	1034,3560	11,5518	849,3760	14,7905	852,0216	16,2414
18	1100,2120	8,5303	898,5462	3,0176	902,4198	11,5018
19	1179,5940	16,7680	962,1792	20,0289	951,0790	3,5441
20	1251,1420	14,9196	1015,5255	22,0770	1019,1833	18,4144

Novamente, os tempos desta aplicação ilustram que quando o pipe enche, a aplicação paralela começa a apresentar um melhor desempenho. No caso apresentado, a criptografia paralela até 5 textos apresenta um desempenho pior nas versões paralelas. Como a aplicação utiliza um pipe com 6 processos, a partir de 6 textos no pipe, os tempos invertem e as aplicações paralelas começam a apresentar um desempenho melhor. Para 20 textos, a diferença entre a aplicação sequencial e a aplicação paralela gerada automaticamente chega a 231,9587 segundos (sem considerar o desvio padrão).

A saída do teste t de Student pode ser vista a seguir. Nela, é possível perceber que o valor de *p-value* é 0.9917, indicando que não há diferenças significativas nos valores



obtidos:

```
Welch Two Sample t-test
data:  a1 and a2
t = 0.010426, df = 38, p-value = 0.9917
alternative hypothesis:  true difference in means is not equal to 0
95 percent confidence interval:
-184.5586 186.4694
sample estimates:
mean of x mean of y
526.6975 525.7421
```

## Divide and Conquer

Para o padrão *Divide and Conquer*, duas aplicações de ordenação (BubbleSort e QuickSort) foram implementadas. Estas aplicações foram desenvolvidas utilizando a versão estática do padrão D&C, uma vez que o ambiente de testes não estava configurado para a utilização da primitiva `Spawn` do MPI-2 no momento da realização dos testes.

As aplicações BubbleSort e QuickSort são bastante conhecidas para ordenação de arrays. Assim sendo, os resultados apresentados referem-se a uma ordenação de arrays de 1000000 posições.

A versão sequencial da aplicação BubbleSort para um *array* de tamanho 1000000 apresentou um tempo de 2383,407 segundos, com um desvio padrão de 1,6019 segundos. Os tempos das versões com e sem o mapeamento são apresentados na Tabela 8.7.

Tabela 8.7 – Tempos obtidos para a aplicação BubbleSort com array de tamanho 1000000

Condição de Parada	Manual		Automático	
	Média	Desvio Padrão	Média	Desvio Padrão
100000	3019,2890	1,7200	3026,4180	9,2157
125000	2988,9800	8,8487	2987,1130	5,9941
250000	2836,2280	4,8468	2842,6280	9,4951
500000	2268,5740	5,4021	2270,9550	6,8605

A análise estatística mostra que as diferenças não são significantes entre a versão automática e a versão manual:

```
Welch Two Sample t-test
data:  bubble1 and bubble2
t = -0.014212, df = 6, p-value = 0.9891
alternative hypothesis:  true difference in means is not equal to 0
```

95 percent confidence interval:

-607.9742 600.9527

sample estimates:

mean of x mean of y

2778.268 2781.778

A condição de parada na primeira coluna da Tabela 8.7 refere-se à finalização da divisão e começo da conquista. Assim, foram testadas as versões com paradas em 100000, 125000, 250000 e 500000 elementos. Para estas condições, a quantidade de processos participantes da árvore do padrão paralelo é:

- 100000: 16 processos;
- 125000: 6 processos;
- 250000: 4 processos;
- 500000: 2 processos.

Percebe-se que quanto maior a condição de parada, menor o tempo de execução. O tempo sequencial foi ultrapassado quando da utilização da condição de parada 500000, quando apenas 2 processos participam da comunicação. Mais uma vez, a geração automática e código mostrou-se bastante apropriada, com resultados condizentes com a implementação manual.

Nos mesmos moldes, foi implementada a aplicação QuickSort. QuickSort trata-se de um algoritmo de ordenação muito mais otimizado do que BubbleSort. Para um *array* de 1000000 posições, a versão sequencial do QuickSort ordenou o *array* em 0,1395 segundos em média, com um desvio padrão de 0,00006 segundos. A Tabela 8.8 apresenta os tempos de execução das implementações paralelas.

Tabela 8.8 – Tempos obtidos para a aplicação QuickSort com array de tamanho 1000000

Condição de Parada	Manual		Automático	
	Média	Desvio Padrão	Média	Desvio Padrão
100000	0,3231	0,0503	0,2833	0,0019
125000	0,2638	0,0041	0,2637	0,0038
250000	0,2254	0,0036	0,2238	0,0021
500000	0,1801	0,0022	0,1806	0,0015

O teste estatístico para a aplicação em questão pode ser visto abaixo:

Welch Two Sample t-test

data: quick1 and quick2

t = 0.27058, df = 5.5671, p-value = 0.7965

```

alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
-0.08421714 0.10471714
sample estimates:
mean of x mean of y
0.24810 0.23785

```

No teste, o valor de *p-value* com valor de 0.7965 mostra que não existem diferenças significativas entre os valores obtidos.

É possível perceber que a utilização deste tipo de abordagem para um *array* deste tamanho não foi satisfatória em relação à versão sequencial. Entretanto, para a análise do funcionamento do modelo proposto, os resultados seguiram as tendências da versão não automática.

## Modelo Genérico

A implementação utilizada para analisar o desempenho do modelo genérico foi retirada de [Bur17c]. A aplicação chama-se *Search*, e ela procura um valor inteiro entre *A* e *B*, tal que uma função qualquer *f* aplicada a este valor resulte em um determinado valor *C*. No exemplo executado [Bur17c], os valores são  $F(1,674,924,981) = 45$ .

A aplicação *Search* não possui comunicação entre os processos, tratando-se de um exemplo bastante interessante para avaliação. Nesta aplicação, apenas o *rank* e a quantidade de processos são necessários para a aplicação. Os resultados do gráfico da Figura 8.7 mostram que ambas aplicações comportaram-se de maneira muito similar, apresentando praticamente o mesmo desempenho e o mesmo comportamento.

Mesmo nas situações em que a aplicação se comporta de maneira diferente, como por exemplo a execução com 6 processos (onde o tempo de execução aumenta, ao invés de diminuir, que seria mais natural), a aplicação automática seguiu o comportamento.

Outras aplicações com o modelo genérico foram testadas, tais como uma topologia em anel e uma aplicação com operação com matrizes, por exemplo. No entanto, resultados quantitativos não foram mostrados, em virtude de que estas aplicações não possuíam correspondentes implementados sem o mapeamento e tampouco sequenciais.

### 8.3.2 Memória

O mapeamento de memória foi testado com três aplicações: multiplicação de matrizes (*MatrixMult*), o conjunto de *Mandelbrot* e uma versão do algoritmo de *Dijkstra*.

