Universidade Federal do Rio Grande do Norte
Centro de Tecnologia
Programa de Pós-Graduação em Engenharia Elétrica e de Computação

Implementação Paralela Escalável e Eficiente do Algoritmo Simplex Padrão em Arquitetura *Multicore*

Demétrios A. M. Coutinho

Orientador: Prof. Dr. Samuel Xavier de Souza

Coorientador: Prof. Dr. Jorge Dantas de Melo

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica da UFRN (área de concentração: Engenharia de Computação) como parte dos requisitos para obtenção do título de Mestre em Ciências.

Número de ordem PPgEE: M417

Natal, RN, Janeiro de 2014

UFRN / Biblioteca Central Zila Mamede.
Catalogação da Publicação na Fonte

Coutinho, Demétrios A. M.
Implementação paralela escalável e eficiente do algoritmo simplex padrão em arquitetura *multicore* / Demétrios A. M. Coutinho. – Natal, RN, 2014.
98 f. : il..

Orientador: Prof. Dr. Samuel Xavier de Souza.
Coorientador: Prof. Dr. Jorge Dantas de Melo.

Dissertação (Mestrado) – Universidade Federal do Rio Grande do Norte. Centro de Tecnologia. Programa de Pós-Graduação em Engenharia Elétrica e de Computação.

1. Simplex – Dissertação. 2. CPLEX – Dissertação. 3. Eficiência paralela – Dissertação. 4. Escalabilidade paralela – Dissertação. 5. Programação linear – Dissertação. I. Souza, Samuel Xavier de. II. Melo, Jorge Dantas de. III. Universidade Federal do Rio Grande do Norte. III. Título.

RN/UF/BCZM CDU 519.852

Demétrios A. M. Coutinho

Implementação Paralela Escalável e Eficiente do Algoritmo Simplex Padrão na Arquitetura *Multicore*

Defesa de mestrado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica e de Computação da UFRN (área de concentração: Engenharia de Computação) como parte dos requisitos para obtenção do título de Mestre em Ciências.

Trabalho aprovado. Natal, RN, 24 de Janeiro de 2014:

Prof. Dr. Samuel Xavier de Souza
Orientador

Prof. Dr. Jorge Dantas de Melo
coorientador

Prof. Dr. Daniel Aloise
Examinador interno

Prof. Dr. