CONSTRUÇÃO DE NÚCLEOS PARALELOS DE ÁLGEBRA LINEAR COMPUTACIONAL COM EXECUÇÃO GUIADA POR FLUXO DE DADOS

Brunno Figueirôa Goldstein

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientadores: Felipe Maia Galvão França
Leandro Augusto Justen Marzulo

Rio de Janeiro
Setembro de 2015

CONSTRUÇÃO DE NÚCLEOS PARALELOS DE ÁLGEBRA LINEAR COMPUTACIONAL COM EXECUÇÃO GUIADA POR FLUXO DE DADOS

Brunno Figueirôa Goldstein

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:

Prof. Felipe Maia Galvão França, Ph.D.

Prof. Leandro Augusto Justen Marzulo, D.Sc.

Prof. Luiz Mariano Paes de Carvalho Filho, Ph.D.

Prof. Claudio Luis de Amorim, Ph.D.

RIO DE JANEIRO, RJ – BRASIL
SETEMBRO DE 2015

Goldstein, Brunno Figueirôa

Construção de Núcleos Paralelos de Álgebra Linear Computacional com Execução Guiada por Fluxo de Dados/Brunno Figueirôa Goldstein. – Rio de Janeiro: UFRJ/COPPE, 2015.

XV, 58 p.: il.; 29, 7cm.

Orientadores: Felipe Maia Galvão França

Leandro Augusto Justen Marzulo

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2015.

Referências Bibliográficas: p. 48 – 50.

1. Dataflow. 2. Métodos Iterativos. 3. Solver Triangular Parallelo. 4. Matriz Esparsa. I. França, Felipe Maia Galvão *et al.*. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

Aos meus pais pelo dom da vida e pelo amparo ao longo desses anos. À minha amada Juliana por todo apoio e carinho durante toda minha vida acadêmica.

Agradecimentos

Agradeço a Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) e o Centro de Pesquisas Leopoldo Américo Miguez de Mello (CENPES) da Petrobras pelo suporte financeiro.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

CONSTRUÇÃO DE NÚCLEOS PARALELOS DE ÁLGEBRA LINEAR COMPUTACIONAL COM EXECUÇÃO GUIADA POR FLUXO DE DADOS

Brunno Figueirôa Goldstein

Setembro/2015

Orientadores: Felipe Maia Galvão França
Leandro Augusto Justen Marzulo

Programa: Engenharia de Sistemas e Computação

Núcleos de Álgebra Linear possuem um papel fundamental em diversos sistemas de simulação de reservatório petrolíferos empregados no mercado atualmente. Devido ao crescimento dos problemas simulados por esses sistemas, o tempo de processamento de tais núcleos tornou-se um fator determinante para viabilidade dos simuladores. Contudo, devido ao grande potencial de paralelismo oferecido pelas CPUs modernas, a implementação dos núcleos paralelos mostrou-se uma opção promissora para diminuição do tempo de processamento dessas aplicações. Acredita-se que a aplicação de modelos paralelos, mais especificamente o modelo *dataflow*, possam tirar maior proveito da dependência de dados presente nos núcleos apresentados. O presente trabalho consiste em implementar núcleos paralelos de álgebra linear utilizando o modelo *dataflow* como base. Para tal, os ambientes *Trebuchet* e *Sucuri* foram utilizadas. Os núcleos implementados, foram avaliados com um conjunto de dados reais provenientes de simuladores utilizados no mercado. Os resultados foram comparados com versões implementadas em *OpenMP* e *Intel® MKL*. Pode-se observar que a baixa granularidade de um conjunto de núcleos, somados a sobrecarga dos ambientes *dataflow*, limitaram o desempenho das aplicações. Porém, os núcleos Produto Matriz Vetor e Produto Matriz Matriz, mostraram-se promissores, chegando a superar as versões de comparação para o primeiro caso.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

CREATING PARALLEL LINEAR ALGEBRA KERNELS USING DATAFLOW MODEL

Brunno Figueirôa Goldstein

September/2015

Advisors: Felipe Maia Galvão França
Leandro Augusto Justen Marzulo

Department: Systems Engineering and Computer Science

Linear algebra kernels are of fundamental importance to many petroleum reservoir simulators used extensively by the industry. Due to the growth of problems simulated by these systems, the processing time of each kernel became a determinant to the feasibility of simulators. However, as a result of a highly potential parallelism offered by modern CPUs, the parallel implementation of some algebra linear kernels proved to be a promising option to reduce the execution time of these applications. We believe that some parallel models, the dataflow model to be more specific, could take some advantages from the data dependencies that some linear algebra kernels have. This work aims to present a set of parallel linear algebra kernels implemented by applying the dataflow model. To do so, the dataflow virtual machine *Trebuchet* and the dataflow library *Sucuri* were applied. The kernels implementations, as well as the dataflow model, were evaluated using real data extracted from reservoir simulators used by the industry. Results were compared with *OpenMP* and *Intel® Math Kernel Library* implementations. We could note that the low granularity of some kernels added to the overhead of each dataflow environment put a lid on some applications performance. However, the Matrix Vector Product and the Matrix Matrix Product kernels demonstrated to be promising, overcoming the state of art implementation.

Sumário

Lista de Figuras **xi**

Lista de Tabelas **xiv**

1 Introdução **1**

2 Modelo guiado por Fluxo de Dados **3**

2.1 O Modelo *Dataflow* 3

2.2 Ambientes de Programação *Dataflow* 4

2.2.1 Máquina Virtual *Trebuchet* 4

2.2.2 Biblioteca *Sucuri* 6

3 Sistemas Lineares **10**

3.1 Solução de Sistemas Lineares 10

3.1.1 Métodos Iterativos 11

3.2 Precondicionadores 11

3.2.1 Fatoração Incompleta 12

3.2.2 Outros Precondicionadores 12

3.3 Matrizes Esparsas 13

3.3.1 Estrutura e Compactação 13

3.3.1.1 *Compressed Sparse Row* - CSR 13

3.3.1.2 *Block Compressed Sparse Row* - BCSR 14

3.3.1.3 Outras Compactações 15

4 Núcleos de Álgebra Linear Computacional **16**

4.1 Solver Triangular Esparso 16

4.1.1 STS Paralelo 18

4.1.1.1 *Level-Scheduling* 18
4.1.1.2 *Synchronization Sparsification* 21
4.1.2 STS Paralelo guiado por Fluxo de Dados 22
4.2 Multiplicação Matriz Vetor 23
4.2.1 SpMV e GEMV Paralelo 23
4.2.2 SpMV e GEMV Paralelo guiado por Fluxo de Dados 24
4.3 Fatoração LU 24
4.3.1 Determinante Paralelo 25
4.3.2 Determinante Paralela guiado por Fluxo de Dados 26
4.4 Multiplicação de Matrizes 27
4.4.1 GEMM Paralelo 27
4.4.2 GEMM Paralelo guiado por Fluxo de Dados 28

5 Experimentos e Resultados **30**
5.1 Metodologia 30
5.1.1 Ambientes de Execução 30
5.1.2 Conjunto de Dados 31
5.1.2.1 Matrizes Genéricas 31
5.1.2.2 Matrizes de Reservatórios Petrolíferos 32
5.1.3 Métrica para Medição dos Resultados 33
5.2 Avaliação do Núcleo STS Paralelo 33
5.3 Avaliação do Núcleo SpMV Paralelo 35
5.4 Avaliação do Núcleo GEMV Paralelo 36
5.5 Avaliação do Núcleo Determinante Paralelo 38
5.6 Avaliação do Núcleo GEMM Paralelo 39

6 Conclusões e Trabalhos Futuros **46**

Referências Bibliográficas **48**

A Iniciativas - Projeto CENPES **51**
A.1 Análise das Matrizes Esparsas 51
A.2 Impacto de Compactação das Matrizes Esparsas 52
A.3 Técnica de Precisão Mista 52

B Códigos Fonte **54**

Lista de Figuras

2.1 Exemplo de programa *Dataflow*. 4
2.2 Exemplo de código Trebuchet. 5
2.3 Arquitetura TALM. 6
2.4 Arquitetura da biblioteca *Sucuri*. 7
2.5 Exemplo de grafo Sucuri composto por dois subgrafos. 8
2.6 Exemplo de criação de um programa utilizando o nó especializado Fork-Join na *Sucuri*. 9

3.1 Exemplo de compactação da matriz esparsa em formato CSR. 14
3.2 Exemplo de compactação da matriz esparsa em formato BCSR. 15

4.1 Passo 1 do algoritmo Level-Scheduling. 20
4.2 Passo 2 do algoritmo Level-Scheduling. 20
4.3 Passo 3 do algoritmo Level-Scheduling. 21
4.4 Level Scheduling. 21
4.5 Synchronization Sparsification. 21
4.6 Grafo criado pela biblioteca Sucuri. 22
4.7 Exemplo de divisão da matriz em blocos de linhas. 23
4.8 Grafo *dataflow* das aplicações SpMV e GEMV. 24
4.9 Etapas do algoritmo Determinante. 25
4.10 Grafo *dataflow* da aplicação LU. 26
4.11 Exemplo de aplicação do algoritmo *Processor farm*. 28
4.12 Exemplo de grafo da aplicação GEMM utilizando o modelo *dataflow*. 29

5.1 Comparação de performance (*Speedup*) do núcleo STS entre implementação *OpenMP* e *Sucuri* para matrizes tipo *IMPES* no Ambiente 1. 34
5.2 Comparação de performance (*Speedup*) do núcleo STS entre implementação *OpenMP* e *Sucuri* para matrizes tipo *FIM* no Ambiente 1 35
5.3 Comparação de performance (*Speedup*) do núcleo SpMV entre a biblioteca *MKL* da Intel® e a máquina virtual *Trebuchet* no Ambiente 1. 36
5.4 Performance (*Speedup*) do núcleo GEMV utilizando a biblioteca *MKL* da Intel no Ambiente 1. 37
5.5 Performance (*Speedup*) do núcleo GEMV utilizando a biblioteca *Sucuri* no Ambiente 1. 37
5.6 Performance (*Speedup*) do núcleo Determinante implementado em *OpenMP* no Ambiente 1. 38
5.7 Performance (*Speedup*) do núcleo Determinante utilizando a biblioteca *Sucuri* no Ambiente 1. 39
5.8 Performance (*Speedup*) do núcleo GEMM utilizando a biblioteca *MKL* da Intel variando o número de *threads* no Ambiente 1. 41
5.9 Performance (*Speedup*) do núcleo GEMM na *Sucuri* variando o número de *threads* no Ambiente 1. 41
5.10 Performance (*Speedup*) do núcleo GEMM na *Sucuri* variando o número de *cores* no Ambiente 2. 42
5.11 Performance (*Speedup*) do núcleo GEMM na *Sucuri* variando o número de *threads* no Ambiente 2. 43
5.12 Tempo gasto do núcleo GEMM na *Sucuri* variando o percentual de linhas da matriz C transferida para o nó central utilizando o Ambiente 2 e o Cenário 3. 44
5.13 Performance (*Speedup*) do núcleo GEMM na *Sucuri* variando o número de *cores* no Ambiente 2 e Cenário 3. 44

5.14 Comparação de performance (*Speedup*) do núcleo GEMM implementado em Python *MPI* e Sucuri variando o número de *cores* no Ambiente 2. 45

Lista de Tabelas

5.1 Matrizes esparsas reais da coleção da Universidade da Flórida [1]. . . 32
5.2 Matrizes de Reservatórios Petrolíferos com diferentes formulações e tamanhos. 33

Lista de Códigos

4.1 Substituição direta 17
4.2 Substituição reversa 18
B.1 Código em C para substituição direta sem bloco. 54
B.2 Código em C para Substituição Reversa sem bloco. 55
B.3 Código em C para Substituição Direta em bloco. 55
B.4 Código em C para Substituição Reversa em bloco. 56
B.5 Código Sucuri para Substituição Direta sem bloco. 57
B.6 Código Sucuri para Substituição Reversa sem bloco. 57
B.7 Código Sucuri para Multiplicação de Matrizes utilizando o algoritmo *Processor Farm*. 58

Capítulo 1

Introdução

Simular modelos de reservatórios petrolíferos tornou-se uma difícil e complexa tarefa ao longo dos anos. Com o crescimento dos modelos simulados, bem como o número de fenômenos físicos envolvidos, o tempo de processamento passou a ser um ponto crítico para a viabilidade dos simuladores.

Dentre os algoritmos que dominam o tempo de processamento de cada simulação, destacam-se os núcleos de álgebra linear computacional, mais precisamente os envolvidos na resolução de sistemas lineares. Tais sistemas possuem propriedades que dificultam o desempenho dos algoritmos empregados, como por exemplo, a esparsidade e as centenas de milhões de linhas presentes na matriz de coeficientes.

Para explorar o real potencial das CPUs modernas e, consequentemente, melhorar o desempenho dos simuladores, modelos paralelos são utilizados na implementação dos núcleos de álgebra linear. Porém, explorar paralelismo em sistemas onde o número de cores cresce rapidamente pode tornar-se uma tarefa complexa devido gerenciamento das *threads* e a sincronização entre elas.

Trabalhos recentes em modelos de programação paralela mostram que o modelo de execução guiado por fluxo de dados (*Dataflow*)[2] explora o paralelismo das aplicações de forma natural. Instruções são executadas assim que todos os seus operandos de entrada estiverem disponíveis, ou seja, assim que o dado estiver disponível. Tal característica simplifica o desenvolvimento das aplicações paralelas pois é necessário apenas definir as dependências entre pequenos trechos do código.

A biblioteca *Sucuri*[3] e a máquina virtual *Trebuchet*[4] implementam o modelo *dataflow* híbrido, onde pequenas porções de código, que seguem o modelo por fluxo

de controle, são executadas segundo a regra de fluxo de dados. A biblioteca *Sucuri* possibilita a criação de grafos *dataflow* a partir de um conjunto de nós interligados por arestas. Cada nó é associado a uma função e cada aresta indica as dependências de dados para executar a mesma. Já a *Trebuchet*, possibilita a paralelização através de anotações no código. Trechos com paralelismo em potencial são identificados e as anotações transformam o código sequencial em um grafo *dataflow* a ser executado em paralelo.

Este trabalho propõe a aplicação do modelo *dataflow* em implementações paralelas de álgebra linear computacional utilizando as ferramentas *Sucuri* e *Trebuchet*. Os núcleos escolhidos são utilizados durante a resolução sistemas lineares presentes em simulações de reservatório, tais como o solver triangular esparso, as multiplicações matriz-vetor densa e esparsa, o cálculo do determinante e a multiplicação de matrizes densas. Para avaliar tais núcleos, um conjunto de implementações, utilizando as bibliotecas *OpenMP* e *Intel® Math Kernel Library (MKL)*, foi desenvolvido.

O restante deste trabalho está dividido da seguinte forma: *(i)* o Capítulo 2 apresenta o modelo *Dataflow*, bem como os ambientes utilizados para desenvolver as aplicações seguindo este modelo; *(ii)* o Capítulo 3 apresenta as características dos sistemas lineares, os métodos utilizados para solucioná-los, algumas técnicas de precondicionamento e as estruturas de compactação das matrizes esparsas; *(iii)* o Capítulo 4 descreve todos os núcleos de álgebra linear computacional utilizados neste trabalho, bem como a implementação da versão paralela guiada por fluxo de dados de cada um; *(iv)* o Capítulo 5 apresenta os experimentos e resultados obtidos; *(vi)* a conclusão e os trabalhos futuros são discutidos no Capítulo 6.

Capítulo 2

Modelo guiado por Fluxo de Dados

Neste capítulo são explicados os conceitos sobre o modelo *dataflow* e os ambientes de programação paralela utilizados neste trabalho. Inicialmente é apresentado o modelo guiado por fluxo de dados (*dataflow*). Em seguida será feito o detalhamento dos ambientes de programação *dataflow* utilizados.

2.1 O Modelo *Dataflow*

As CPUs atuais são guiadas por fluxo de controle, ou seja, executam uma sequência pré-definida de instruções indicada por um contador de programa. Conforme cada instrução for sendo executada, este contador é incrementado. Tal modelo, chamado de *Von Neumann*, é sequencial em sua essência.

O modelo *Dataflow* [2] surgiu como uma alternativa ao modelo *Von Neumann*, diferenciando-se pelo fato de que as instruções são executadas seguindo o fluxo de dados ao invés do fluxo de instruções. Uma das vantagens desse modelo é que ele expõe o paralelismo das instruções de forma natural.

Porém, devido a incompatibilidade com as linguagens imperativas, bem difundidas naquele período, o modelo *dataflow* acabou caindo em desuso. Com o advento da programação paralela, o modelo *Dataflow* ressurgiu como uma saída para criação de programas paralelos devido ao fato de que tal modelo expõe paralelismo de forma natural, como exemplifica a Figura 2.1.