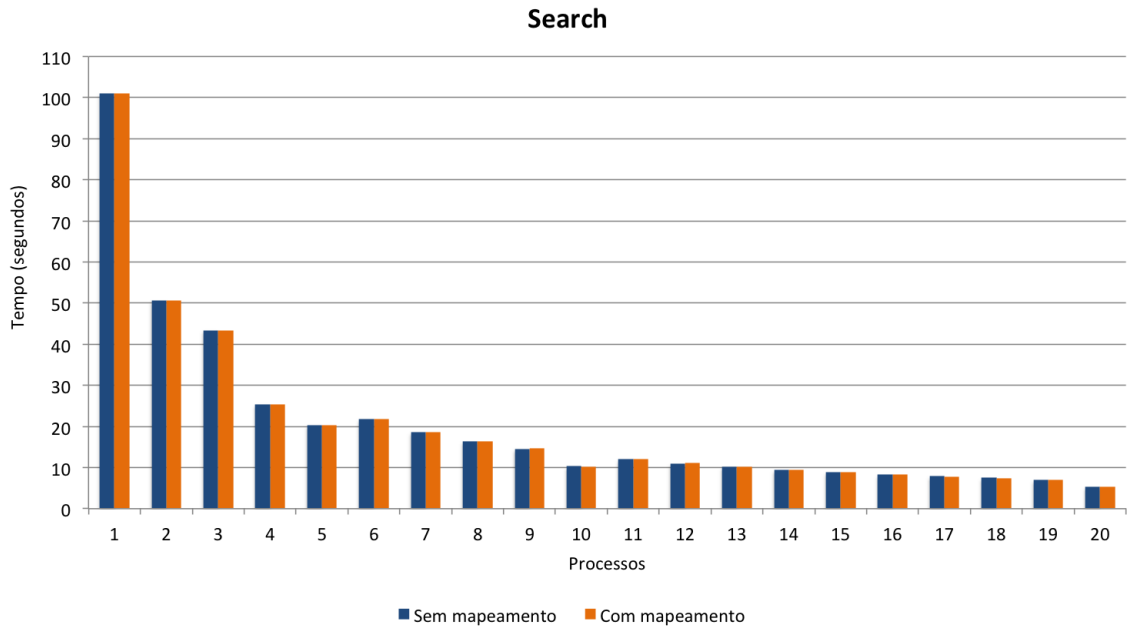


Figura 8.7 – Resultado da aplicação Search

## MatrixMult

A Figura 8.8 apresenta os tempos para a execução da aplicação de multiplicação de duas matrizes de dimensões 2000x2000. No gráfico, é possível visualizar que o mapeamento de memória apresentou bons resultados.

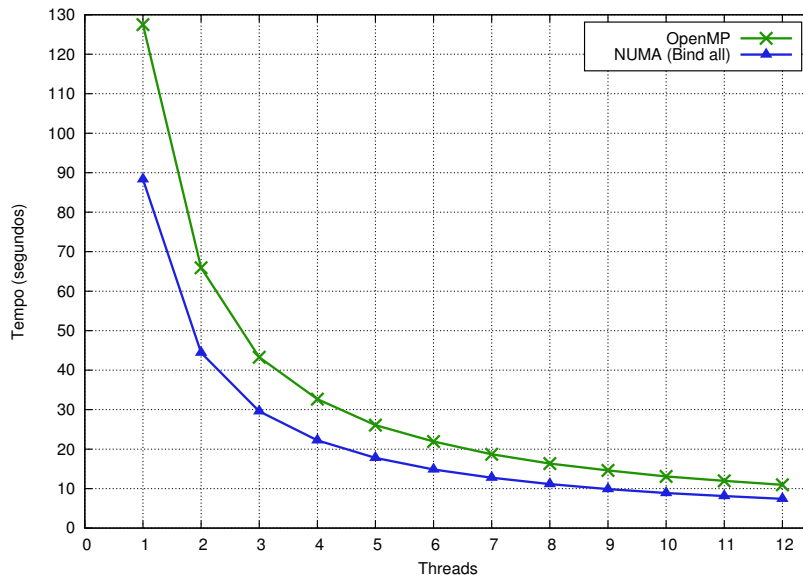


Figura 8.8 – Resultados da aplicação MatrixMult utilizando políticas de alocação de memória

Os tempos de execução foram comparados com a utilização do OpenMP sem alocação eficiente de memória. Quando a política *bind\_all* é aplicada, no entanto, percebe-se que os tempos de execução em comparação à versão apenas com OpenMP são menores.

Com apenas uma *thread* é possível verificar a diferença dos tempos de execução. Isto se deve ao fato de que, com os dados alocados mais próximos de quem vai utilizá-los, o tempo de acesso a estes dados conseqüentemente diminuirá. O mesmo acontece para o restante das execuções. Além do melhor desempenho, a versão com afinidade de memória segue o mesmo padrão de comportamento verificado na versão com OpenMP.

A Tabela 8.9 mostra os tempos de execução obtidos para cada política implementada nesta aplicação: *bind\_all*, *bind\_block*, *cyclic\_block* e *cyclic*.

Tabela 8.9 – Tempos para a aplicação *MatrixMult* utilizando diferentes políticas de alocação de memória

Threads	Bind-all		Bind-block		Cyclic-block		Cyclic	
	Média	Des. Pad.	Média	Des. Pad.	Média	Des. Pad.	Média	Des. Pad.
1	88,3217	1,0078	87,8475	0,4767	89,2825	0,3281	88,2517	0,7772
2	44,4234	0,4425	44,0204	0,1641	44,6780	0,3571	44,3902	0,4924
3	29,5661	0,2631	29,4814	0,2309	29,7321	0,1639	29,5963	0,3048
4	22,2104	0,1712	22,0241	0,1255	22,4329	0,1607	22,3010	0,3565
5	17,7755	0,1756	17,6982	0,1303	17,8872	0,1143	17,7481	0,1211
6	14,8322	0,1549	14,7295	0,0873	14,9306	0,0692	14,8621	0,1837
7	12,7483	0,1599	12,6620	0,0717	12,7681	0,1157	12,6800	0,0926
8	11,1290	0,1115	11,0732	0,0799	11,2022	0,1187	11,1167	0,1081
9	9,8718	0,0718	9,8657	0,1116	9,9741	0,0394	9,8935	0,1093
10	8,8736	0,0678	8,8559	0,0525	8,9586	0,0442	8,8975	0,1118
11	8,1001	0,0760	8,0569	0,0591	8,1476	0,0421	8,0826	0,0680
12	7,4167	0,0570	7,4055	0,0591	7,4910	0,0658	7,4535	0,0867

Nota-se que todas as políticas apresentaram um ganho de desempenho em relação ao OpenMP nos valores mostrados. Entretanto, as diferenças de tempos entre as diferentes políticas não foram muito significativas. Isto talvez deva-se ao fato de que a arquitetura utilizada apresenta apenas 2 nodos NUMA, e as alocações não se distanciam tanto umas das outras.

## Mandelbrot

O conjunto de *MandelBrot* foi utilizado para testes de mapeamento de memória. A implementação foi retirada de [Bur17b], e o mapeamento da memória foi realizado diretamente na versão original.

A aplicação *Mandelbrot* para uma imagem de tamanho 10000x10000 apresentou resultados conforme ilustrado na Figura 8.9. Assim como na aplicação de multiplicação de matrizes, a aplicação *Mandelbrot* obteve um desempenho melhor do que a versão apenas com OpenMP quando aplicada uma política de afinidade de memória, e observa-se que o comportamento das curvas é similar.

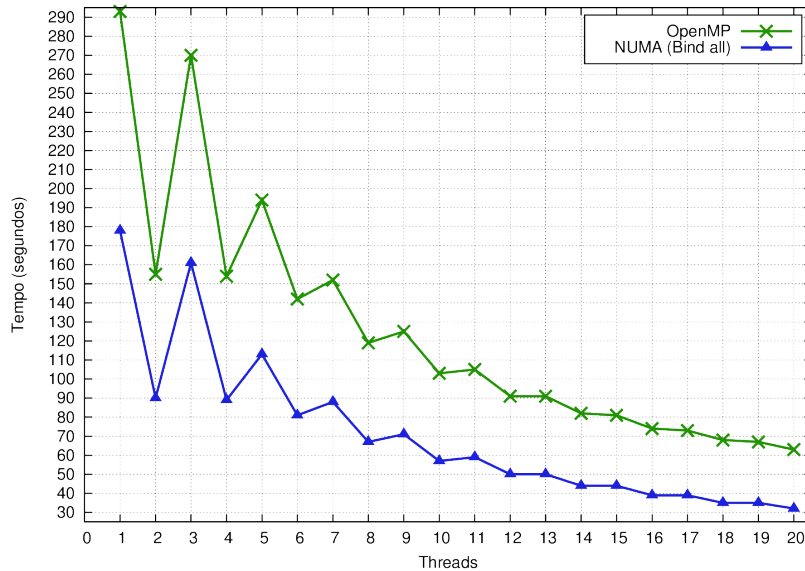


Figura 8.9 – Resultados obtidos pela aplicação Mandelbrot utilizando política de alocação de memória

## Dijkstra

A implementação da distância mínima de Dijkstra foi retirada de [Bur17a]. Para o algoritmo de Dijkstra, foram encontrados resultados conforme ilustra o gráfico da Figura 8.10. Os tempos apresentados referem-se a um grafo de 40000 nodos. O gráfico dos tempos de execução ilustra bem a diferença entre as versões.

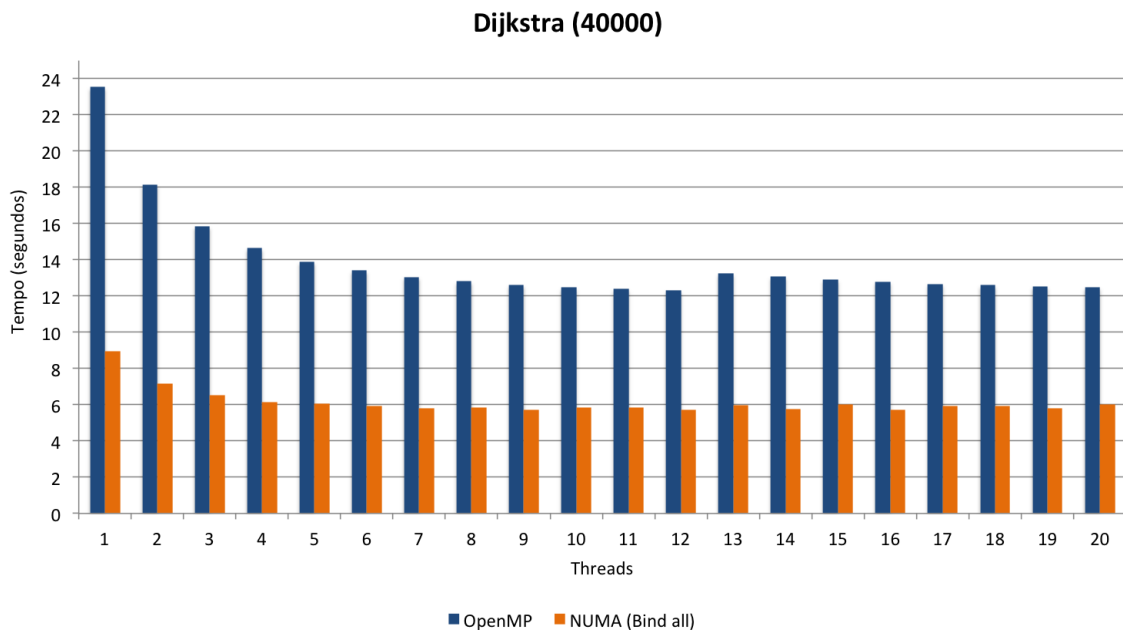


Figura 8.10 – Resultados obtidos com a aplicação Dijkstra para um grafo de 40000 nodos

Pode ser verificado que os resultados com a aplicação da política de memória *bind\_all* começam menores já com 1 *thread* apenas, continuando com valores similares a

partir de 4 processos. O mesmo pode ser verificado com a aplicação sem o mapeamento de *threads*.

### 8.3.3 Híbrido

Para a avaliação do modelo de mapeamento híbrido, a aplicação *MatrixMult* foi utilizada. A aplicação, assim como nos testes do mapeamento de memória, foram adaptadas para utilizar *threads* e, assim, viabilizar a definição de políticas de afinidade de memória.

Para a aplicação de multiplicação de matrizes neste contexto, o mapeamento híbrido apresentou resultados melhores do que aqueles obtidos apenas com o mapeamento de processos pesados, descrito anteriormente. Este fato pode ser visualizado no gráfico da Figura 8.11, que mostra os resultados para a utilização de 2 *threads* em cada nodo para matrizes de dimensões 2000x2000.

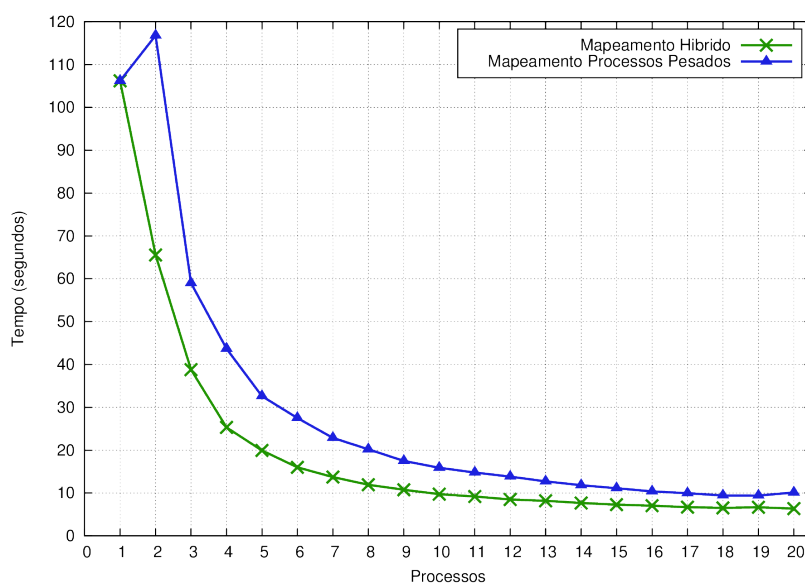


Figura 8.11 – Tempos obtidos com a aplicação híbrida *MultMatrix* com 2 *threads* por processo

Nota-se que desde a utilização de 2 processos, mesmo com apenas 2 *threads*, o mapeamento híbrido apresenta um desempenho melhor do que o obtido com 2 processos no mapeamento de processos pesados. O ganho obtido continua até 20 processos. A Tabela 8.10 apresenta os tempos de execução da aplicação multiplicação de matrizes para os dois mapeamentos.