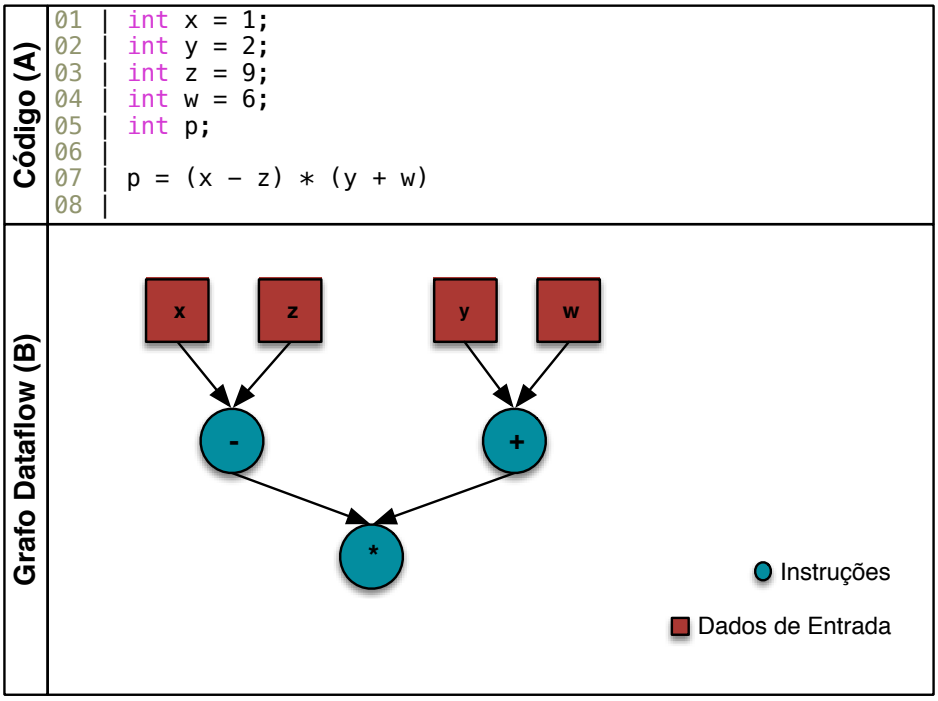


Figura 2.1: Exemplo de programa *Dataflow*. Painel (A) exibe um exemplo de código em alto nível. Já no painel (B), o grafo dataflow do código em (A).

2.2 Ambientes de Programação *Dataflow*

Para implementar os núcleos paralelos de álgebra linear seguindo o modelo *dataflow*, a máquina virtual *Trebuchet*[4] e a biblioteca *Sucuri*[3] foram utilizadas. Tais ambientes baseiam-se no conceito de *dataflow híbrido*, onde os blocos de códigos *Von Neumann* são escalonados segundo o fluxo de dados.

2.2.1 Máquina Virtual *Trebuchet*

A máquina virtual *dataflow Trebuchet* foi desenvolvida com o intuito de possibilitar a paralelização de códigos sequencias através de anotações no código fonte da aplicação. Regiões com potencial paralelismo são identificadas e transformadas em super-instruções, sendo apenas necessário informar os dados de entrada e saída dessas regiões. Com base nessas informações, o compilador Coulliard [4] é capaz de gerar um grafo de fluxo de dados juntamente com o código *C* referente a cada super-instrução.

A Figura 2.2, painel *A*, exibe um exemplo de região de código, em linguagem *C*, a ser paralelizada. No painel *B*, o mesmo trecho de código anotado com diretivas da *Trebuchet*.

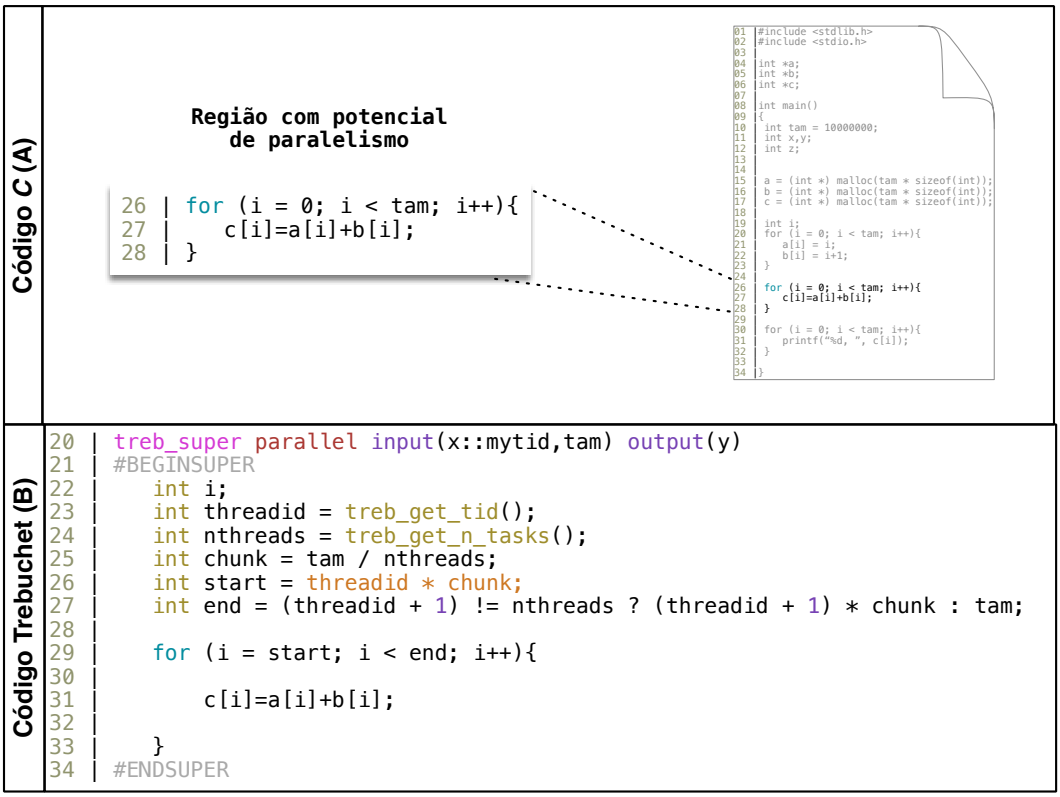


Figura 2.2: Exemplo de código Trebuchet. Painel (A) exibe um exemplo de região de código com potencial de paralelismo. Já no painel (B), o mesmo trecho de código porém com as diretivas de paralelização da *Trebuchet*.

Com o intuito de explorar o potencial do *dataflow* nas arquiteturas correntes, o modelo de execução utilizado na máquina virtual *Trebuchet* é o TALM (*TALM is an Architecture and Language for Multithreading*). Este modelo é flexível, possibilitando a implementação em ambientes com arquiteturas heterogêneas. A Figura 2.3, painel A, exibe a arquitetura do modelo TALM. Ela é constituída por um conjunto de Elementos de Processamento (EPs) que são interligados a uma rede de comunicação. Já o painel B, exibe o conjunto de estruturas compõem uma EP.

Cada EP é responsável por executar um conjunto de instruções. Tais instruções possuem um conjunto de dados de entrada que são armazenados em uma lista. Quando é verificado o casamento de uma instrução, presente na Lista de Instruções, e um dado de entrada, os mesmos são transferidos para a Fila de Prontos. Assim que

uma das EPs estiver disponível, ela irá remover uma entrada da Fila de Prontos para ser processado. Cada EP também possui um *buffer* de comunicação, responsável por receber as mensagens provenientes de outras EPs.

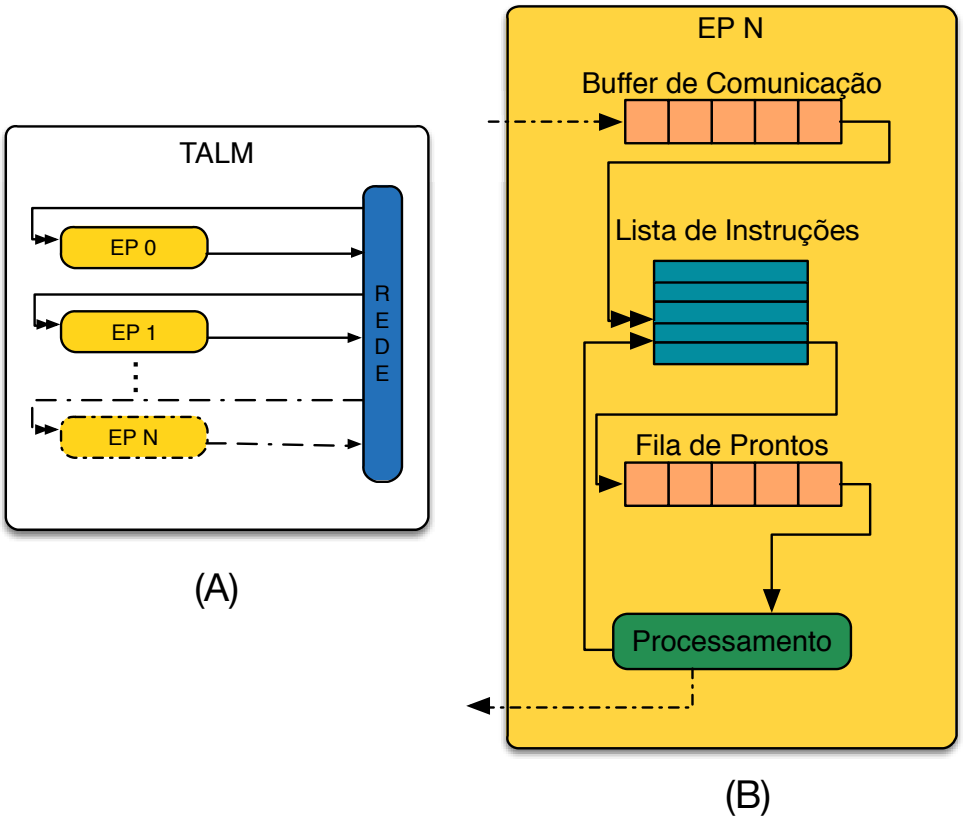


Figura 2.3: Arquitetura TALM. Painel (A) exibe o conjunto de Elementos de Processamento interconectados a uma rede de comunicação. O painel (B) detalha os EPs, exibindo of buffers, listas e filas que compõem cada um.

2.2.2 Biblioteca *Sucuri*

A biblioteca *Sucuri*, escrita em linguagem Python, possibilita a implementação em alto nível de algoritmos paralelos seguindo o modelo *dataflow*. De maneira intuitiva, a *Sucuri* permite a construção de um grafo *dataflow* da aplicação através da criação de nós e arestas.

A *Sucuri* é composta por três estruturas principais descritas abaixo:

Grafo: Estrutura composta por um conjunto de nós e arestas. Cada nó possui:

- Função a ser executada ao receber todos os operandos de entrada;

- Lista de nós destinatários que dependem dos operandos resultantes;
- Lista de operandos recebidos e aguardando casamento;

Escalonador: Responsável pela comunicação entre os nós do *Grafo*, bem como pelo envio de tarefas aos *Trabalhadores*; É composto por uma *Unidade de Casamento* e uma *Fila de Prontos*;

Trabalhador: Responsável por processar as tarefas enviadas pelos escalonadores.

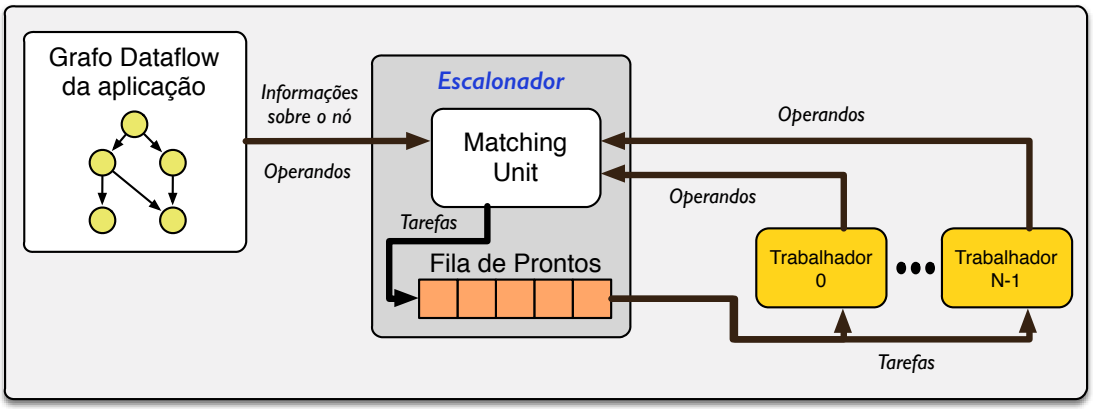


Figura 2.4: Arquitetura da biblioteca *Sucuri*.

Na Figura 2.4 é possível visualizar a interação entre os principais componentes da biblioteca. O *Escalonador* é responsável por alimentar a lista de operandos presente em cada nó do *Grafo*. Caso haja casamento, ou seja, todas as dependências para processar aquele nó tenham sido satisfeitas, uma tarefa será criada e adicionada a *Fila de Prontos*. Quando o *Escalonador* receber um operando contendo a mensagem *REQUEST_TASK* de um *Trabalhador*, uma tarefa da *Fila de Prontos* será removida e enviada para o *Trabalhador* processá-la.

A biblioteca *Sucuri* foi desenvolvida para execução em ambientes heterogêneos. É possível utilizá-la em sistemas multicore ou em um cluster de multicores com o máximo de abstração possível, sendo necessário apenas a alteração da flag *mpi_enabled* para tal.

Para sistemas com cluster de multicores, as estruturas da *Sucuri* são replicadas em cada máquina porém apenas um *Escalonador* fica responsável por realizar o

casamento de operandos e a criação das tarefas. Fica a cargo das réplicas apenas o trabalho de redirecionar as tarefas do *Escalonador* principal para seus respectivos *Trabalhadores*. Tal característica prejudica a escalabilidade das aplicações como vemos nos experimentos apresentados no Capítulo 5.

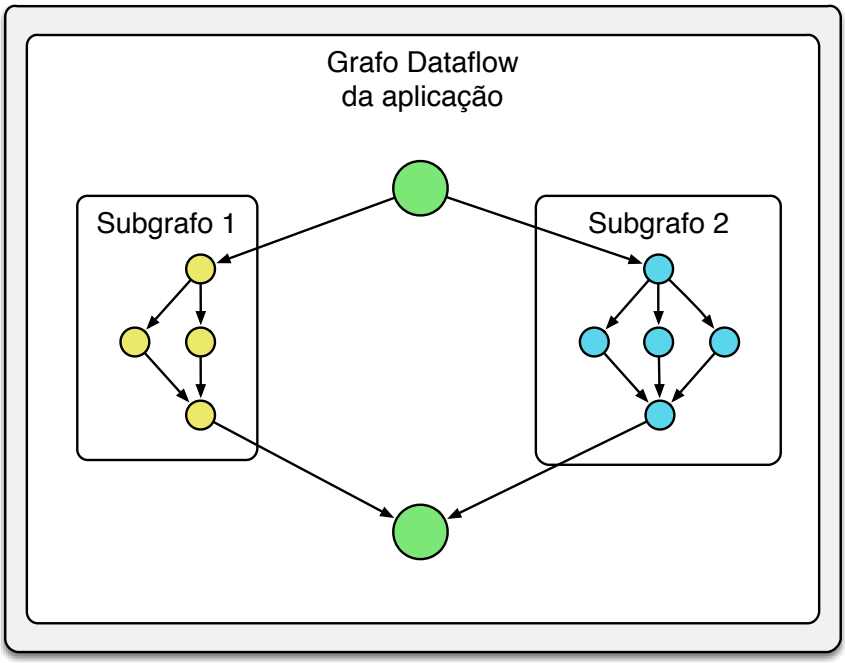
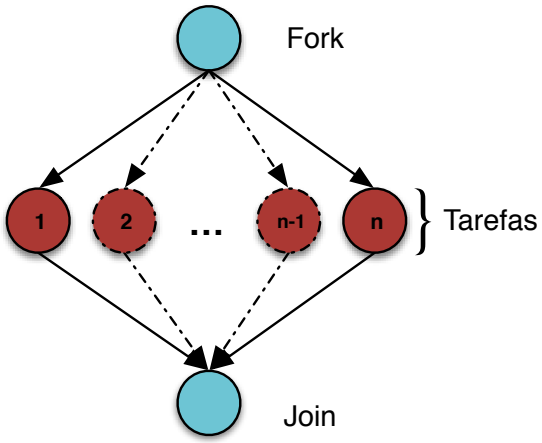


Figura 2.5: Exemplo de grafo Sucuri composto por dois subgrafos.

O *Grafo* da aplicação pode ser composto por múltiplos subgrafos como exibido na Figura 2.5. Cada subgrafo é considerado um nó no grafo principal, tornando possível a criação de nós especializados. Tais nós são capazes de representar modelos paralelos ou até mesmo, executar determinadas funções sem a necessidade de qualquer implementação.

Um exemplo de nó especializado, que implementa o modelo paralelo Fork-Join, pode ser visto na Figura 2.6. Neste caso, as funções *fork*, *join* e *dot_par* são implementadas mas a criação do grafo é feita automaticamente pelo função *ForkJoinGraph()*.

Grafo Dataflow (A)



Criação do Grafo (B)

```
01 | from Sucuri import *
02 |
03 | sucuriGrahp = ContainerGraph()
04 |
05 | sched = Scheduler(sucuriGrahp, nprocs)
06 |
07 | fjGraph = ForkJoinGraph(fork, dot_par, ntasks, join)
08 |
09 | graph.add(fjGraph)
10 |
11 | sched.start()
```

Função dos nós paralelos (C)

```
20 |
21 | def dot_par(id, args):
22 |
23 |   tid = get_tid(id)
24 |
25 |   blockDim = matSize / ntasks
26 |   start = tid * blockDim
27 |   end = matSize if (tid + 1) == ntasks else start
   |                                              + blockDim
28 |
29 |   cij = alpha * np.dot(matrixA[start:end, 0:matSize],
   |                        matrixB[0:matSize, 0:matSize])
30 |
31 |
32 |   return [start, end, cij]
33 |
```

Figura 2.6: Exemplo de criação de um programa utilizando o nó especializado Fork-Join na *Sucuri*. O painel *A* exibe o grafo *dataflow* da aplicação. Os painéis *B* e *C* mostram o código para criação do grafo e a função paralela presente nos nós *Tarefas* respectivamente.