Além dos tempos com a utilização de 2 *threads*, a Tabela 8.11 traz os tempos obtidos com a utilização de 20 *threads* por nodo quando a política de memória *bind\_all* é aplicada.

Tabela 8.10 – Tempos obtidos pela aplicação MultMatrix híbrida com 2 *threads* por nodo.

Processos	Mapeamento Híbrido		Mapeamento de Processos Pesados	
	Média	Desvio Padrão	Média	Desvio Padrão
2	65,5426	1,4630	116,7701	4,1378
3	38,8065	0,7649	59,0009	1,9697
4	25,3159	0,2069	43,6892	1,3221
5	19,9296	0,7646	32,6136	0,5053
6	16,0167	0,2195	27,4985	0,5393
7	13,7417	0,0827	22,8614	0,5907
8	11,9459	0,1928	20,1991	0,5496
9	10,7629	0,3309	17,5130	0,3514
10	9,7277	0,1257	15,9022	0,2064
11	9,1850	0,0730	14,7830	0,4302
12	8,4936	0,0450	13,8408	0,7277
13	8,1808	0,2645	12,7343	0,5037
14	7,6787	0,1294	11,8299	0,2291
15	7,2981	0,1899	11,1231	0,4432
16	7,0536	0,1658	10,3784	0,2015
17	6,6919	0,0454	9,9696	0,1775
18	6,5257	0,0640	9,4435	0,0742
19	6,6791	0,3216	9,4108	0,2673
20	6,3815	0,0368	10,1150	0,3838

Tabela 8.11 – Tempos obtidos pela aplicação MultMatrix híbrida com 20 *threads* por nodo.

Processos	Mapeamento Híbrido		Mapeamento de Processos Pesados	
	Média	Desvio Padrão	Média	Desvio Padrão
2	11,9541	0,0626	116,7701	4,1378
3	7,1327	0,1322	59,0009	1,9697
4	5,3090	0,0617	43,6892	1,3221
5	4,3235	0,1252	32,6136	0,5053
6	3,6865	0,0532	27,4985	0,5393
7	3,4066	0,0620	22,8614	0,5907
8	3,2684	0,0253	20,1991	0,5496
9	3,2176	0,0304	17,5130	0,3514
10	3,2615	0,0459	15,9022	0,2064
11	3,1449	0,0389	14,7830	0,4302
12	3,2320	0,0166	13,8408	0,7277
13	3,2854	0,0151	12,7343	0,5037
14	3,1833	0,0461	11,8299	0,2291
15	3,3430	0,0784	11,1231	0,4432
16	3,4226	0,0297	10,3784	0,2015
17	3,5067	0,0473	9,9696	0,1775
18	3,5514	0,0647	9,4435	0,0742
19	3,6652	0,0372	9,4108	0,2673
20	3,6782	0,0166	10,1150	0,3838



Os tempos são ainda melhores neste caso. Com 20 *threads* e com 2 processos, por exemplo, o tempo obtido já é de aproximadamente 12 segundos, enquanto o mapeamento apenas dos processos pesados apresenta aproximadamente 116 segundos. Com 11 *threads* e com a política *bind\_all*, obteve-se o menor tempo de execução, com 3,1449 segundos (e 0,0389 segundos de desvio padrão).



## 9. CONCLUSÕES E TRABALHOS FUTUROS

A programação paralela, principalmente para ambientes híbridos, não é uma tarefa trivial. Neste contexto, torna-se interessante a tentativa de desenvolver um processo que permita gerar aplicações híbridas de forma automática. Esta geração automática, no entanto, passa por uma série de obstáculos, principalmente para tornar esta tarefa transparente para o usuário final. Entretanto, diversas ferramentas, bibliotecas e mecanismos existentes podem auxiliar nesta tarefa. É o caso do MPI, do OpenMP, da interface MAI e dos padrões de programação paralela apresentados no decorrer do trabalho, que foram combinados no modelo criado.

O objetivo do trabalho, então, passava por esta criação automática, e cinco questões de pesquisa foram definidas. Foram definidas 5 hipóteses, que respondem a estas questões de pesquisa. Assim, uma breve análise sobre as hipóteses será realizada, de acordo com os resultados obtidos com o desenvolvimento deste trabalho:

- **Hipótese 1 (H1):** *As primitivas de comunicação (troca de mensagens) entre os diferentes nodos de um cluster podem ser totalmente abstraídas.* Através do desenvolvimento do mapeamento de processos pesados, é possível verificar que, no processo criado, o usuário não tem contato com primitivas MPI de comunicação, ficando totalmente a cargo do modelo desenvolvido controlar a sincronia e comunicação. Assim, a hipótese pode ser avaliada como verdadeira.
- **Hipótese 2 (H2):** *A aplicação de políticas de alocação eficiente de memória em máquinas NUMA pode ser realizada de maneira transparente.* Da mesma forma, o desenvolvimento de um mapeamento de memória que realiza as chamadas a primitivas de baixo nível para alocação de memória confirma esta hipótese. Usuários não necessitam conhecer as funções da biblioteca MAI, tampouco de funções da própria NUMA API. Desta forma, pode-se dizer que esta hipótese é válida, e os exemplos apresentados corroboram com esta resposta.
- **Hipótese 3 (H3):** *Os recursos computacionais de um cluster, bem como a interação entre eles, podem ser descritos a partir de uma interface gráfica.* O desenvolvimento de um protótipo viabiliza a interação dos usuários com o modelo desenvolvido. Assim, usuários criam seus grafos na ferramenta gráfica, possibilitando a visualização do ambiente do cluster de maneira mais interessante. Logo, a hipótese pode ser considerada verdadeira.
- **Hipótese 4 (H4):** *Padrões de programação paralela já conhecidos podem estar disponíveis para o usuário.* A utilização de skeletons no modelo de mapeamento de processos pesados e no mapeamento híbrido facilita o controle da execução das aplicações.

Quando o padrão já é conhecido, pode ser melhor controlado, evitando dúvidas no desenvolvimento. Desta maneira, disponibilizar padrões paralelos já definidos é de fato importante. No modelo descrito no trabalho, alguns modelos bem conhecidos foram incorporados no modelo, e a hipótese é verdadeira.

- **Hipótese 5 (H5):** *Se bem estruturado, um modelo automático para criação de aplicações híbridas pode gerar aplicações com um desempenho próximo ao obtido em aplicações criadas manualmente.* Esta hipótese pode ser avaliada de acordo com os resultados obtidos. Aplicações implementadas com a utilização da geração automática apresentaram desempenho bastante similar àquelas realizadas manualmente. A Tabela 9.1 apresenta as diferenças (em segundos) dos tempos obtidos com algumas das aplicações apresentadas anteriormente. A comparação foi realizada através da subtração dos tempos obtidos com a versão utilizando o mapeamento desenvolvido no trabalho daqueles obtidos manualmente<sup>1</sup>.

Tabela 9.1 – Diferenças (em segundos) dos tempos obtidos com o mapeamento proposto. Valores positivos indicam um aumento no tempo de execução da aplicação, enquanto valores negativos indicam diminuição neste mesmo tempo.

Processos	N-Queens	CountElements	FindText	MatrixMult
2	1,5364	0,6919	1,3865	15,1934
3	0,7597	0,5351	0,9938	7,8980
4	0,5252	0,4482	0,4333	2,6320
5	0,9363	0,6164	0,3681	1,2786
6	0,8045	0,4196	0,2784	1,1004
7	0,5747	0,4601	0,4218	0,6246
8	0,4760	0,7419	0,2687	1,4795
9	0,4254	0,4399	0,1756	1,2217
10	0,0846	0,4624	0,1892	1,2387
11	0,1602	0,4845	0,1591	0,9750
12	0,4889	0,3984	0,0254	1,3748
13	0,2528	0,6579	0,0354	0,7471
14	0,1396	0,7855	0,1070	0,6653
15	0,2327	1,0026	0,0994	-0,5512
16	0,1773	1,0291	0,1707	-0,7597
17	0,1827	0,7585	0,1558	-0,6221
18	0,4040	0,8938	0,2950	-0,6356
19	0,1975	0,5003	0,1266	-0,2671
20	0,3455	0,9834	0,0609	0,4383

Na Tabela 9.1, é possível perceber que existem diferenças negativas, indicando que a versão com o mapeamento apresentado obteve um desempenho melhor do que a versão manual. No geral, os tempos das aplicações ficaram próximos, à luz da maior facilidade de geração dos códigos com a solução descrita neste trabalho. Além disto, o comportamento dos resultados seguiu o mesmo padrão das aplicações tradicionais.

<sup>1</sup>Os tempos utilizados para a obtenção dos resultados da Tabela 9.1 são os tempos apresentados no Capítulo 8.

Finalmente, a análise estatística realizada nos tempos de execução (utilizando o teste t de Student) indicam que não existem diferenças significativas nos tempos de execução. Este é um fator muito importante, pois mostra que o código automático não apresentou perda significativa de desempenho em relação ao código manual. Com isto, a última hipótese também pode ser avaliada como verdadeira.

Alguns fatos enriquecem o modelo desenvolvido, e merecem destaque, tais como: a possibilidade da divisão em um número x de tarefas indicadas pelo usuário; a utilização da redundância de dados (borda); a criação dinâmica de processos no padrão D&C (*Spawn*) e; a possibilidade de acesso a algumas informações globais.

Outro ponto importante neste sentido, trata-se da possibilidade de o usuário utilizar o modelo para criar diferentes configurações para suas aplicações, permitindo, inclusive, comparar o desempenho entre as versões. Com isto, é possível avaliar que tipo de política é mais adequada para aquela aplicação.

Além disto, também merece destaque a presença do gerador de código paralelo no protótipo desenvolvido. Este é um artefato bastante interessante, pois é com ele que as primitivas MPI são abstraídas do programador. Apenas com as informações do grafo do usuário e dos dados a serem comunicados, os códigos paralelos são criados para o programador apenas incluir seus códigos sequenciais.

A utilização de skeletons no desenvolvimento do modelo de mapeamento de processos pesados foi de extrema importância. Primeiramente, estes padrões pré-definidos e já bem conhecidos da literatura trazem maior facilidade no controle da execução da aplicação paralela, além de fornecerem alta capacidade de abstração. Além disto, a criação do controle e de todas as funcionalidades dos *skeletons* apresentados norteou o desenvolvimento de modelos genéricos, nos quais o programador não precisa necessariamente criar sua aplicação vinculada a algum modelo. No entanto, o arquivo *template* criado pela ferramenta fica cada vez mais complexo e possui mais métodos à medida em que mais nodos e arcos vão sendo inseridos no grafo.

Com isto, alguns prós e contras da utilização do modelo proposto podem ser elencados. Os principais (segundo o autor) são:

- Prós:
  - criação de código paralelo de forma automática;
  - programador não precisa conhecimento das primitivas de comunicação de baixo nível da biblioteca;
  - criação de modelos de programação paralela já existentes de maneira ágil e prática;
  - não é necessário controlar deadlocks, pois o código gerado realiza as chamadas necessárias para garantir que este problema não ocorra no código final;

- **Contras:**
  - pode não ser trivial simplesmente completar o template com o código sequencial já existente;
  - alguns problemas são mais difíceis de paralelizar e precisam de modelos menos rígidos, o que se torna inviável de realizar;
  - tipos de dados enviados são apenas tipos primitivos, não sendo possível enviar tipos mais sofisticados, como arquivos, por exemplo;
  - existe uma curva de aprendizado para aprender a utilizar a interface gráfica, e a utilização incorreta pode levar a inconsistência nos resultados.

Finalmente, os modelos foram desenvolvidos sempre com o intuito de facilitar a utilização do usuário e de abstrair dele questões mais vinculadas ao hardware. De acordo com os resultados obtidos, acredita-se que os objetivos do trabalho foram atingidos. Além disto, o protótipo desenvolvido possui praticamente todas as funcionalidades de uma versão final, e a ferramenta será disponibilizada para a comunidade acadêmico-científica tão logo uma versão com testes de usabilidade da interface gráfica sejam realizados.

## 9.1 Trabalhos Futuros

Embora se tenha obtido bons resultados com o desenvolvimento realizado, alguns trabalhos futuros são imprescindíveis para aprimorar o modelo desenvolvido. A seguir, alguns deles são elencados:

- **Aninhamento de padrões:** é possível permitir o aninhamento de padrões nos modelos desenvolvidos. Por exemplo, uma aplicação que utiliza o modelo Pipe pode ter um dos processos com uma computação demasiadamente grande. Neste caso, por exemplo, este processo do Pipe pode ser o Master de um modelo aninhado Master/Slave, no qual outros processos vão ajudá-lo a realizar sua tarefa;
- **Balanceamento e escalonamento:** módulos de balanceamento de carga e escalonamento podem ser incorporados ao modelo, podendo até mesmo levar em conta características do próprio hardware informado pelo usuário (memória, processador, *cores* etc.);
- **XML de entrada:** como alternativa à entrada das informações necessárias através de uma interface gráfica, pode ser incorporado ao sistema uma entrada textual, no formato XML, por exemplo. Esta alternativa poderia facilitar, inclusive, a criação de scripts para testes automatizados das aplicações a serem paralelizadas;

- **Novos padrões:** é possível disponibilizar diferentes padrões de programação paralela já conhecidos no modelo. Estes padrões podem estar previamente definidos no sistema, facilitando ainda mais a implementação das aplicações paralelas, pois as primitivas de comunicação estariam previamente definidas, evitando algum tipo de deadlock criado acidentalmente pelo usuário;
- **Primitivas MPI:** em diversas situações é interessante (e até mesmo necessário) utilizar diferentes primitivas MPI, como barreiras e reduce, por exemplo. Além disto, pode ser previsto o envio de diferentes tipos de dados (objetos criados pelo usuário, por exemplo), ao invés de enviar apenas tipos primitivos do MPI pela rede. Neste caso, utilizar-se-ia uma forma de serialização de dados, que quando enviados pela rede seriam reconhecidos pelo outro nó.