Capítulo 3

Sistemas Lineares

Este capítulo apresenta o problema alvo deste trabalho, bem como alguns conceitos de álgebra linear computacional necessários. Primeiro será feita uma breve descrição sobre solução de sistemas lineares de grande porte, bem como os métodos utilizados para isto. Em seguida, será detalhado o precondicionador ILU, utilizado ao longo deste trabalho. Por final, conceitos sobre matrizes esparsas e formatos de compactação são apresentados.

3.1 Solução de Sistemas Lineares

Um dos principais objetivos deste trabalho é solucionar sistemas lineares de grande porte, cuja forma matricial pode ser representada da seguinte forma

$$Ax = b$$

onde A é uma matriz esparsa quadrada com os coeficientes do conjunto de sistemas resultantes da discretização de equações diferencias parciais (EDPs). x é o vetor de incógnitas do problema alvo e b é o lado direito, ou seja, um vetor de constantes.

Tais sistemas são considerados de grande porte pois a ordem da matriz A tende a ter, atualmente, dezenas ou centenas de milhões de linhas. Devido a este fato, a escolha dos métodos envolvidos para soluciona-los, de forma eficiente, é de extrema importância. Os principais métodos são divididos em dois grupos, os diretos e os iterativos. Os métodos diretos são tidos como exatos pois solucionam os sistemas afim de encontrar a solução exata. Já os métodos iterativos, calculam uma solução aproximada do sistema a partir de uma estimativa inicial. São compostos por um

número de iterações onde a solução final deverá atender alguns critérios de parada, como por exemplo, a redução do resíduo.

O principal fator que difere na escolha entre os métodos diretos e iterativos, na solução de sistemas lineares, é o número de operações. Métodos diretos requerem $O(n * nnz)$ operações, enquanto os métodos iterativos possuem $O(nnz)$, onde nnz é o número de elementos não nulos da matriz. Com o crescimento dos sistemas, e consequentemente a ordem da matriz A, os métodos diretos tornaram-se proibitivos, sendo necessário a abordagem iterativa.

3.1.1 Métodos Iterativos

Dentre os métodos iterativos utilizados para solucionar sistemas lineares de grande porte, destacam-se os métodos baseados na projeção em espaços de Krylov [5]. Devido as boas propriedades numéricas e computacionais, tais métodos são, atualmente, utilizados em larga escala pela indústria. Neste grupo de métodos, destacam-se:

- *Gradientes Conjugados* (CG) para matrizes simétricas, positivas e definidas;
- *Generalized Minimum Residual* (GMRes) para matrizes não-simétricas.

A eficiência desses métodos depende do problema alvo e de outros fatores, tais como, o precondicionador utilizado.

3.2 Precondicionadores

O precondicionamento é um fator de extrema importância ao solucionar sistema lineares de forma iterativa. Tais técnicas afetam não só a performance dos métodos de Krylov mas também sua viabilidade, pois a número de iterações até a solução convergir está ligada diretamente com a técnica de precondicionamento empregada. De acordo com Saad [5], toma-se como precondicionamento qualquer transformação feita no sistema original cujo sistema transformado é equivalente porém mais simples.

Considere a seguinte transformação

$$M^{-1}Ax = M^{-1}b$$

onde M é uma matriz não singular. Tal transformação, considerada um precondicionamento pela esquerda, não altera os valores do vetor solução x, tornando possível a aplicação de métodos iterativos no sistema precondicionado ao invés do original.

De acordo com Saad[5], para que o operador M seja considerado eficiente, as seguinte regras contraditórias devem ser levadas em consideração:

- A aplicação de M tem que ser eficiente a ponto de superar o custo de sua construção;
- M^{-1} tem que ser aproximar de A^{-1} a ponto do resultado da operação AM^{-1} estar próximo da matriz Identidade.

3.2.1 Fatoração Incompleta

Um dos precondicionadores utilizados durante a solução iterativa de sistemas lineares é a fatoração incompleta da matriz A. Considerando a fatoração completa a seguir

$$A = LU$$

na fatoração incompleta, os fatores $\widetilde{L}$ e $\widetilde{U}$ são aproximações dos fatores L e U respectivamente [5]. Tal sistema passa a ser considerado da seguinte forma

$$M = \widetilde{L}\widetilde{U}$$

onde M é o precondicionador utilizado. Quando os elementos de M coincidem com os de A, segundo a regra $m_{ij} = a_{ij} \; \forall \; a_{ij} \neq 0$, a fatoração incompleta é considerada de nível zero.

3.2.2 Outros Precondicionadores

Outros precondicionadores, utilizados atualmente, como Inversa Aproximada e a Decomposição de Domínios possuem abordagem diferentes da Fatoração Incompleta. No caso da Inversa Aproximada, a matriz esparsa M é uma aproximação de A^{-1}. Já no caso da Decomposição de Domínios, a abordagem *dividir para conquistar* é utilizada. Subproblemas são criados a partir do problema maior e os mesmos são resolvidos paralelamente.

3.3 Matrizes Esparsas

Matrizes esparsas são matrizes cujo o número de elementos não nulos é relativamente pequeno quando comparado ao número total de elementos da matriz. Tal esparsidade por surgir em diversas aplicações práticas. Neste caso, tal esparsidade ocorre devido a discretização de equações diferenciais parciais (EDPs).

3.3.1 Estrutura e Compactação

Devido ao grau de esparsidade das matrizes resultantes da discretização de equações diferenciais parciais (EDPs), diversos formatos de compactação foram criados para essas estruturas. Seria inviável o armazenamento de todos os não-zeros, bem como a computação dos mesmos.

Saad [6] apresenta um conjunto de formatos, cada qual aplicado a necessidade de um determinado problema. Dentre esse conjunto de formatos, destacam-se o *Compressed Sparse Row* (CSR), o *Block Compressed Sparse Row* (BCSR), o *Compressed Sparse Column* (CSC) e o *Coordinate* (COO). A escolha do tipo de compactação pode impactar drasticamente na performance da aplicação, pois o padrão de acesso e a quantidade de bytes necessários para se acessar um determinado valor na matriz influenciam diretamente nesta questão. Linguagens como *C*, que possuem armazenamento de *arrays* por linhas contíguas, tendem a tirar melhor proveito dos formatos tipo CSR. Já para linguagens como *Fortran*, cujo armazenamento em memória é feito por coluna, a compactação CSC é a mais indicada.

3.3.1.1 *Compressed Sparse Row* - CSR

O formato CSR compacta a matriz esparsa por linhas, ou seja, os elementos da matriz são armazenados seguindo a ordem que aparecem nas linhas. Para isso, três estruturas do tipo *array* são utilizados, dois para indicar a posição e um para indicar o valor do elemento.

A Figura 3.1 exibe um exemplo de matriz esparsa juntamente com suas estruturas que compõem a compactação CSR. O primeiro vetor, chamado de *row_prt*, possui tamanho *n+1*, onde *n* é o número de linhas da matriz esparsa. Os valores armazenados indicam o número de elementos não-zero em uma determinada linha,

sendo este valor calculado através da equação

$$e_i = row\_prt[i+1] - row\_prt[i]$$

com i indicando a linha desejada e e_i o número de elementos naquela linha. As estruturas *col_prt* e *val* possuem tamanho igual ao número de não-zeros da matriz. A primeira armazena um valor do tipo inteiro que representa a coluna em que aquele elemento está na matriz. Já a estrutura *val*, armazena os valores dos elementos da matriz.

$$\begin{pmatrix} 1 & 5.1 & & & 2.3 & \\ & 2 & 3.4 & & & 2.9 \\ 9.2 & & 3 & 8.8 & & \\ & & & 4 & & 7.1 \\ & & & 4.5 & 5 & \\ 1.5 & & & & 9.1 & 6 \end{pmatrix}$$

row_prt	0	3	7	11	15	18	22									
col_prt	0	1	4	1	2	5	0	2	3	3	5	3	4	0	4	5
val	1	5.1	2.3	2	3.4	2.9	9.2	3	8.8	4	7.1	4.5	5	1.5	9.1	6

Figura 3.1: Exemplo de compactação da matriz esparsa em formato CSR.

3.3.1.2 *Block Compressed Sparse Row* - BCSR

O formato BCSR possui compactação equivalente ao CSR. Porém, cada elemento da matriz original passa a ser considerado uma submatriz densa de ordem m chamados de blocos. Os ponteiros *row_prt* e *col_prt* não indicam mais elementos e sim a localização dos blocos. A estrutura *val* armazena os valores de cada bloco, organizando em sequências de linhas contíguas. Seu tamanho passa a ser calculado da seguinte forma:

$$val\_size = nnz * m^2$$

onde *nnz* é o número de blocos não-zeros e m a ordem dos blocos.

A Figura 3.2 mostra um exemplo de compactação no formato BCSR. Nota-se que as estruturas são similares ao formato CSR, com exceção da estrutura *val*, que passa a ter m^2 vezes mais entradas.

$$\mathrm{AA} = \begin{pmatrix} A_{11} & A_{12} & & & A_{15} & \\ & A_{22} & A_{23} & & & A_{26} \\ A_{31} & & A_{33} & A_{34} & & \\ & & & A_{44} & & A_{46} \\ & & & A_{54} & A_{55} & \\ A_{61} & & & & A_{65} & A_{66} \end{pmatrix}$$

$$A_{ii} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$$

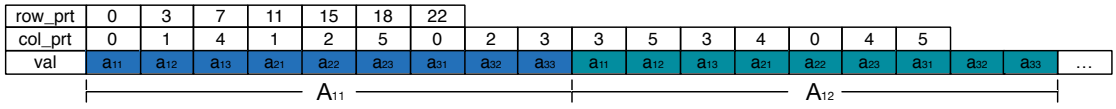


Figura 3.2: Exemplo de compactação da matriz esparsa em formato BCSR. A matriz AA é composta por submatrizes, chamadas de blocos, do tipo A_{ii}. Os valores de cada bloco são armazenados no array *val* por linha e de forma contígua.

3.3.1.3 Outras Compactações

Outros formatos foram criados com o intuito de otimizar o acesso a um determinado padrão na matriz [6]. O formato DIA, por exemplo, armazena a matriz esparsa por diagonal. Já o formato COO, que armazena os índices linha e coluna de cada elemento, possui um formato mais intuitivo, porém ocupa um maior espaço em memória. O CSC, por outro lado, é um CSR otimizado para acesso as colunas da matriz.

Capítulo 4

Núcleos de Álgebra Linear Computacional

Neste capítulo são apresentados os núcleos de álgebra linear computacional implementados e avaliados neste trabalho. Inicialmente os conceitos sobre cada núcleo são discutidos, bem como sua implementação sequencial. Em seguida, o estado da arte em modelos e implementações paralelas são introduzidas para cada núcleo. Por fim, o modelo e implementação guiado por fluxo de dados é apresentado.

4.1 Solver Triangular Esparso

Um dos núcleos mais importantes na aplicação do método de Krylov 3.1.1 é o solver triangular esparso (STS). Considerando a aplicação da fatorização ILU como precondicionador, o sistema a ser resolvido passa a ser:

$$U^{-1}L^{-1}x = b \tag{4.1}$$

onde x e b são vetores densos e L e U são matrizes triangulares esparsas não singulares inferior e superior respectivamente.

O sistema 4.1 pode ser decomposto em outros dois sistemas (4.2) e solucionados através dos métodos de substituição direta e reversa.

$$\begin{cases} Ly = b \\ y = Ux \end{cases} \tag{4.2}$$

Os algoritmos em C para o método de substituição direta e reversa, utilizando a variação popular da compressão CSR para a matriz esparsa [7], são apresentados no Algoritmo 4.1 e 4.2.

```
1  void usolve(csr *U, dvec *y, dvec *x){
2    int i, j;
3
4    for (i = U->n - 1; i >= 0; --i){
5      x->x[i] = y->x[i];
6
7      for (j = U->i[i+1]; j > U->i[i]; --j){
8        if(i == U->j[j - 1]){
9          x->x[i] /= U->x[j - 1];
10         break;
11       }else{
12         x->x[i] -= U->x[j - 1] * x->x[U->j[j - 1]];
13       }
14     }
15   }
16 }
```

Algoritmo 4.1: Substituição direta

```
void lsolve(csr *L, dvec *x, dvec *y){
  int i, j;

  for (i = 0; i< L->n; i++){
    y->x[i] = x->x[i];

    for (j = L->i[i]; j < L->i[i+1]; j++){
      if(i == L->j[j]){
        y->x[i] /= L->x[j];
        break;
      }else{
        y->x[i] -= L->x[j] * y->x[L->j[j]];
      }
    }
  }
}
```

Algoritmo 4.2: Substituição reversa

4.1.1 STS Paralelo

O algoritmo do STS, utilizando os métodos de substituição, é sequencial em sua essência. Porém, no caso onde os fatores L e U, resultantes da fatoração incompleta, são esparsos, a paralelização do algoritmo se torna viável. O padrão de esparsidade, ou seja, a distribuição de elementos não-zeros da matriz, é explorado a fim de transformar as matrizes em grafos de dependência.

4.1.1.1 *Level-Scheduling*

O algoritmo *Level-Scheduling*, apresentado por Naumov em [8], utiliza a estratégia de representar as matrizes resultantes de uma fatorização ILU em digrafos acíclicos (*DAGs*). Nestes digrafos, cada nó representa uma linha da matriz e as arestas as dependências entre cada linha.

O *Level-Scheduling* é dividido em duas fases, análise e solução. Na primeira fase, o grafo de dependência é criado a partir do padrão de esparsidade da matriz. Bem como o agrupamento dos nós em níveis, considerando as arestas de dependência. A

segunda fase é uma iteração sobre os níveis criados, onde os dados de cada nó são computados paralelamente.

Para solvers iterativos, a fase de análise é executada apenas uma vez enquanto que a fase de solução é aplicada dezenas de vezes para a mesma matriz. Mesmo que a fase de análise seja custosa, o número alto de iterações da fase de solução irá amortizar o tempo gasto para criar o grafo de dependências.

Considere o seguinte sistema linear abaixo:

$$\begin{pmatrix} 1 & & & & & \\ & 2 & & & & \\ x & & 3 & & & \\ x & x & & 4 & & \\ & x & & & 5 & \\ & & x & x & & 6 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \end{pmatrix} \tag{4.3}$$

Durante a fase de análise, o algoritmo irá percorrer as linhas da matrix L executando os seguintes passos:

Passo 1: Procurar por nós que não possuem dependência, ou seja, nós com não-zeros apenas na diagonal. Nós 1 e 2 são adicionados são adicionados ao primeiro nível do DAG;

Passo 2: Percorrer novamente a matriz considerando as dependências dos nós 1 e 2 resolvidas. Neste caso, os nós 3, 4 e 5 são adicionados ao nível dois e suas dependências mapeadas pelas arestas no grafo;

Passo 3: Adicionar os nó restante ao último nível, bem como as arestas de dependência no grafo.

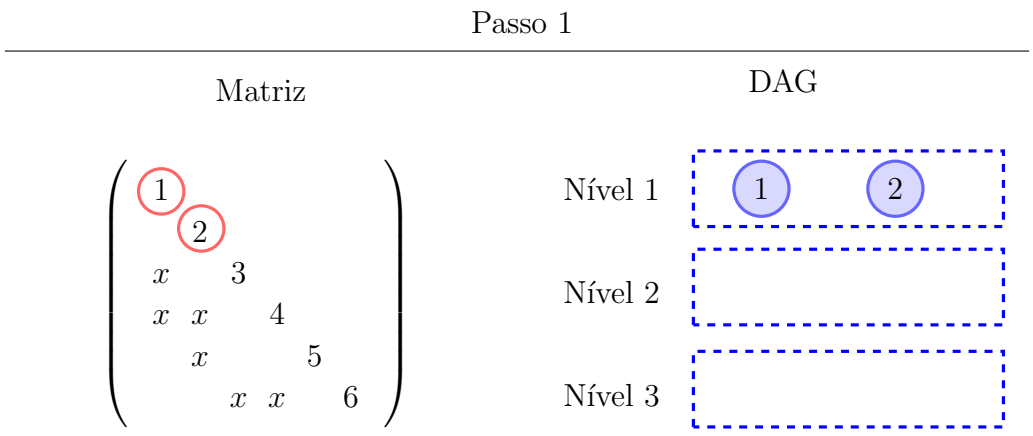


Figura 4.1: Primeiro passo do algoritmo Level-Scheduling para o sistema linear (4.3).

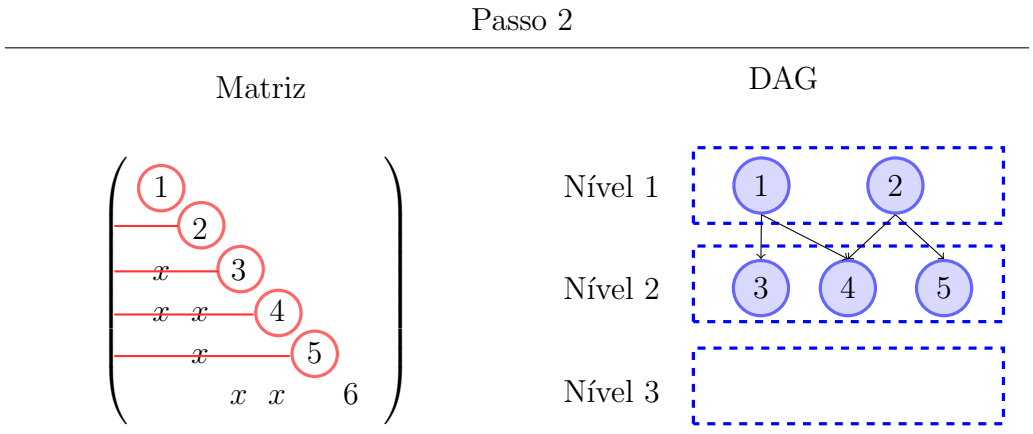


Figura 4.2: Segundo passo do algoritmo Level-Scheduling para o sistema linear (4.3).