Todas estas alternativas, em decorrência do desenvolvimento do trabalho apresentado, visam tornar a criação de aplicações paralelas mais simples e objetivas. Com isto, pesquisadores de diversas áreas podem aumentar o desempenho de seus programas sem a necessidade de um conhecimento prévio de primitivas de programação paralela.





## REFERÊNCIAS BIBLIOGRÁFICAS

- [AdTGR99] Acacio, M. E.; de Teruel, P. E. L.; Garcia, J. M.; Reverte, O. C. “The MPI-Delphi Interface: a Visual Programming Environment for Clusters of Workstations”. In: International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), 1999, pp. 1730–1736.
- [BAA+92] Babaoğlu, O.; Alvisi, L.; Amoroso, A.; Davoli, R.; Giachini, L. A. “PARALEX: an Environment for Parallel Programming in Distributed Systems”. In: 6th International Conference on Supercomputing (ICS), 1992, pp. 178–187.
- [Bar] Barney, B. “POSIX Threads Programming”. Capturado em: [computing.llnl.gov/tutorials/pthreads](http://computing.llnl.gov/tutorials/pthreads), Abril 2017.
- [BBMŠ14] Böhm, S.; Běhálek, M.; Meca, O.; Šurkovský, M. “KAIRA: Development Environment for MPI Applications”. In: International Conference on Applications and Theory of Petri Nets and Concurrency (PETRI NETS), 2014, pp. 385–394.
- [BCGH05] Benoit, A.; Cole, M.; Gilmore, S.; Hillston, J. “Flexible Skeletal Programming with Eskel”. In: 11th International Euro-Par Conference on Parallel Processing (Euro-Par), 2005, pp. 761–770.
- [BDG+91] Beguelin, A.; Dongarra, J.; Geist, G. A.; Manchek, R.; Sunderam, V. S. “Graphical Development Tools for Network-Based Concurrent Supercomputing”. In: ACM/IEEE conference on Supercomputing (SUPERCOMPUTING), 1991, pp. 435–444.
- [BDGS93] Beguelin, A.; Dongarra, J.; Geist, A.; Sunderam, V. “Visualization and Debugging in a Heterogeneous Environment”, *Computer*, vol. 26–6, Junho 1993, pp. 88–95.
- [BSF+91] Bolosky, W. J.; Scott, M. L.; Fitzgerald, R. P.; Fowler, R. J.; Cox, A. L. “NUMA Policies and their Relation to Memory Architecture”, *International Conference on Architectural Support for Programming Languages and Operating Systems*, vol. 19–2, Abril 1991, pp. 212–221.
- [Bur17a] Burkardt, J. “Dijkstra Graph Distance Algorithm using OpenMP”. Capturado em: [people.sc.fsu.edu/~jburkardt/c\\_src/dijkstra\\_openmp/dijkstra\\_openmp.html](http://people.sc.fsu.edu/~jburkardt/c_src/dijkstra_openmp/dijkstra_openmp.html), Abril 2017.
- [Bur17b] Burkardt, J. “Mandelbrot Image using OpenMP”. Capturado em: [people.sc.fsu.edu/~jburkardt/c\\_src/mandelbrot\\_openmp](http://people.sc.fsu.edu/~jburkardt/c_src/mandelbrot_openmp), Abril 2017.

- [Bur17c] Burkardt, J. “Parallel Search for Integer so that  $F(J) = C$ ”. Capturado em: [people.sc.fsu.edu/~jburkardt/cpp\\_src/search\\_mpi/search\\_mpi.html](http://people.sc.fsu.edu/~jburkardt/cpp_src/search_mpi/search_mpi.html), Abril 2017.
- [Cas09] Castro, M. B. “NUMA-ICTM: uma Versão Paralela do ICTM Explorando Estratégias de Alocação de Memória para Máquinas NUMA”, Dissertação de Mestrado, Pontifícia Universidade do Rio Grande do Sul, Porto Alegre, BR, 2009, 81p.
- [CCCZ05] Chan, F.; Cao, J.; Chan, A. T. S.; Zhang, K. “Visual Programming Support for Graph-Oriented Parallel-Distributed Processing: Research Articles”, *Software: Practice and Experience*, vol. 35–15, Dezembro 2005, pp. 1409–1439.
- [CFR<sup>+</sup>09] Castro, M. B.; Fernandes, L. G.; Ribeiro, C. P.; Mehaut, J.-F.; Aguiar, M. “NUMA-ICTM: a Parallel Version of ICTM Exploiting Memory Placement Strategies for NUMA Machines”. In: 10th International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC), 2009, pp. 1–8.
- [CGL98] Chan, A.; Gropp, W.; Lusk, E. “User’s Guide for MPE: Extensions for MPI Programs”, Relatório Técnico, University of Chicago, Chicaco, US, 1998, 36p.
- [CJP07] Chapman, B.; Jost, G.; Pas, R. V. D. “Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)”. Cambridge, US: The MIT Press, 2007, 378p.
- [Col91] Cole, M. “Algorithmic Skeletons: Structured Management of Parallel Computation”. Cambridge, US: MIT Press, 1991, 137p.
- [Col04] Cole, M. “Bringing Skeletons out of the Closet: a Pragmatic Manifesto for Skeletal Parallel Programming”, *Parallel Computing*, vol. 30–3, Março 2004, pp. 389–406.
- [DAD<sup>+</sup>08] Dupros, F.; Aochi, H.; Ducellier, A.; Komatitsch, D.; Roman, J. “Exploiting Intensive Multithreading for the Efficient Simulation of 3D Seismic Wave Propagation”. In: 11th IEEE International Conference on Computational Science and Engineering (CSE), 2008, pp. 253–260.
- [DBM<sup>+</sup>09] Dave, C.; Bae, H.; Min, S. J.; Lee, S.; Eigenmann, R.; Midkiff, S. “CETUS: a Source-to-Source Compiler Infrastructure for Multicores”, *Computer*, vol. 42–12, Dezembro 2009, pp. 36–42.
- [De 06] De Rose, C. A. F. “Fundamentos de Processamento de Alto Desempenho”. In: Comissão Regional de Alto Desempenho (CRAD-RS), 2006, pp. 11–38.
- [Dem97] Demaine, E. D. “A Threads-Only MPI Implementation for the Development of Parallel Programs”. In: 11th International Symposium on High Performance Computing Systems (HPCS), 1997, pp. 153–163.