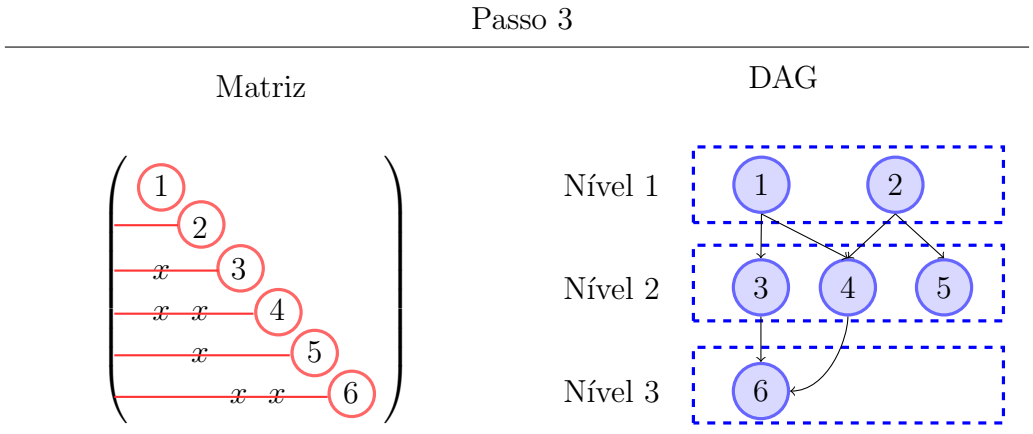


Figura 4.3: Terceiro e último passo do algoritmo Level-Scheduling para o sistema linear (4.3).

4.1.1.2 *Synchronization Sparsification*

O método *Synchronization Sparsification*, apresentador por J.Park em [9], aprimora a técnica *Level-Scheduling*. As barreiras implícitas entre os níveis são removidas e os nós são agrupados em threads a fim de tornar o grão das tarefas mais grosso.

Considere os exemplos abaixo:

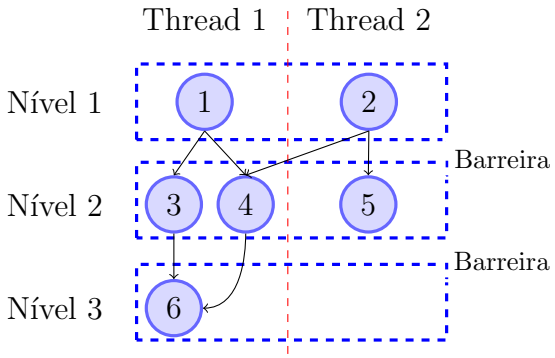


Figura 4.4: Level Scheduling.

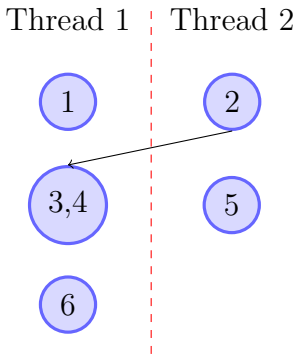


Figura 4.5: Synchronization Sparsification.

Na Figura 4.4 o *DAG* criado pelo algoritmo level-scheduling apresenta pontos de sincronização (barreiras) implícitos entre os níveis 1 e 2. Mesmo que o nó 2

seja processado mais rápido que o nó 1, para processar o nó 5 a *Thread* 2 terá que aguardar a *Thread* 1 para prosseguir.

Na Figura 4.5, o *DAG* criado pelo algoritmo *synchronization sparsification* não possui barreiras e a comunicação entre as threads é feita ponto a ponto. Ou seja, assim que a *Thread* 2 processar o nó 2, uma mensagem será enviada para *Thread* 1 liberando a execução dos nós 3 e 4.

Ainda na Figura 4.5, é possível notar que as arestas de dependência entres os nós da mesma threads são removidas, já que todo processamento naquele contexto será sequencial. Com isso, o custo de comunicação entre os nós é reduzido drasticamente.

4.1.2 STS Paralelo guiado por Fluxo de Dados

O modelo guiado por fluxo de dados (*dataflow*) baseia-se em distribuir o dado a ser computado em nós que compõem um grafo. As dependências para computar um dado nó são representadas pelas as arestas deste grafo. Assim que um nó é computado, uma mensagem é enviada para os nós filhos a fim de liberar a computação dos mesmos.

Os algoritmos *level-scheduling* e *synchronization sparsification* apresentados em 4.1.1.1 e 4.1.1.2 possuem um conceito similar ao modelo Dataflow, já que tratam o problema como um grafo de dependências. Porém, as barreiras implícitas do *level-scheduling* e a remoção das arestas presente no *synchronization sparsification* não são aplicadas na implementação deste modelo.

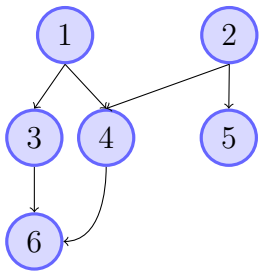


Figura 4.6: Grafo criado pela biblioteca Sucuri.

O grafo criado pela biblioteca Sucuri [3] é apresentado na Figura 4.6 para o sistema linear (4.3). Como dito anteriormente, não há níveis com pontos de sincronização, bem como remoção de arestas redundantes. Cada *thread* disponível irá

executar um nó com estado "pronto", ou seja, com todas as dependências resolvidas.

Neste exemplo é possível notar que no máximo 3 *threads* processam nós paralelamente. Mesmo número que a técnica *level-scheduling*, porém sem barreiras e uma a mais que a técnica *synchronization sparsification*.

4.2 Multiplicação Matriz Vetor

A Multiplicação Matriz Vetor (MMV) é uma operação importante em problemas de álgebra linear computacional. Segundo Bell [10], *"ela representa o custo computacional dominante em diversos métodos iterativos para solução de sistemas lineares em larga escala"*. Tal operação é representada pela seguinte equação:

$$y \leftarrow Ax + y \tag{4.4}$$

onde x e y são vetores densos e A uma matriz densa ($GEMV$) ou esparsa ($SpMV$).

O núcleo MMV é considerado *memory-bound* por possuir uma taxa de 2 *flops* por não zero na matriz A. Tal fato limita o desempenho máximo do algoritmo pela banda máxima da memória principal disponível no ambiente de execução.

4.2.1 SpMV e GEMV Paralelo

A método mais difundido para paralelização dos núcleos SpMV e GEMV é através da divisão da matriz de entrada em blocos de linhas. O número de divisões é baseado no número de threads ou processos disponíveis no ambiente, a fim de tornar o tamanho das tarefas o mais balanceado possível.

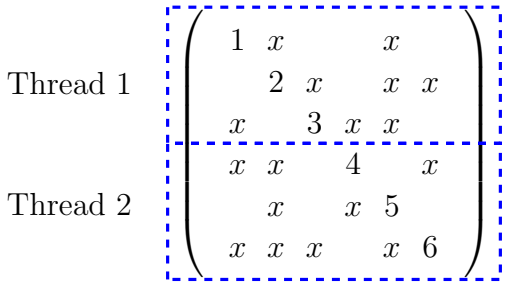


Figura 4.7: Divisão da matriz em blocos de linhas baseado no número de threads disponíveis.

4.2.2 SpMV e GEMV Paralelo guiado por Fluxo de Dados

A implementação *dataflow* consistem em utilizar o conceito de divisão da matriz em blocos de linhas. Cada bloco de linhas é considerado um nó no grafo principal que é executado paralelamente.

A Figura 4.8 exibe o grafo *dataflow* da aplicação com base no exemplo da Figura 4.7. Nota-se que o padrão utilizado foi o Fork-Join, onde o primeiro nó (Fork) lê os dados da matriz A de entrada, bem como os vetores x e y. Em seguida, n nós processam paralelamente os dados referentes ao bloco da matriz que lhe foi entregue pelo nó *Fork*. Por fim, o nó *Join* recebe o bloco do vetor y processado por cada nó e exibe o resultado.

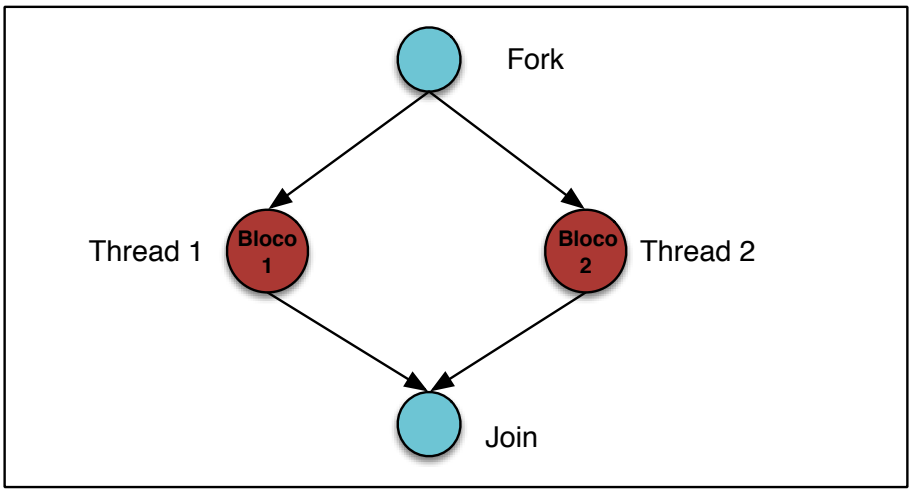


Figura 4.8: Grafo *dataflow* das aplicações SpMV e GEMV com base no exemplo da Figura 4.7.

4.3 Fatoração LU

O cálculo do determinante possui certa relevância quando se deseja determinar algumas propriedades da matriz, como por exemplo a inversibilidade da mesma. Uma das maneiras de se calcular tal valor é através da decomposição LU. Tal operação é uma variação do cálculo do determinante com o intuito de otimizar a operação. Vamos considerar a seguinte equação:

$$det(A) = det(L) * det(U) \tag{4.5}$$

onde A é uma matriz densa e L e U são as matrizes inferior e superior respectivamente.

O determinante de matrizes triangular são calculados a partir do produto dos elementos de suas diagonais. Como toda diagonal de L é composta por elementos com valor igual a 1, o $det(A)$ passa a ser calculado da seguinte forma:

$$det(A) = \prod_{i=1}^{n} l_{ii} * \prod_{i=1}^{n} u_{ii} \tag{4.6}$$

$$det(A) = \prod_{i=1}^{n} u_{ii} \tag{4.7}$$

Portanto, para calcular o determinante de A, basta apenas calcular os valores na diagonal da matriz U e multiplica-los.

4.3.1 Determinante Paralelo

O algoritmo sequencial, para calcular o determinante de uma matriz, é composto por *k-1* etapas, onde k é a ordem da matriz A. Durante cada etapa, um conjunto de valores é atualizado com base nas linhas e colunas indicadas por k.

$$\mathrm{k} = 1 \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$$

$$\mathrm{k} = 2 \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$$

$$\mathrm{k} = 3 \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$$

Figura 4.9: Etapas do algoritmo Determinante com decomposição LU na matriz original.

A paralelização do algoritmo é feita dentro de cada etapa k. A submatriz a ser atualizada é dividida em blocos de linhas e associada a uma thread para ser

processada. A Figura 4.9 exibe um exemplo com 3 etapas onde as submatrizes a serem atualizadas e paralelizadas são destacadas. A cada etapa do algoritmo, a ordem da submatriz a ser processada é reduzida até restar apenas uma submatriz de ordem 1.

4.3.2 Determinante Paralela guiado por Fluxo de Dados

O algoritmo paralelo guiado por fluxo de dados segue o mesmo padrão das versões paralelas em *OpenMP*. A Figura 4.10, painel A, exibe o grafo *dataflow* criado na *Sucuri* para a aplicação LU paralela. Tal grafo é composto por um conjunto de subgrafos com o padrão fork-join interligados, cada um representando uma etapa k do algoritmo paralelo. O painel B detalha um subgrafo específico do grafo original onde threads são criadas para processar um conjunto linhas da matriz em paralelo.

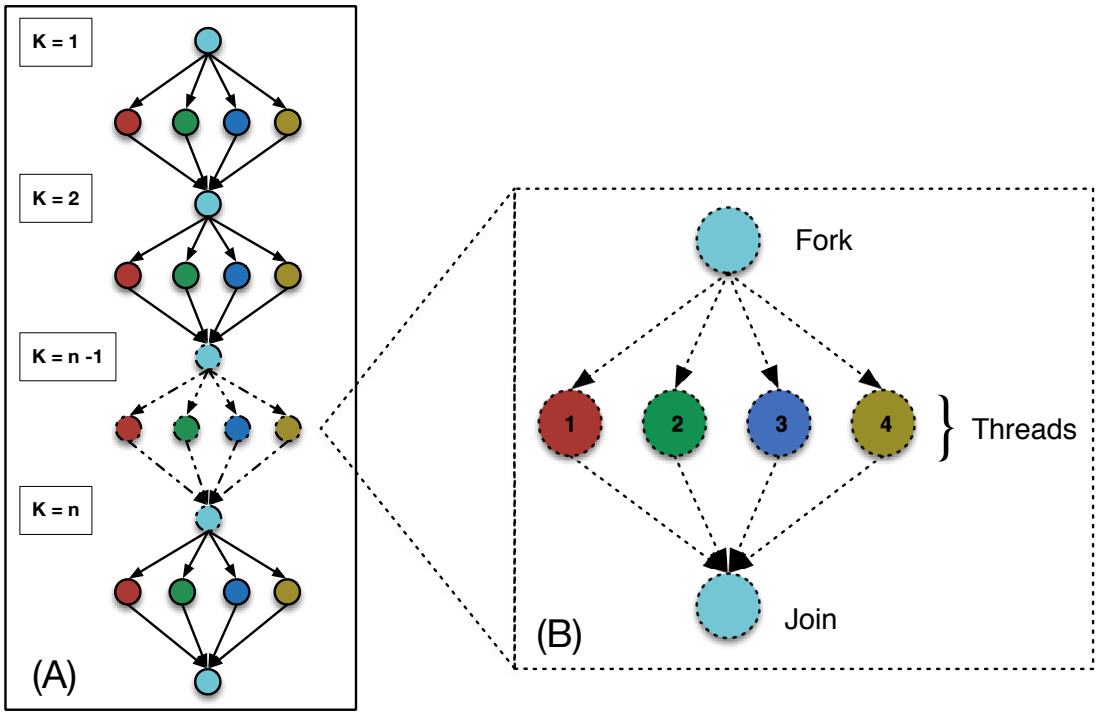


Figura 4.10: O grafo *dataflow* da aplicação LU é apresentado no painel A, bem como o detalhamento de um subgrafo no painel B.

4.4 Multiplicação de Matrizes

Dentre os núcleos apresentados neste trabalho, o Multiplicação de Matrizes (GEMM) é o que possui maior complexidade em termos de número de operações. Por ser aplicado em problemas de diversas áreas, este núcleo é tido por Demmel [11] como "algoritmo mais estudado em computação de alto desempenho".

A operação realizada por tal núcleo é definida por:

$$C = \alpha * (A) * B + \beta * C \tag{4.8}$$

onde A, B e C são matrizes densas de ordem n e α e β são escalares.

Diferente das outras operações apresentadas, o GEMM é considerado *cpu-bound*, ou seja, a aplicação despende mais tempo processando um determinado dado do que transferindo ele entre a memória e o processador.

4.4.1 GEMM Paralelo

O algoritmo GEMM possui diversas implementações paralelas, dentre elas podemos ressaltar os seguintes:

Block: As matrizes A, B e C, de ordem n, são decompostas em submatrizes quadradas, de ordem n/m, como mostra a Equação 4.9. Cada elemento da matriz em blocos C é então processado conforme a Equação 4.10, resultando na matriz presente na Equação 4.11.

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \tag{4.9}$$

$$C_{ij} = \sum_{k=1,2} A_{ik} * B_{kj} \tag{4.10}$$

$$\begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix} \tag{4.11}$$

Processor farm: As matrizes C e A, de ordem n, são decompostas em m blocos de linhas. Cada processo/thread recebe uma réplica da B e um conjunto de blocos linha das matrizes C e A para serem processados.

A Figura 4.11 exibe um exemplo da divisão das matrizes C e A em quatro blocos linha que serão processados em quatro threads/processos paralelamente. A matriz B será replicada em cada processo.

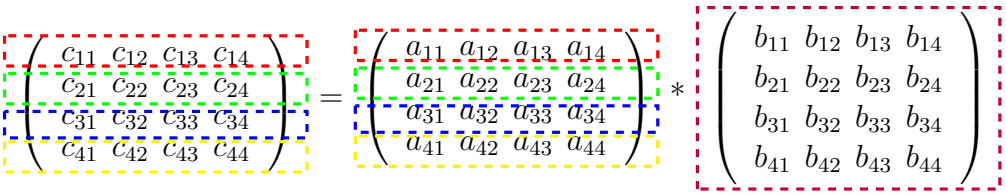


Figura 4.11: Exemplo de aplicação do algoritmo *Processor farm* com a divisão das matrizes C e A em blocos de linhas.

4.4.2 GEMM Paralelo guiado por Fluxo de Dados

Para paralelização do núcleo GEMM, seguindo o modelo *dataflow*, foi utilizado o algoritmo *processor farm* citado anteriormente. Neste caso, cada bloco de linhas das matrizes A e C são considerados um nó no grafo. Os nós *Fork* e *Join* são responsáveis pela divisão e pela junção dos blocos respectivamente. Para o caso onde o ambiente de execução possui apenas memória compartilhada, as matrizes A e B são únicas e compartilhadas entre as threads. Em ambientes com memória distribuída (cluster), as matrizes A e B são replicadas em cada nó do cluster.

A Figura 4.12 exibe o grafo *dataflow* da aplicação GEMM para o exemplo na Figura 4.11.

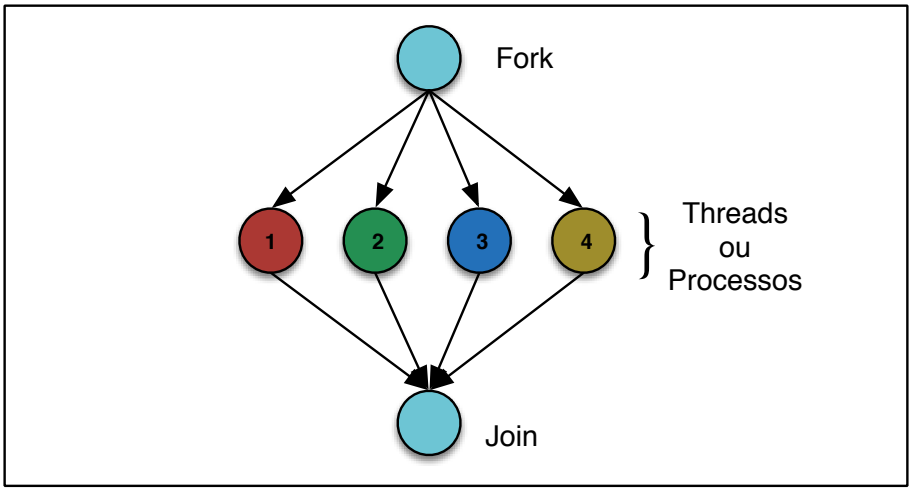


Figura 4.12: Exemplo de grafo da aplicação GEMM utilizando o modelo *dataflow*.

Capítulo 5

Experimentos e Resultados

5.1 Metodologia

Para avaliar o modelo *dataflow* aplicado nos núcleos STS, SpMV, GEMV, GEMM e LU, um conjunto experimentos foi realizado com diferentes dados de entrada e ambientes de execução . Medidas como tempo de processamento e *speedup* foram utilizados como base para comparação entre implementações *OpenMP* [12], *Intel® MKL*[13] e *Dataflow*.

Nas Subseção 5.1.1 os ambientes de execução utilizados nos experimentos serão detalhados, bem como o conjunto de dados na Subseção 5.1.2. A métrica utilizada para avaliação do desempenho dos resultados é descrita na Subseção 5.1.3.

5.1.1 Ambientes de Execução

Para avaliação dos núcleos paralelos, dois ambientes de execução foram utilizados. No primeiro, composto por apenas uma máquina, foram executados os experimentos de memória compartilhada. Já o segundo ambiente, composto por um conjunto de 36 máquinas, foi utilizado nos experimentos de memória distribuída. As configurações de cada ambiente são detalhadas abaixo.

- Ambiente 1:
  - Intel® Xeon(R) CPU E5-2650 v2 @ 2.60GHz (max turbo frequency 2.80GHz);

  - 16 cores (hyper-threading desativada), 20MB LLC, 64GB RAM (ECC ligada);
  - Linux gnu 3.16-2-amd64 #1 SMP Debian 3.16.3-2 (2014-09-20) x86_64 GNU/Linux;
  - Compilador GCC - Versão 4.9.2 (with -O3 flag);
  - Compilador Intel® - Versão Composer XE 2013 14.0.3 20140422 (with -O3 flag);

- Ambiente 2:
  Cluster [1] composto por 32 nós interligados por uma rede 100 Megabit. Cada nó possui a seguinte configuração:
  - Intel® Core(TM) i5-3330 CPU @ 3.00GHz
  - 4 cores, 6MB LLC, 8GB RAM;
  - Linux 3.13.0-39-generic #1 SMP Ubuntu 3.13.0-39-generic x86_64 GNU/-Linux;
  - Compilador GCC - Versão 4.8.4 (with -O3 flag);

5.1.2 Conjunto de Dados

O conjunto de dados utilizados nos experimentos *SpMV* e *STS* são apresentados nesta sub-seção. Tais experimentos diferem dos outros no fato de que as matrizes utilizadas são extraídas de problemas reais. Já nos demais experimentos, o conjunto de dados é composto por matrizes sintéticas criadas randomicamente. O formato de compactação utilizado nas matrizes esparsas foi o *Compressed Sparse Row (CSR).*

5.1.2.1 Matrizes Genéricas

Para medir a performance do núcleo *dataflow SpMV*, um conjunto de matrizes esparsas da coleção da Universidade da Flórida [1] foi selecionado. Tais matrizes, detalhadas na Tabela 5.1, possuem diferentes tamanhos, padrões de esparsidade e

[1] É importante ressaltar que o Ambiente 2 não é um cluster profissional e sim um conjunto de computadores com propósito didático. A rede utilizada neste ambiente não foi projetada com o intuito de alcançar melhor performance.

estrutura e são provenientes de diferentes tipos de problemas, tais como, análise de DNA, dinâmica de fluidos e elementos finitos.

Grupo	Nome	Linhas/Colunas	Não zeros	Precisão	Estrutura
Rudnyi	water_tank	60.740	2.035.281	Dupla	Não simétrica
Botonakis	FEM_3D_thermal2	147.900	3.489.300	Dupla	Não simétrica
PARSEC	Ga41As41H72	268.096	18.488.476	Dupla	Simétrica
Freescale	circuit5M_dc	3.523.317	14.865.409	Dupla	Não simétrica

Tabela 5.1: Matrizes esparsas reais da coleção da Universidade da Flórida [1].

5.1.2.2 Matrizes de Reservatórios Petrolíferos

Uma simulação de reservatório petrolífero é composta por uma sequência de iterações não lineares. Cada iteração não linear é composta por outra sequência de iterações lineares, onde um conjunto de sistemas é solucionado. Esses conjuntos de sistemas, resultantes da discretização de equações diferenciais parciais (EDPs), formam as matrizes esparsas utilizadas neste trabalho. É importante ressaltar que tais matrizes são referentes aos piores casos em todas as simulações, ou seja, às iterações lineares onde o simulador despendeu mais tempo para solucioná-las.

Os métodos para solução das EDPs, resultantes do cálculo do escoamento bifásico, utilizados durante a simulação foram o *IMPES (implicit pressure, explicit saturation)* e o *FIM (fully implicit method).* A escolha de tais métodos implica em um grande impacto na estrutura das matrizes pois o número de componentes (pressão, saturação, etc.) a ser solucionado muda. Para o caso *IMPES*, a matriz não possui estrutura em blocos, ou seja, cada elemento representa apenas uma variável de dupla precisão. Já no método FIM, cada elemento da matriz é representado por blocos densos de tamanho variável.

A Tabela 5.2 exibe o nome, número de linhas/colunas, ordem/tamanho dos blocos, número de não zeros (número de blocos para o caso FIM) e o método utilizado para solucionar as EDPs.

	Propriedades			
Nome	#Linhas/Colunas	Ordem dos Blocos	#Não zeros	Método
unit 10	66.077	4	3,2e6	FIM
unit 12	66.077	1	4,2e5	IMPES
unit 27	408.865	1	2,7e6	IMPES
fit 8	182.558	4	9,9e6	FIM
fit 11	55.380	4	6,5e5	FIM

Tabela 5.2: Matrizes de Reservatórios Petrolíferos com diferentes formulações e tamanhos.

5.1.3 Métrica para Medição dos Resultados

A métrica utilizada para medição de performance dos resultados deste trabalho é o *Speedup* real. Tal medida é calculada com base no tempo sequencial e paralelo das aplicações como mostra a Equação 5.1.

$$Speedup = \frac{t_{sequencial}}{t_{paralelo}} \tag{5.1}$$

onde $t_{sequencial}$ e $t_{paralelo}$ são os tempos de execução das aplicações sequenciais e paralelas respectivamente.

Versões em *OpenMP* e *Intel® MKL* foram desenvolvidas para comparação entre as implementações *dataflow* de cada núcleo apresentado no Capítulo 4. Os *speedups* são calculados com base em implementações sequencias utilizando a mesma linguagem da versão paralela, ou seja, linguagem *C* para versão *OpenMP* e *Intel® MKL* e linguagem *Python* para versão *Sucuri*.

5.2 Avaliação do Núcleo STS Paralelo

O núcleo STS foi desenvolvido utilizando as bibliotecas *Sucuri* e *OpenMP*. As duas implementações foram avaliadas com os métodos *IMPES* e *FIM* descritos na Subseção 5.1.2.2.

A Figura 5.1 mostra os resultados obtidos para o solver triangular esparso em matrizes do tipo *IMPES*. Os resultados apresentam uma queda de performance da versão *Sucuri* em todos os cenários. Fato que ocorre com a versão *OpenMP* apenas no cenário com 16 *threads*.

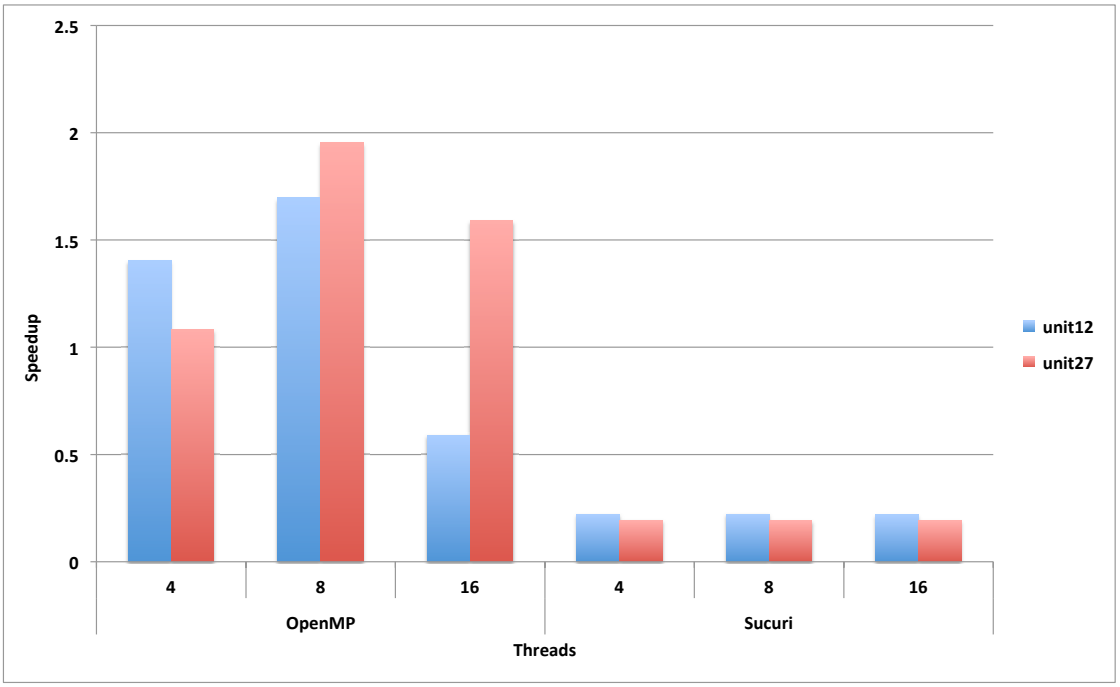


Figura 5.1: Comparação de performance (*Speedup*) do núcleo STS entre implementação *OpenMP* e *Sucuri* para matrizes tipo *IMPES* no Ambiente 1.

Para as matrizes do tipo *FIM*, como mostra a Figura 5.2, a versão *Sucuri* apresenta um ganho de performance acima da versão *OpenMP* nos cenários com 4 e 8 *threads*. Porém, devido a baixa taxa de operações ponto flutuante por entrada da matriz, a versão *Sucuri* não escalou com o aumento do número de *threads*.

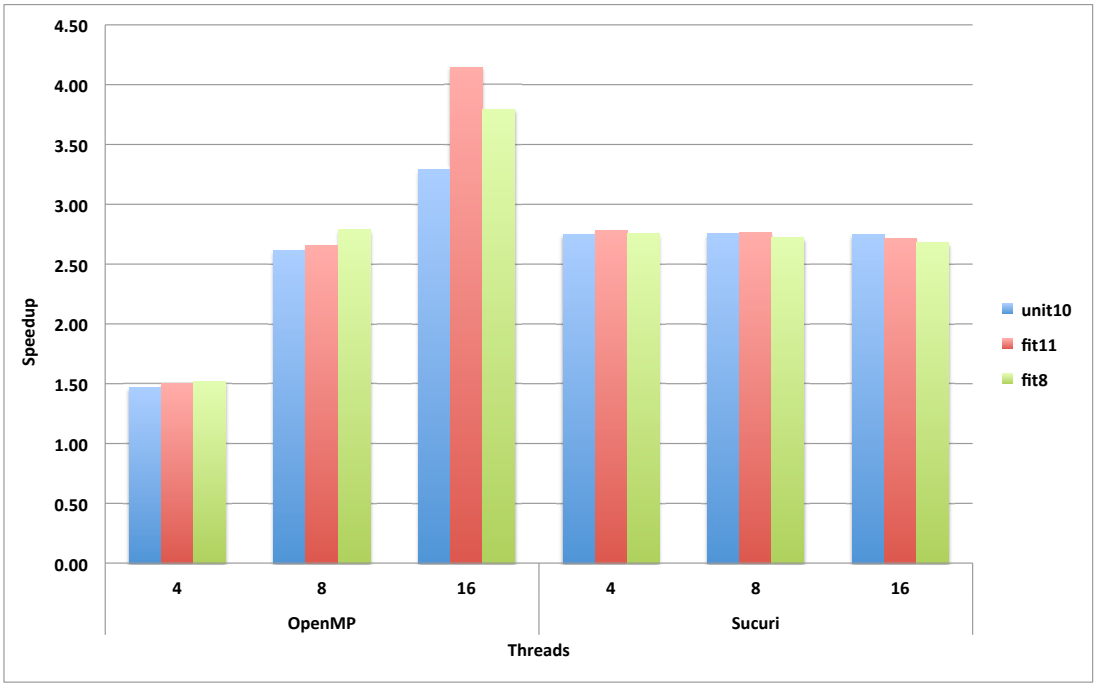


Figura 5.2: Comparação de performance (*Speedup*) do núcleo STS entre implementação *OpenMP* e *Sucuri* para matrizes tipo *FIM* no Ambiente 1

5.3 Avaliação do Núcleo SpMV Paralelo

O núcleo SpMV foi avaliado na máquina virtual *Trebuchet* e comparado com a implementação fornecida pela biblioteca *MKL* da Intel®. Um conjunto de matrizes, apresentadas na Subseção 5.1.2.1, foi utilizado e o número de *threads* fixado em 16. O *speedup*, para os dois cenários, foi caculado com base nos valores de tempo da implementação *MKL* com apenas 1 *thread*.

Como pode ser visto na Figura 5.3, a biblioteca Intel® *MKL* obteve melhor performance apenas no caso da matriz *water_tank*. Tal fato ocorre pois esta matriz possui um baixo número de linhas, desfavorecendo a máquina virtual *Trebuchet* que possui um *overhead* de inicialização. Já para os outros casos, a performance da *Trebuchet* superou a versão *MKL* pois custo de inicialização da máquina virtual foi amortizado.

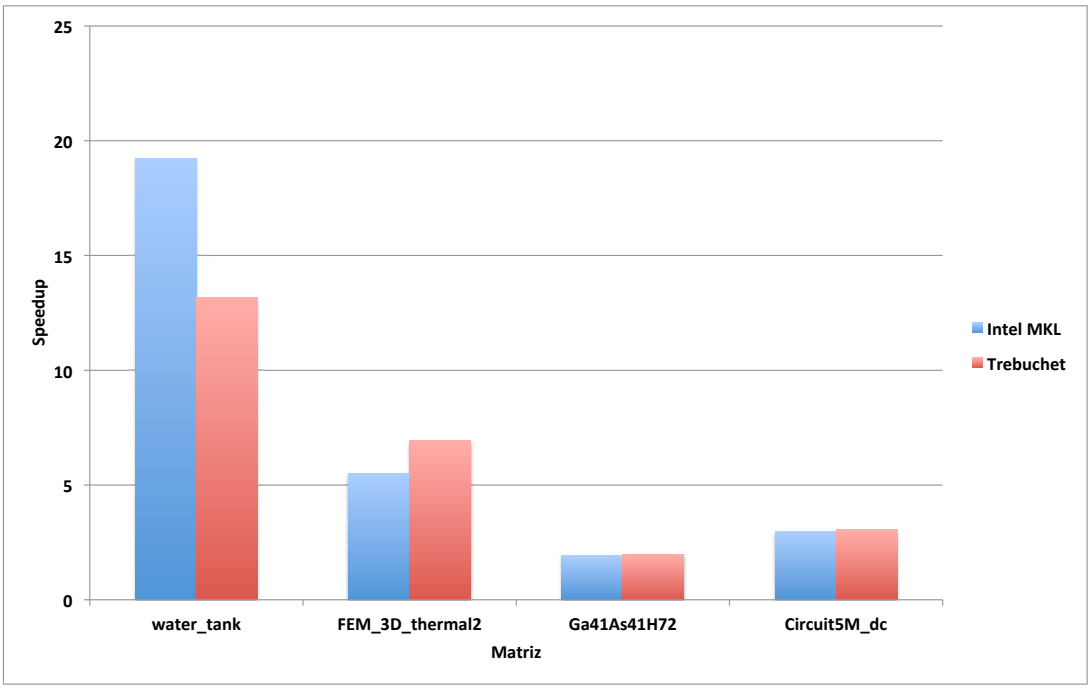


Figura 5.3: Comparação de performance (*Speedup*) do núcleo SpMV entre a biblioteca *MKL* da Intel® e a máquina virtual *Trebuchet* no Ambiente 1.