- [DM98] Dagum, L.; Menon, R. “OpenMP: an Industry Standard API for Shared-Memory Programming”, *IEEE computational science and engineering*, vol. 5–1, Janeiro 1998, pp. 46–55.
- [DRN03] De Rose, C. A. F.; Navaux, P. O. A. “Arquiteturas Paralelas”. Porto Alegre, BR: Sagra Luzzatto, 2003, 32p.
- [FAGC16] Fabian, J. L. Q.; Alonso, G. R.; García, M. A. C.; Cornejo, M. A. “Hiding Parallel MPI Programming behind Graphical Workflow Specifications”. In: *Computational Science and Computational Intelligence (CSCI)*, 2016, pp. 578–583.
- [FSCL06] Falcou, J.; Sérot, J.; Chateau, T.; Lapresté, J. T. “QUAFF: Efficient C++ Design for Parallel Skeletons”, *Parallel Computing*, vol. 32–7, Setembro 2006, pp. 604–615.
- [HE14] Hothorn, T.; Everitt, B. S. “A Handbook of Statistical Analyses using R”. Boca Raton, US: CRC Press, 2014, 376p.
- [HLK03] Huang, C.; Lawlor, O. S.; Kalé, L. V. “Adaptive MPI”. In: *16th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2003, pp. 306–322.
- [Hoe06] Hoeflinger, J. P. “Extending OpenMP to Clusters (White Paper)”, Relatório Técnico, Intel Corporation, Santa Clara, US, 2006, 10p.
- [HQL+91] Hatcher, P. J.; Quinn, M. J.; Lapadula, A. J.; SeEVERS, B. K.; Anderson, R. J.; Jones, R. R. “Data-Parallel Programming on MIMD Computers”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 2–3, Julho 1991, pp. 377–383.
- [JK09] Jensen, K.; Kristensen, L. M. “Coloured Petri Nets: Modelling and Validation of Concurrent Systems”. Springer Science & Business Media, 2009.
- [KDF96] Kacsuk, P.; Dózsa, G.; Fadgyas, T. “Designing Parallel Programs by the Graphical Language GRAPNEL”, *Microprocessing and Microprogramming*, vol. 41–8, Abril 1996, pp. 625–643.
- [Kle04] Kleen, A. “A NUMA API for Linux”. Capturado em: [www.halobates.de/numaapi3.pdf](http://www.halobates.de/numaapi3.pdf), Abril 2017.
- [KSS96] Kleiman, S.; Shah, D.; Smaalders, B. “Programming with Threads”. Ann Arbor, US: Sun Soft Press, 1996, 534p.
- [LB95] Lewis, B.; Berg, D. J. “Threads Primer: a Guide to Multithreaded Programming”. Mountain View, US: Prentice Hall Press, 1995, 352p.

- [LIG17] LIG. “LIG - Laboratoire d’Informatique de Grenoble”. Capturado em: [www.liglab.fr](http://www.liglab.fr), Abril 2017.
- [Lin05] Lin, X. “Active Layout Engine: Algorithms and Applications in Variable Data Printing”, Relatório Técnico, Hewlett & Packard Laboratories, Palo Alto, CA, USA, 2005, 33p.
- [LSB07] Lee, J.; Sato, M.; Boku, T. “Design and Implementation of OpenMPD: an OpenMP-Like Programming Language for Distributed Memory Systems”. In: 3rd International Workshop on OpenMP (IWOMP), 2007, pp. 143–147.
- [MPI17] MPI. “MPI: A Message-Passing Interface Standard”. Capturado em: [mpi-forum.org/docs/mpi-1.1/mpi-11-html/mpi-report.html](http://mpi-forum.org/docs/mpi-1.1/mpi-11-html/mpi-report.html), Abril 2017.
- [MQ93] McCallum, D.; Quinn, M. J. “A Graphical User Interface for Data-Parallel Programming”. In: 26th Hawaii International Conference on System Sciences (HICSS), 1993, pp. 5–13.
- [MSD17] MSDN. “MSDN Home Page”. Capturado em: [www.msdn.microsoft.com](http://www.msdn.microsoft.com), Abril 2017.
- [NB92] Newton, P.; Browne, J. C. “The CODE 2.0 Graphical Parallel Programming Language”. In: 6th International Conference on Supercomputing (ICS), 1992, pp. 167–177.
- [ND94] Newton, P.; Dongarra, J. “Overview of VPE: A Visual Environment for Message-Passing Parallel Programming”, Relatório Técnico, University of Tennessee, Knoxville, US, 1994, 8p.
- [PJN08] Pérache, M.; Jourden, H.; Namyst, R. “MPC: a Unified Parallel Runtime for Clusters of NUMA Machines”. In: 14th International Euro-Par Conference on Parallel Processing (Euro-Par), 2008, pp. 78–88.
- [PVM17] PVM. “PVM Home Page”. Capturado em: [www.csm.ornl.gov/pvm](http://www.csm.ornl.gov/pvm), Abril 2017.
- [RE16] Ricardo, A.; Esmeraldo, G. “Uma Linguagem Visual para Suporte à Programação de Aplicações Paralelas de Alto Desempenho”. In: XVII Simpósio em Sistemas Computacionais de Alto Desempenho - Workshop de Iniciação Científica (WIC-WSCAD), 2016, pp. 56–61.
- [RGR+11] Raeder, M.; Griebler, D.; Ribeiro, N. S.; Fernandes, L. G.; Castro, M. B. “A Hybrid Parallel Version of ICTM for Cluster of NUMA Machines”. In: IADIS International Conference on Applied Computing (AC), 2011, pp. 291–298.

- [Rib11] Ribeiro, N. S. “Explorando Programação Híbrida no Contexto de Clusters de Máquinas NUMA”, Dissertação de Mestrado, Pontifícia Universidade do Rio Grande do Sul, Porto Alegre, BR, 2011, 81p.
- [RM10] Ribeiro, C. P.; Mehaut, J.-F. “MAI: Memory Affinity Interface”, Relatório Técnico, Institut National de Recherche en Informatique et en Automatique (INRIA), Grenoble, FR, 2010, 29p.
- [RMC+09] Ribeiro, C. P.; Mehaut, J.-F.; Carissimi, A.; Castro, M. B.; Fernandes, L. G. “Memory Affinity for Hierarchical Shared Memory Multiprocessors”. In: 21st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), 2009, pp. 59–66.
- [RMM+08] Ribeiro, C. P.; Marangozova, V.; Mehaut, J.-F.; Dupros, F.; Carissimi, A. “Explorando Afinidade de Memória em Arquiteturas NUMA”. In: IX Simpósio em Sistemas Computacionais (WSCAD-SSC), 2008, pp. 76–83.
- [Rol08] Rolfe, T. J. “A Specimen MPI Application: N-Queens in Parallel”. Capturado em: [penguin.ewu.edu/~trolfe/MpiQueen/MPI\\_Queen.doc](http://penguin.ewu.edu/~trolfe/MpiQueen/MPI_Queen.doc), Abril 2017.
- [RR13] Rauber, T.; Rüniger, G. “Parallel Programming for Multicore and Cluster Systems”. Berlin, DE: Springer Science & Business Media, 2013, 516p.
- [SGG09] Silberschatz, A.; Galvin, P. B.; Gagne, G. “Operating System Concepts”. Hoboken, US: J. Wiley & Sons, 2009, 972p.
- [SMT00] Sakayori, Y.; Miura, M.; Tanaka, J. “Programming Environment Specified for Interprocessor Communications Based on Graphical User Interface”. In: International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), 2000, pp. 2779–2785.
- [SOHL+98] Snir, M.; Otto, S.; Huss-Lederman, S.; Walker, D.; Dongarra, J. “MPI - The Complete Reference”. Cambridge, US: MIT Press, 1998, 429p.
- [SSS13] Spiegel, M.; Schiller, J.; Srinivasan, R. “Probabilidade e Estatística: 897 Problemas Resolvidos”. Porto Alegre, BR: Bookman, 2013, 440p.
- [Yan13] Yaneva, V. “Visual Parallel Programming”, Dissertação de Mestrado, The University of Edinburgh, Brighton, UK, 2013, 75p.
- [Zom96] Zomaya, A. “Parallel and Distributed Computing Handbook”. New York, US: McGraw-Hill, 1996, 1184p.