5.4 Avaliação do Núcleo GEMV Paralelo

A análise de performance do núcleo GEMV foi realizada utilizando a biblioteca *MKL* da *Intel*® e biblioteca *dataflow Sucuri* no Ambiente 1. Um conjunto de matrizes pseudo-aleatórias com precisão dupla foram geradas, em tempo de execução, com os seguintes tamanhos: 5.000, 10.000, 40.000 e 46.000[2]. É importante ressaltar que durante todas as execuções a mesma semente de geração pseudo-aleatória foi utilizada, criando assim matrizes com os mesmos valores em cada teste.

As Figuras 5.4 e 5.5 mostram a performance da aplicação na biblioteca *MKL* e *Sucuri* respectivamente. É possível notar que o problema possui uma escalabilidade baixa no caso onde a biblioteca *MKL* é utilizada. Tal fato ocorre pois a implementação em linguagem *C* é extremamente rápida, mesmo na versão sequencial. Com isso, matrizes pequenas tendem a não ter uma boa escalabilidade. Por outro lado, a implementação na *Sucuri* obteve bons resultados devido ao fato de que a versão sequencial em *Python* apresenta uma performance muito inferior quando

[2]O máximo suportado pelo ambiente de experimentos foi a matriz com tamanho de 46.000 por 46.000.

comparada com a versão em *C*.

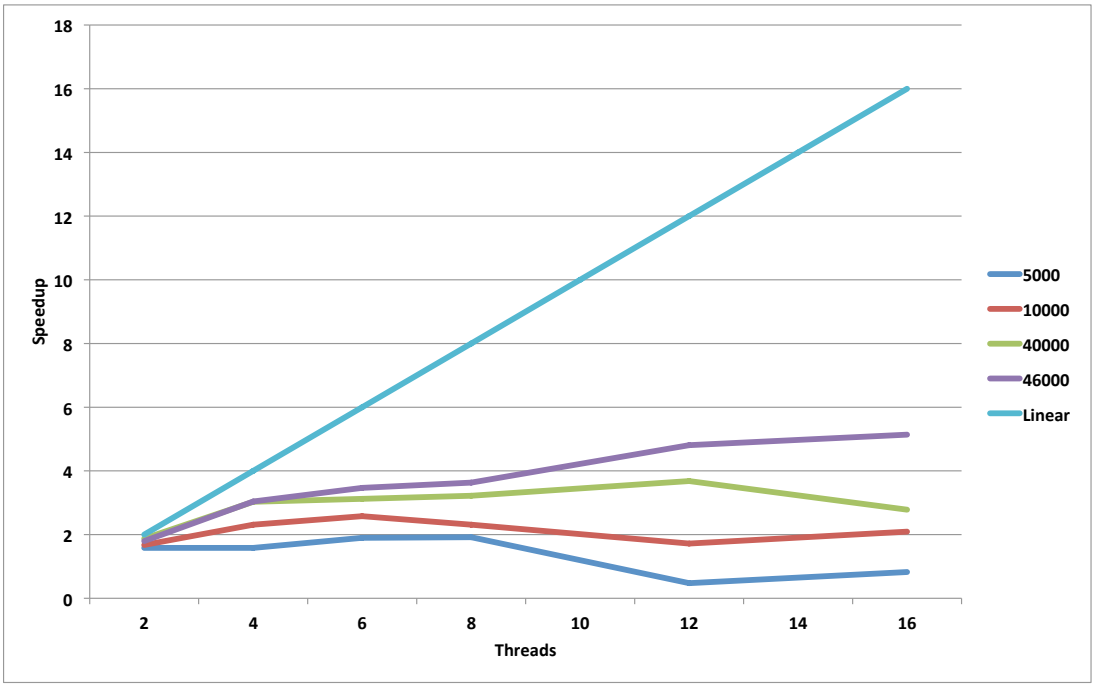


Figura 5.4: Performance (*Speedup*) do núcleo GEMV utilizando a biblioteca *MKL* da Intel no Ambiente 1.

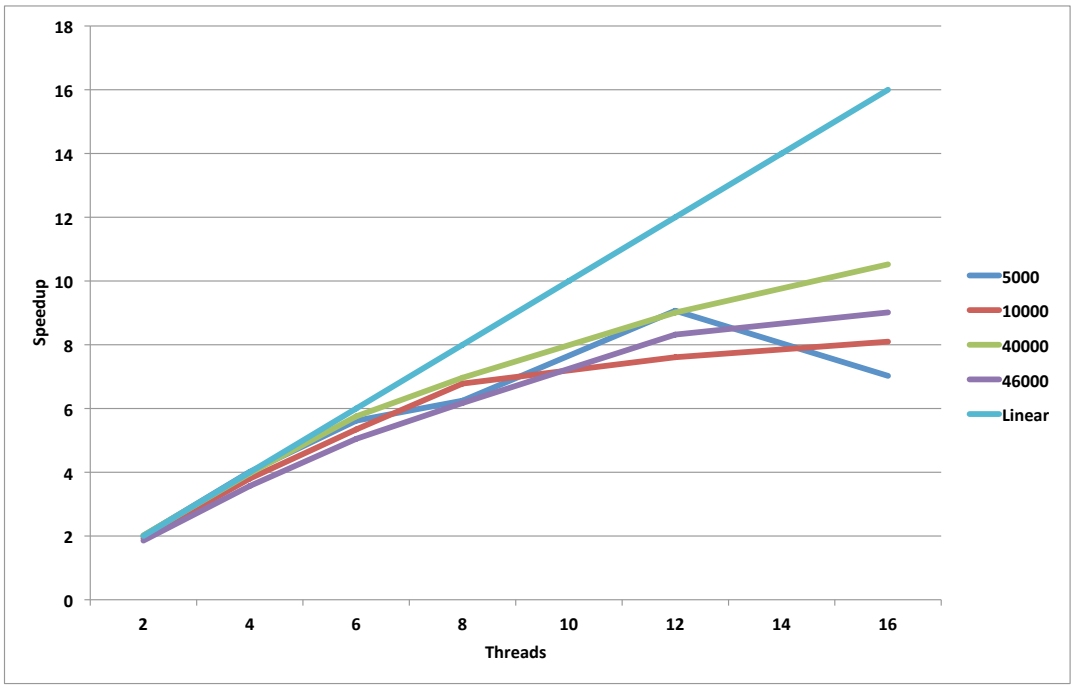


Figura 5.5: Performance (*Speedup*) do núcleo GEMV utilizando a biblioteca *Sucuri* no Ambiente 1.

5.5 Avaliação do Núcleo Determinante Paralelo

O núcleo Determinante foi avaliada através de implementações utilizando as bibliotecas *OpenMP* e *Sucuri* no Ambiente 1. Um conjunto de matrizes pseudo-aleatórias com precisão dupla e de tamanhos 1.000, 5.000 e 10.000 foram geradas conforme feito na Seção 5.4. No caso da aplicação *Sucuri*, apenas a entrada com tamanho 1.000 foi avaliada pois as demais não executaram em tempo viável.

As Figuras 5.6 e 5.7 mostram os resultados para as implementações em *OpenMP* e *Sucuri*. É possível notar que o *speedup* máximo da aplicação em *OpenMP* é de 4.2, mesmo variando até 16 *threads*. No caso da implementação *Sucuri*, a aplicação obteve uma queda de performance quando comparada a versão sequencial. Tal fato pode ser explicado pela estrutura do grafo utilizado para paralelizar o núcleo Determinante em *dataflow*. Um conjunto de subgrafos fork-join interligados, onde cada *thread* criada pelo nó fork processa um pequeno conjunto de dados, adiciona um custo muito alto de comunicação comparado ao custo de processamento.

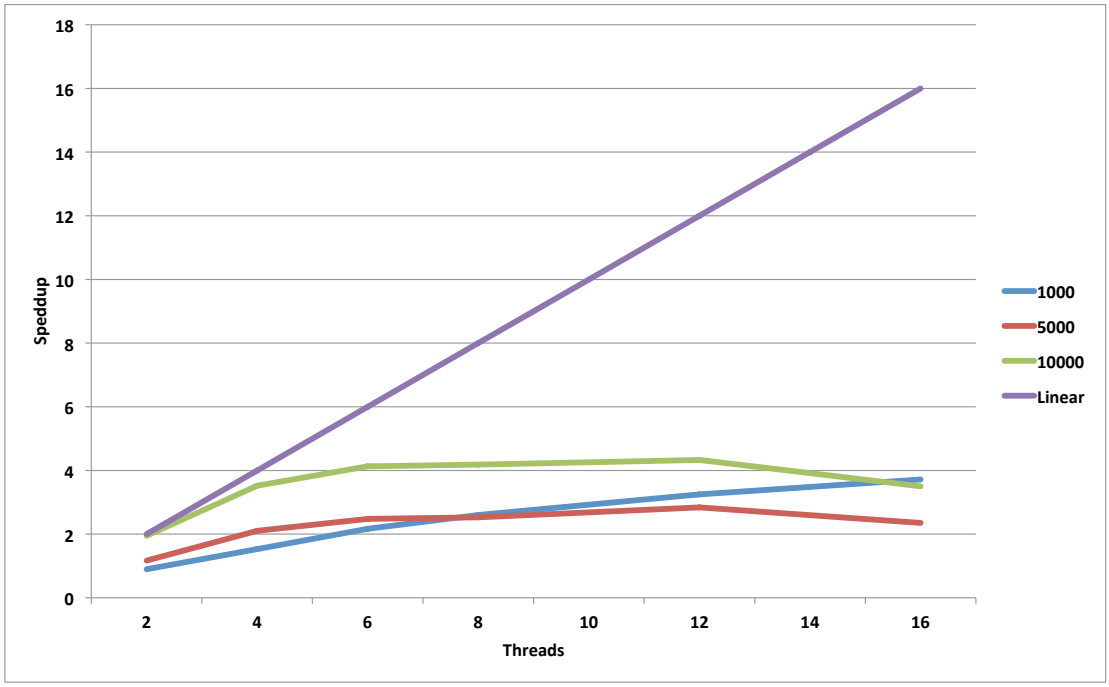


Figura 5.6: Performance (*Speedup*) do núcleo Determinante implementado em *OpenMP* no Ambiente 1.

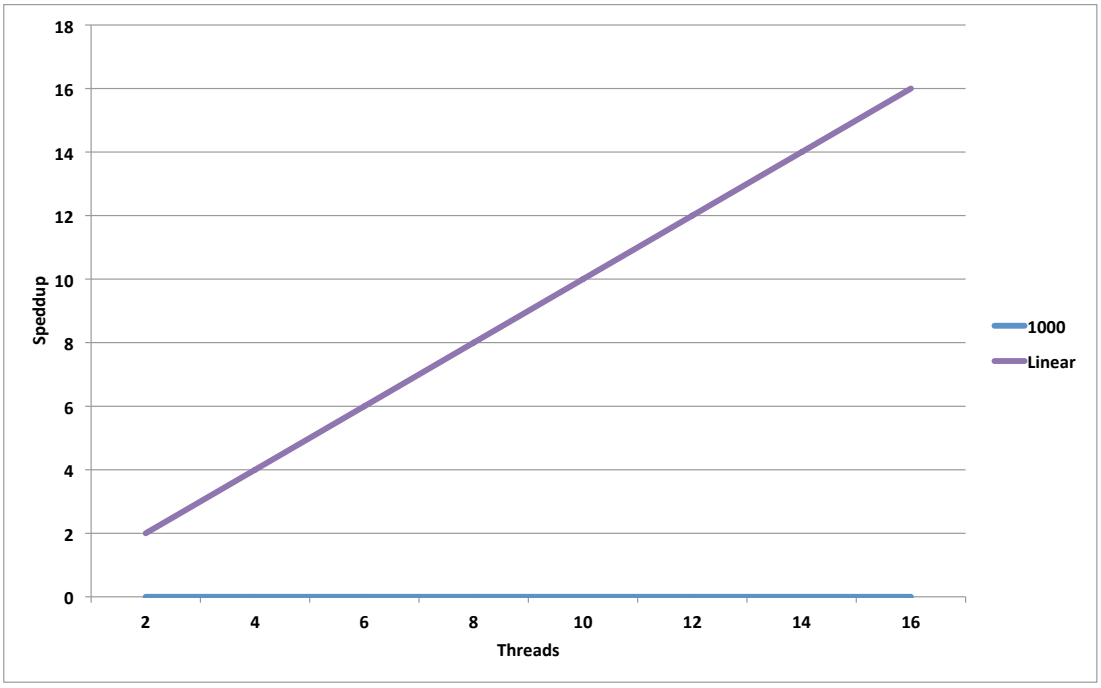


Figura 5.7: Performance (*Speedup*) do núcleo Determinante utilizando a biblioteca *Sucuri* no Ambiente 1.

5.6 Avaliação do Núcleo GEMM Paralelo

O desempenho do núcleo GEMM, implementando o algoritmo *Processor Farm*, foi avaliado utilizando-se matrizes pseudo-aleatórias de precisão dupla, com diferentes tamanhos e nos ambientes de execução 1 e 2. Os diferentes cenários de avaliação são descritos abaixo:

- Cenário 1:
  - Ordem das matrizes: 5.000, 10.000 e 15.000;
  - Número de threads: 2, 4, 6, 8, 12 e 16.
- Cenário 2:
  - Ordem da matriz: 10.000;
  - Número de nós: 2, 4, 6, 8, 12, 16;
  - Número de Processos *Sucuri* (por nó): 1;

  - Número de Tarefas[3] *Sucuri*: 2, 4, 6, 8, 12, 16.
  - Número de *Threads* (por processo *Sucuri*): 4.

- Cenário 3:
  - Ordem da matriz: 10.000;
  - Número de nós: 2, 4, 6, 8, 12, 16, 32;
  - Número de Processos *Sucuri* (por nó): 1;
  - Número de Tarefas[4] *Sucuri*: 2, 4, 6, 8, 12, 16, 32.
  - Número de *Threads* (por processo *Sucuri*): 4.

Nos cenários 2 e 3, um segundo nível de paralelização foi utilizado. *Threads* criadas por chamadas à biblioteca *Intel MKL* foram utilizadas por cada processo Sucuri executado dentro de cada nó do cluster. O número de *Threads* criadas foi baseado na quantidade de cores disponível em cada nó do cluster.

As Figuras 5.9 e 5.8 apresentam os resultados da *Sucuri* e do *Intel® MKL*, respectivamente, para o núcleo GEMM no Ambiente 1. A versão *dataflow* apresenta resultado inferior a versão da biblioteca *MKL* porém satisfatório, tendo em vista que a biblioteca *Sucuri* possui uma carga de comunicação entre os nós *dataflow*. Pode-se ressaltar também o fato de que a implementação da biblioteca *MKL* possui uma otimização mais *baixo nível*, incluindo loops desenrolados e a utilização de funções do tipo *intrinsics*.

[3]O número de tarefas é calculado a partir do número de nós vezes o número de processos utilizados em um determinado experimento.

[4]O número de tarefas é calculado a partir do número de nós vezes o número de processos utilizados em um determinado experimento.

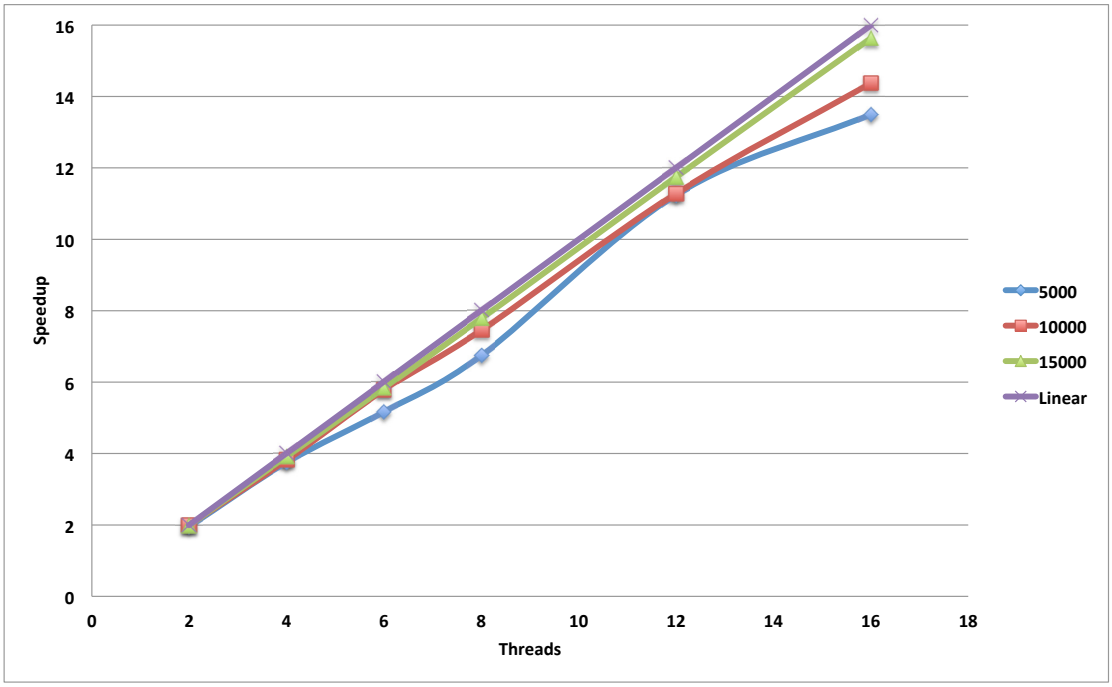


Figura 5.8: Performance (*Speedup*) do núcleo GEMM utilizando a biblioteca *MKL* da Intel variando o número de *threads* no Ambiente 1.

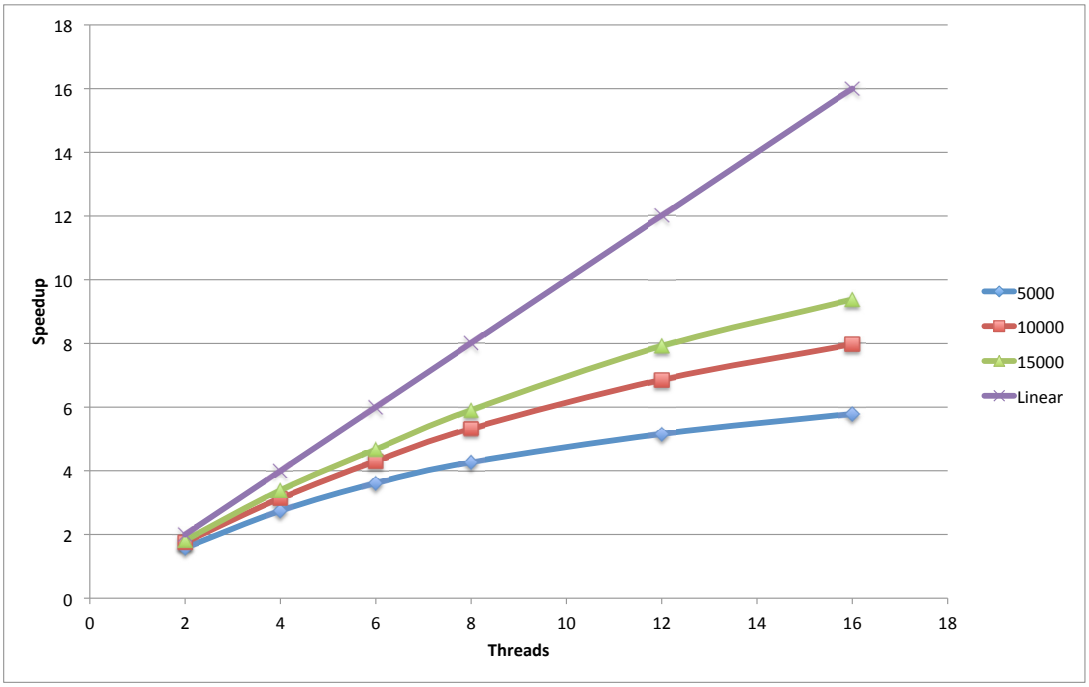


Figura 5.9: Performance (*Speedup*) do núcleo GEMM na *Sucuri* variando o número de *threads* no Ambiente 1.

A avaliação do núcleo GEMM no Ambiente 2, ou seja, em um sistema de memória distribuída, é composta por várias etapas. Primeiramente são realizados testes com o código original para o Cenário 2, onde o tamanho da matriz é fixado e o número

de nós/cores é variado.

Na segunda etapa, o nó *join*, responsável pelo agrupamento dos blocos de linhas que compõem a matriz C original, é removido com o intuito de avaliar a carga de envio dos resultados processados por cada nó para o nó principal.

Com base nos resultados apresentados nos testes sem o nó *join*, uma terceira e última etapa foi realizada utilizando o Cenário 3. Nesta, cada nó diminui gradativamente a porção de linhas da matriz resultante a ser enviada. Variando-se o número de nós envolvidos na computação, o custo de envio da matriz resultante é detalhado para cada caso.

As Figuras 5.10 e 5.11 exibem os resultados para as etapas 1 e 2 respectivamente. É possível notar que o *speedup* sai de um fator de aproximadamente 2.6 para super linear, chegando a um fator de 78 no melhor caso, na Figura 5.11. Porém, tal gráfico não deixa claro o percentual de tempo envolvido com a computação e com a transferência dos resultados, já que não há envio dos resultados para o nó principal.

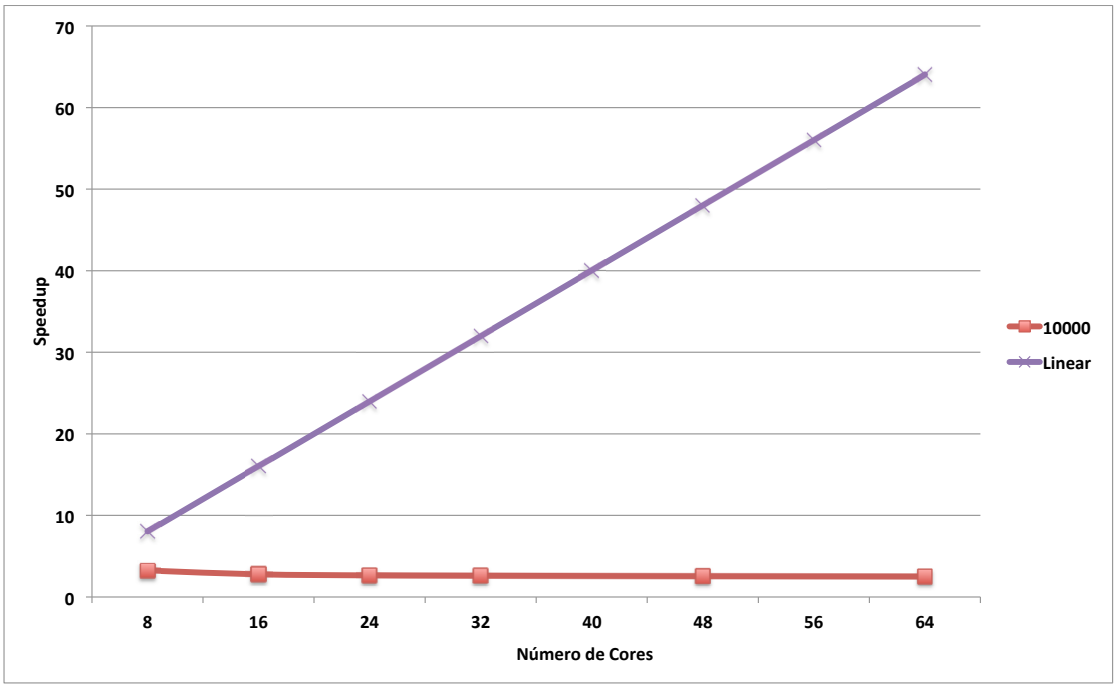


Figura 5.10: Performance (*Speedup*) do núcleo GEMM na *Sucuri* variando o número de *cores* no Ambiente 2.

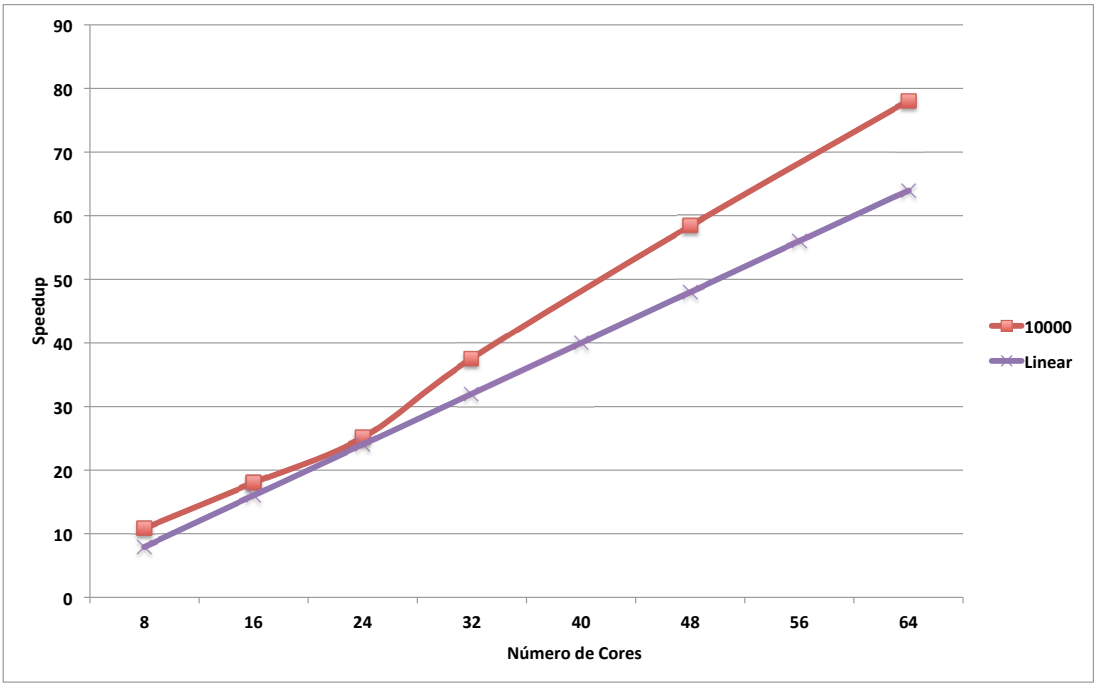


Figura 5.11: Performance (*Speedup*) do núcleo GEMM na *Sucuri* variando o número de *threads* no Ambiente 2.

Os resultados do testes da etapa 3 são exibidos nas Figuras 5.12 e 5.13. Na primeira figura é possível observar o comparativo entre o tempo total da aplicação e o percentual das linhas da matriz resultante transferidas para o nó central. Cada linha representa um cenário com uma certa quantidade de nós do cluster envolvidos na computação. A segunda figura possui os mesmos dados da primeira figura. Porém, o comparativo agora é feito entre o *speedup* da aplicação e o número de cores envolvidos na computação. Neste caso, as linhas representam o percentual das linhas da matriz resultante transferidas.

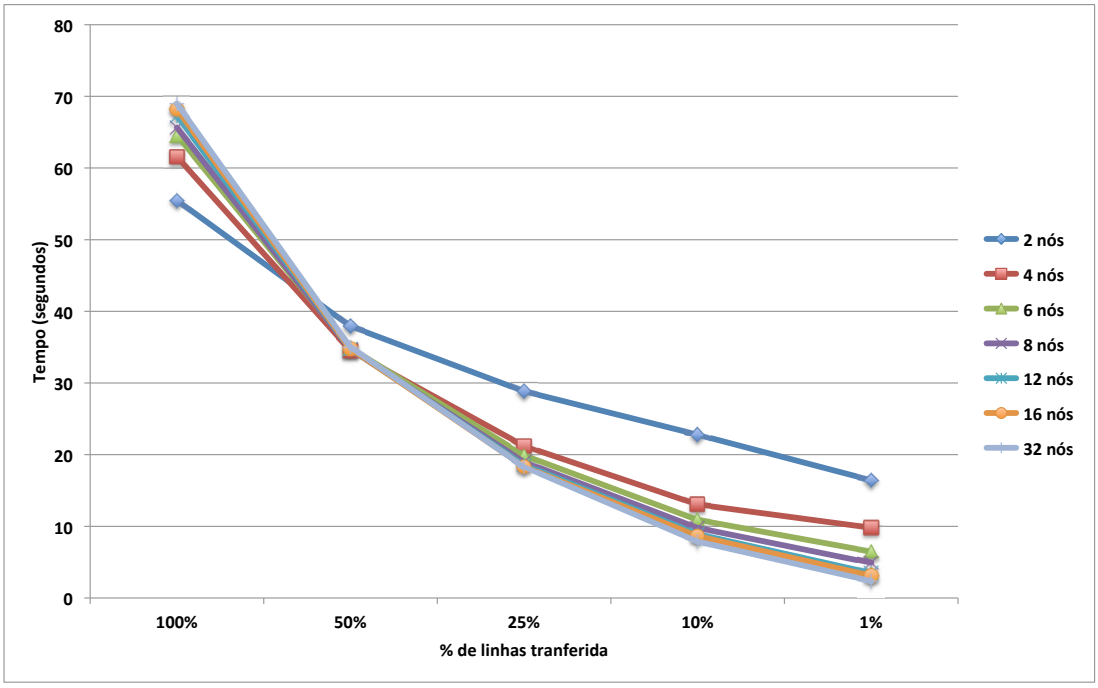


Figura 5.12: Tempo gasto do núcleo GEMM na *Sucuri* variando o percentual de linhas da matriz C transferida para o nó central utilizando o Ambiente 2 e o Cenário 3.

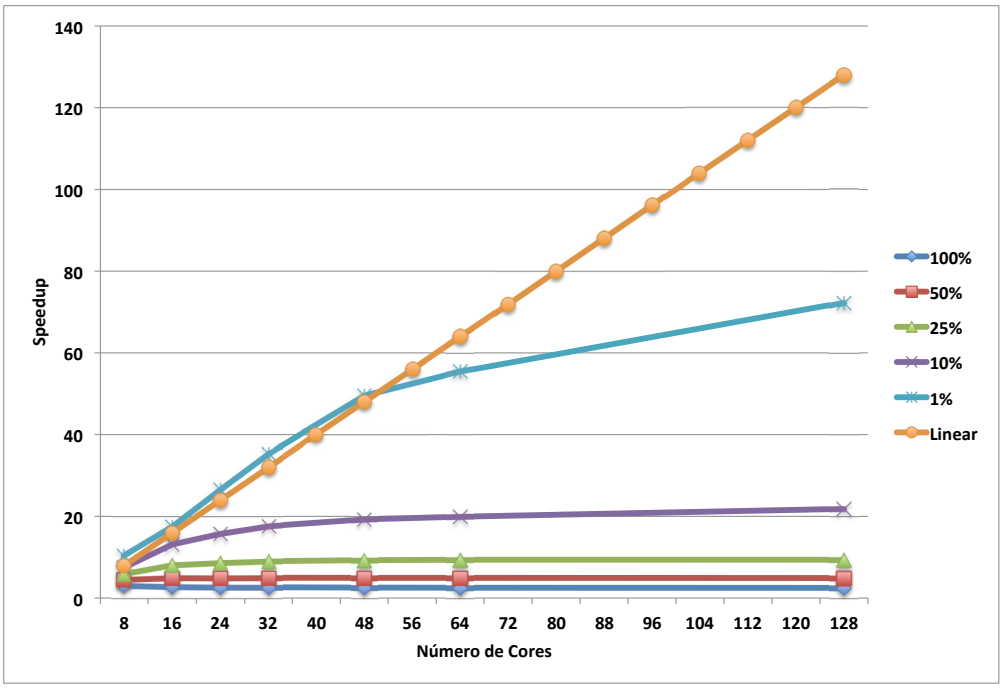


Figura 5.13: Performance (*Speedup*) do núcleo GEMM na *Sucuri* variando o número de *cores* no Ambiente 2 e Cenário 3.

Através dos experimentos realizados nesta etapa, é possível distinguir o impacto da transferência necessária para executar o nó *join*. Com a taxa fixada em 1% das

linhas, a performance do núcleo GEMM chega a ser 72 vezes mais rápida do que a versão sequencial para o caso onde foram utilizados 128 cores. É importante ressaltar, novamente, que as máquinas possuem finalidade didática, ou seja, o cluster não foi construído com o intuito de alcançar uma computação de alto desempenho. Tal fato é comprovado através dos resultados apresentados na Figura 5.14. Neste experimento, a implementação *Sucuri* é comparada com uma implementação, em Python, que utilizando o padrão *Message Passing Interface - MPI*. É possível notar que, mesmo utilizando o padrão estado da arte *MPI*, a implementação do núcleo *GEMM* não escalou como esperado. Portanto, pode-se concluir que a rede de interconexão utilizada, bem como sua topologia, impossibilitaram o real desempenho da aplicação.