## APÊNDICE A – RESULTADO $N$ – *Queens*

Nas Tabelas A.1, A.2 e A.3, podem ser visualizados os tempos obtidos com a aplicação  $N$  – *Queens* para  $N=4$ ,  $N=8$  e  $N=12$ , respectivamente.

Tabela A.1 – Tempos obtidos com a aplicação  $N$  – *Queens* para  $N=4$

Processos	Com Mapeamento	
	Média	Desvio Padrão
2	0,0006	0,0000
3	0,0008	0,0001
4	0,0012	0,0000
5	0,0015	0,0001
6	0,0017	0,0001
7	0,0020	0,0001
8	0,0022	0,0001
9	0,0026	0,0001
10	0,0028	0,0001
11	0,0031	0,0001
12	0,0034	0,0001
13	0,0037	0,0001
14	0,0042	0,0001
15	0,0045	0,0001
16	0,0048	0,0001
17	0,0052	0,0001
18	0,0056	0,0001
19	0,0061	0,0002
20	0,0063	0,0002

Tabela A.2 – Tempos obtidos com a aplicação  $N$  – *Queens* para  $N=8$

Processos	Com Mapeamento	
	Média	Desvio Padrão
2	0,0009	0,0001
3	0,0010	0,0001
4	0,0011	0,0001
5	0,0015	0,0001
6	0,0017	0,0002
7	0,0020	0,0001
8	0,0022	0,0002
9	0,0025	0,0002
10	0,0027	0,0001
11	0,0030	0,0002
12	0,0033	0,0002
13	0,0037	0,0002
14	0,0041	0,0002
15	0,0044	0,0001
16	0,0046	0,0002
17	0,0051	0,0003
18	0,0055	0,0001
19	0,0059	0,0002
20	0,0063	0,0002

Tabela A.3 – Tempos obtidos com a aplicação *N – Queens* para N=12

Processos	Com Mapeamento	
	Média	Desvio Padrão
2	0,0647	0,0143
3	0,0367	0,0047
4	0,0251	0,0037
5	0,0212	0,0042
6	0,0180	0,0027
7	0,0139	0,0020
8	0,0155	0,0007
9	0,0147	0,0016
10	0,0156	0,0008
11	0,0161	0,0004
12	0,0163	0,0004
13	0,0168	0,0004
14	0,0167	0,0016
15	0,0174	0,0004
16	0,0176	0,0006
17	0,0179	0,0011
18	0,0185	0,0007
19	0,0187	0,0006
20	0,0193	0,0004



## APÊNDICE B – RESULTADO *CountElements*

As Figuras B.1 e B.2 apresentam, respectivamente, os gráficos referentes aos testes da aplicação *CountElements* com tamanho 10000000000 e 5000000000.

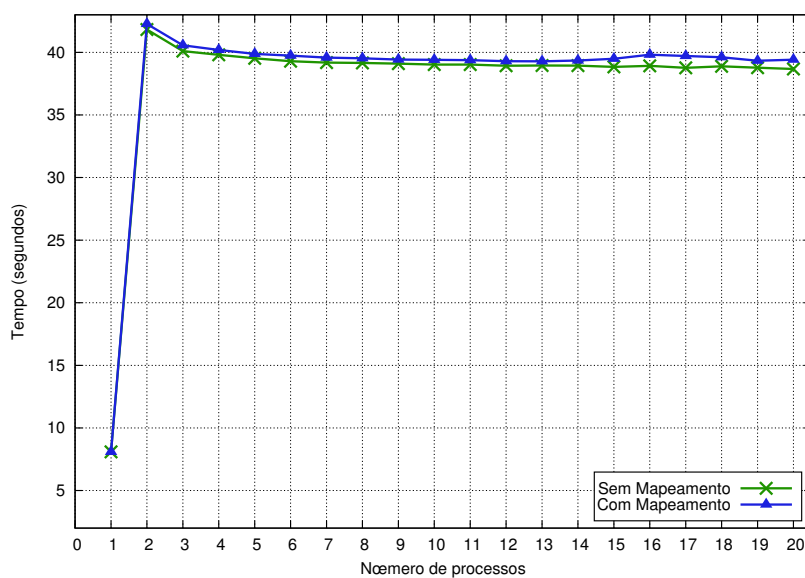


Figura B.1 – *CountElements* para o tamanho 10000000000.

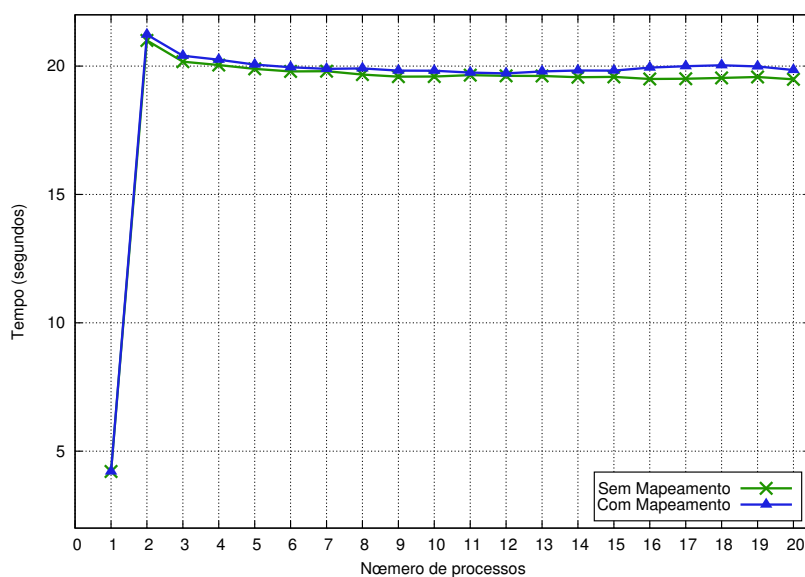


Figura B.2 – *CountElements* para o tamanho 5000000000.