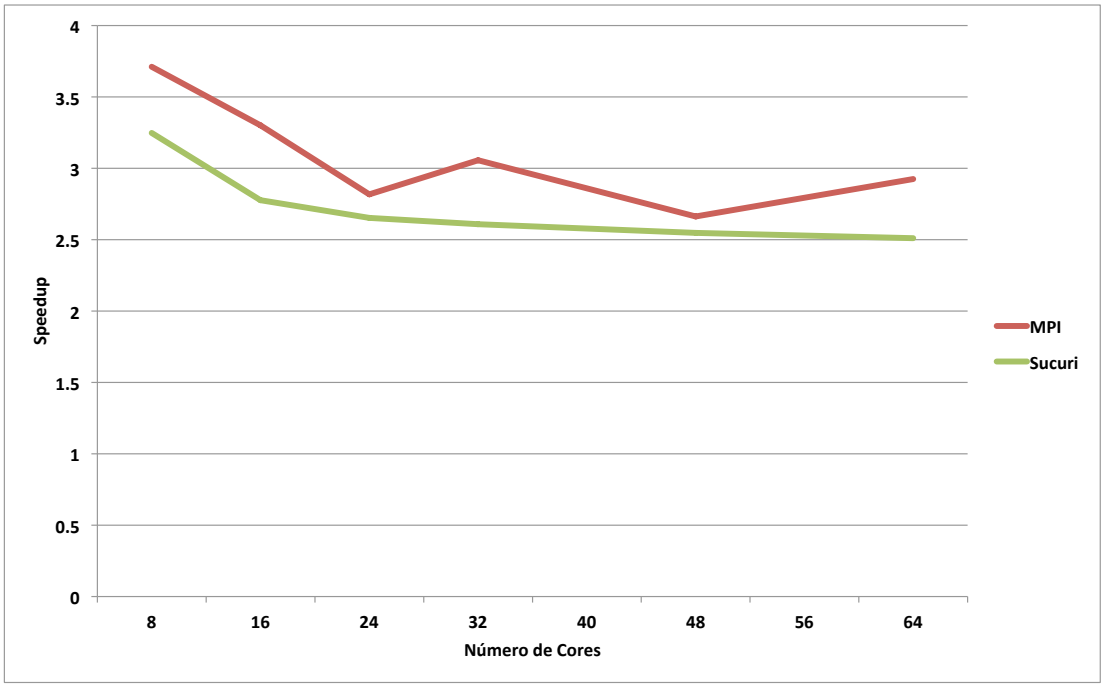


Figura 5.14: Comparação de performance (*Speedup*) do núcleo GEMM implementado em Python *MPI* e Sucuri variando o número de *cores* no Ambiente 2.

Capítulo 6

Conclusões e Trabalhos Futuros

Neste trabalho foram apresentadas implementações paralelas de núcleos de álgebra linear computacional utilizando o modelo guiado por fluxo de dados (*dataflow*). Para desenvolver tais aplicações, dois ambientes, que seguem o modelo de execução *dataflow* híbrido, foram utilizados. O primeiro, intitulado de *Trebuchet*, é uma máquina virtual *dataflow* que permite a paralelização de códigos sequencias na linguagem *C* através de anotações no código fonte da aplicação. O segundo ambiente *dataflow* foi a *Sucuri*. Uma biblioteca desenvolvida em linguagem *Python* que permite a criação e execução de grafos *dataflow*. Nós com funções específicas são criados e interconectados através de arestas que definem as dependências de dados. Os núcleos avaliados neste trabalho foram o solver triangular esparso, as multiplicações matriz vetor densa e esparsa, o cálculo do determinante e a multiplicação de matrizes densas.

Os experimentos avaliaram a performance de cada implementação, bem como a escalabilidade com o aumento do número de cores. Em núcleos onde a computação é mais intensa, a implementação *dataflow* obteve bons resultados, como os caso dos núcleos GEMM, GEMV e SpMV. Apesar da boa performance do núcleo STS utilizando as matrizes do tipo FIM, a aplicação não escalou como esperado. Os núcleos STS, com matrizes IMPES, e Determinante, tiveram uma queda de performance devido a diferença entre o custo de processamento e o custo de comunicação.

O núcleo GEMM, quando avaliado para o ambiente cluster, apresenta uma perda de performance devido ao envio dos dados após o processamento em cada nó. Para essa implementação, é necessário uma abordagem mais aprimorada como a redução

hierárquica. Tal abordagem pode reduzir o gargalo criado ao centralizar todo processo agrupamento no nó *join* e aumentar a performance do núcleo, como demonstrado nas avaliações com redução do tráfego de dados.

Para facilitar o desenvolvimento de aplicações para a *Sucuri*, que utilizem os núcleos apresentados neste trabalho, nós especializados serão criados. Tais nós podem ser instanciados durante a criação do grafo, sendo necessário apenas indicar as dependências.

Com o surgimento de novas arquiteturas paralelas, é esperado que tais ferramentas possibilitem a execução de algoritmos em ambientes heterogêneos com um nível alto de abstração. A integração da biblioteca *Sucuri* com tais arquiteturas é essencial. Sendo assim, um nó especializado, voltado para execução em coprocessadores *Intel Xeon Phi*® [14], será desenvolvido. Bibliotecas que possibilitam tal integração, como a *pyMIC*[15], serão utilizadas afim de tornar a *Sucuri* mais robusta. Com isso, o desenvolvedor terá a possibilidade de disparar nós em diferentes tipos de arquiteturas, tirando proveito do que cada uma tem de melhor a oferecer.

Referências Bibliográficas

[1] DAVIS, T. A., HU, Y., "The University of Florida Sparse Matrix Collection", *ACM Trans. Math. Softw.*, v. 38, n. 1, pp. 1:1–1:25, Dec. 2011.

[2] DENNIS, J. B., MISUNAS, D. P., "A Preliminary Architecture for a Basic Data-flow Processor". In: *Proceedings of the 2Nd Annual Symposium on Computer Architecture*, *ISCA '75*, pp. 126–132, ACM: New York, NY, USA, 1975.

[3] ALVES, T., GOLDSTEIN, B., FRANCA, F., etãl., "A Minimalistic Dataflow Programming Library for Python". In: *Computer Architecture and High Performance Computing Workshop (SBAC-PADW), 2014 International Symposium on*, pp. 96–101, Oct 2014.

[4] MARZULO, L. A. J., *Explorando Linhas de Execução Paralelas com Programação Orientada por Fluxo de Dados*, Ph.D. Thesis, The school where the thesis was written, 2011.

[5] SAAD, Y., *Iterative Methods for Sparse Linear Systems*. 2nd ed. Society for Industrial and Applied Mathematics: Philadelphia, PA, USA, 2003.

[6] SAAD, Y., "SPARSKIT: a basic tool kit for sparse matrix computations - Version 2", 1994.

[7] J., DONGARRA, "Sparse matrix storage formats", In: BAI, Z., DEMMEL, J., DONGARRA, J., etãl. (eds), *Templates for the Solution of Algebraic Eigenvalue Problems*, SIAM, 2000.

[8] NAUMOV, M., *Parallel Solution of Sparse Triangular Linear Systems in the Preconditioned Iterative Methods on the GPU*, Tech. Rep. 001, NVIDIA Corporation, 2011.

[9] PARK, J., SMELYANSKIY, M., SUNDARAM, N., etãl., "Sparsifying Synchronization for High-Performance Shared-Memory Sparse Triangular Solver", In: *Supercomputing*, v. 8488, pp. 124–140, Springer International Publishing, 2014.

[10] NATHAN BELL, M. G., *Efficient sparse matrix-vector multiplication on CUDA*, Tech. Rep. NVR-2008-004, NVIDIA Corporation, 2008.

[11] DEMMEL, J., "Single Processor Machines: Memory Hierarchies and Processor Features", `http://www.cs.berkeley.edu/~demmel/cs267_Spr14/Lectures/lecture02_memhier_jwd14_4pp.pdf`, 2014, Accessed: 14-08-2015.

[12] DAGUM, L., MENON, R., "OpenMP: an industry standard API for shared-memory programming", *Computational Science Engineering, IEEE*, v. 5, n. 1, pp. 46–55, Jan 1998.

[13] "Intel® Math Kernel Library (Intel® MKL)", `https://software.intel.com/en-us/intel-mkl`, Accessed: 18-08-2015.

[14] "Intel® Xeon Phi", `http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html`, Accessed: 30-08-2015.

[15] KLEMM, M., ENKOVAARA, J., "pyMIC: A Python Offload Module for the IntelR Xeon Phi® Coprocessor", *8th European Conference on Python in Science*, 2015.

[16] GOLDSTEIN, B., ALVES, T., SOUZA, M., etãl., "Implementing a Parallel Sparse Matrix-Vector Multiplication Using Dataflow", *Anais XXXV Congresso Nacional de Matemática Aplicada e Computacional*, 2014.

[17] ALVES, T. A. O., MARZULO, L. A. J., FRANCA, F. M. G., etãl., "Trebuchet: Exploring TLP with Dataflow Virtualisation", *Int. J. High Perform. Syst. Archit.*, v. 3, n. 2/3, pp. 137–148, May 2011.

[18] SOLINAS, M., BADIA, R., BODIN, F., etãl., "The TERAFLUX Project: Exploiting the DataFlow Paradigm in Next Generation Teradevices". In: *Digital System Design (DSD), 2013 Euromicro Conference on*, pp. 272–279, Sept 2013.

[19] BABOULIN, M., BUTTARI, A., DONGARRA, J., etãl., "Accelerating scientific computations with mixed precision algorithms", *Computer Physics Communications*, v. 180, n. 12, pp. 2526 – 2533, 2009.

[20] DE OLIVEIRA ALVES, T. A., *Execução Especulativa em uma Máquina Virtual Dataflow*, Master's Thesis, PESC/COPPE/UFRJ, 2010.

[21] FORTES, W. R., *Precondicionadores e solucionadores para resolução de sistemas lineares obtidos de simulação de enchimento de reservatório*, Master's Thesis, PPG-EM UERJ, 2008.

[22] NIEVINSKI, I. C., *JinSol, Interface em Java Para Sistemas Lineares*, Master's Thesis, PPG-EM UERJ, 2013.

Apêndice A

Iniciativas - Projeto CENPES

Este trabalho foi desenvolvido em parceria com o grupo de pesquisa em reservatórios petrolíferos do Centro de Pesquisas e Desenvolvimento Leopoldo Américo Miguez de Mello (CENPES) da Petrobras. Durante minha participação, desenvolvi diversos trabalhos além dos descritos nos capítulos anteriores. Dentre eles destacam-se a análise das matrizes esparsas, o impacto das estruturas de compactação e experimentos com precisão mista.

A.1 Análise das Matrizes Esparsas

As matrizes esparsas utilizadas durante os experimentos são compostas por um conjunto de sistemas lineares específicos de uma iteração do simulador de reservatório. Tal iteração é tida como o *pior caso*, ou seja, a iteração cuja solução do sistema levou mais tempo para ser processada. Uma análise da esparsidade de tais matrizes, bem como a distribuição de não zeros, é de suma importância pois tal fato pode afetar diretamente na performance dos núcleos. Um algoritmo de análise de distribuição foi implementado afim de compreender a estrutura das matrizes de *pior caso*. As informações sobre a distribuição de não-zeros ao longo das linhas e colunas da matriz e a densidade de não-zeros nos blocos, para matrizes do tipo *FIM*, foram extraídas. Os resultados mostraram que, para essas iterações específicas, o percentual de linhas e colunas com apenas 1 elemento não-zero chega a 46% e blocos com densidade de 1 não-zero chega a 79% nos piores casos. As

tabelas com os valores de distribuição de densidade para cada matriz estão listadas no Apêndice B.

A.2 Impacto de Compactação das Matrizes Esparsas

Devido ao tamanho e a grande percentagem de zeros nas matrizes, uma estrutura de dados compacta é necessária para que as mesmas sejam carregadas em memória. Um exemplo de estrutura de compactação popularmente utilizado é o formato CSR. Em tal formato, os elementos não nulos são compactados seguindo a ordem das linhas da matriz original, utilizando dois vetores de ponteiros e um de dados. Com isso, o acesso aos valores não nulos da matriz é feito de forma indireta, ou seja, é necessário dar *"saltos"* de acordo com os valores indicados nos vetores de ponteiros. Este fato compromete otimizações que poderiam ser feitas em tempo de compilação, como o *desenrolar dos loops* e em tempo de execução, como o *prefecth* de dados acessados de forma regular dentro dos *loops*. Para avaliar este impacto, alguns experimentos com o núcleo de álgebra linear $AXPY$ foram realizados. Pode-se perceber que tal estrutura possui impacto na performance da aplicação porém, os resultados ainda não são conclusivos e necessitam um conjunto de testes mais abrangente.

A.3 Técnica de Precisão Mista

Em *M. Baboulin et al.*[19] a técnica de precisão mista, onde operações ponto flutuante com 32 bits e 64 bits são combinadas, é apresentada como uma possível estratégia para ganho de performance em operações de álgebra linear densas e esparsas. Tal técnica foi avaliada através da implementação de um solver triangular esparso, onde os dados de entrada possuem precisão simples e dupla. Um conjunto de matrizes esparsas originadas da discretização em diferenças finitas do operador laplaciano em três dimensões, com diferentes tamanhos, foram criadas. Os resultados obtidos no testes com o solver triangular esparso não refletiram o ganho de perfor-

mance obtido por *M. Baboulin et al.*[19]. Estimamos que a estrutura de compactação possa ter interferido no possível ganho da técnica de precisão mista. Porém, é necessário uma análise mais complexa utilizando ferramentas de perfilamento, como o *Intel® Vtune*, para que tal comportamento seja compreendido.

Apêndice B

Códigos Fonte

Neste Apêndice, algumas implementações utilizadas para avaliar os núcleos são apresentadas.

```
1  void lsolve(csr *L, dvec *x, dvec *y){
2    int i, j;
3
4    for (i = 0; i< L->n; i++){
5      y->x[i] = x->x[i];
6      for (j = L->i[i]; j < L->i[i+1]; j++){
7        if(i == L->j[j]){
8          y->x[i] /= L->x[j];
9          break;
10       }else{
11         y->x[i] -= L->x[j] * y->x[L->j[j]];
12       }
13     }
14   }
15 }
```

Algoritmo B.1: Código em C para substituição direta sem bloco.

```c
void usolve(csr *U, dvec *y, dvec *x){
  int i, j;

  for (i = U->n - 1; i >= 0; --i){
    x->x[i] = y->x[i];
    for (j = U->i[i+1]; j > U->i[i]; --j){
      if(i == U->j[j - 1]){
        x->x[i] /= U->x[j - 1];
        break;
      }else{
        x->x[i] -= U->x[j - 1] * x->x[U->j[j - 1]];
      }
    }
  }
}
```

Algoritmo B.2: Código em C para Substituição Reversa sem bloco.

```c
void blsolveInverseBlock(bcsr *L, dvec *x, dvec *y){
  int i, j, k, l, m;
  int nz = L->bs * L->bs;

  for(i = 0; i< L->n * L->bs; ++i) y->x[i] = 0;

  for (i = 0; i< L->n; ++i){
    for(k = 0; k < L->bs; k++){
      y->x[i * L->bs + k] = x->x[i * L->bs + k];
    }
    for(j = L->i[i]; j < L->i[i + 1] - 1; j++){
      for(k = 0; k < L->bs; k++){
        for(l = 0; l < L->bs; l++){
          y->x[i * L->bs + l] -= L->x[j * nz + k * L->bs + l] *
              y->x[L->j[j] * L->bs + k];
        }
      }
    }
  }
}
```

Algoritmo B.3: Código em C para Substituição Direta em bloco.

```c
void busolveInverseBlock(bcsr *U, dvec *y, dvec *x){
  int i, j, k, l, m;
  int nz = U->bs * U->bs;

  double *xt, t;

  xt = (double*) malloc(U->bs * sizeof(double));

  for (i = U->n -1 ; i >= 0; --i){
    for(k = 0; k < U->bs; k++){
      xt[k] = y->x[i * U->bs + k];
    }
    for (j = U->i[i + 1] - 1; j > U->i[i]; --j){
      for(k = 0; k < U->bs; k++){
        t = xt[k];
        for(l = 0; l < U->bs; l++){
          t -= U->x[j * nz + k * U->bs + l] * x->x[U->j[j] *
              U->bs + l];
        }
        xt[k] = t;
      }
    }
    for(k = 0; k < U->bs; k++){
      for(l = 0; l < U->bs; l++){
        x->x[i * U->bs + k] += U->x[j * nz + k * U->bs + l] *
            xt[l];
      }
    }
  }
}
```

Algoritmo B.4: Código em C para Substituição Reversa em bloco.

```
def lsolveGlobal(self, args=[]):
    global lowerCSRFile
    global vectorY

    for j in xrange(int(lowerCSRFile.row[self.rowIndex]),
    int(lowerCSRFile.row[self.rowIndex+1])):
        if(self.rowIndex == int(lowerCSRFile.column[j])):
            vectorY.vector[self.rowIndex] =
                float(vectorY.vector[self.rowIndex]) /
                float(lowerCSRFile.values[j])
            break
        else:
            vectorY.vector[self.rowIndex] =
                float(vectorY.vector[self.rowIndex]) -
                (float(lowerCSRFile.values[j]) *
                float(vectorY.vector[int(lowerCSRFile.column[j])]))
    return 0
```

Algoritmo B.5: Código Sucuri para Substituição Direta sem bloco.

```
def usolveGlobal(self, args=[]):
    global upperCSRFile
    global vectorY

    for j in xrange(int(upperCSRFile.row[self.rowIndex+1]),
                int(upperCSRFile.row[self.rowIndex]), -1):
        if(self.rowIndex == int(upperCSRFile.column[j - 1])):
            vectorY.vector[self.rowIndex] =
                float(vectorY.vector[self.rowIndex]) /
                float(upperCSRFile.values[j - 1])
            break
        else:
            vectorY.vector[self.rowIndex] =
                float(vectorY.vector[self.rowIndex]) -
                (float(upperCSRFile.values[j - 1]) *
                float(vectorY.vector[int(upperCSRFile.column[j -
                1])]))
    return 0
```

Algoritmo B.6: Código Sucuri para Substituição Reversa sem bloco.

```
def dgemm_par(id, args):
    global matrixA
    global matrixB
    global matSize

    tid = get_tid(id)

    blockDim = matSize / ntasks
    start = tid * blockDim
    end = matSize if (tid + 1) == ntasks else start + blockDim

    cij = alpha * np.dot(matrixA[start:end,0:matSize],
        matrixB[0:matSize,0:matSize])
    return [start, end, cij]
```

Algoritmo B.7: Código Sucuri para Multiplicação de Matrizes utilizando o algoritmo *Processor Farm*.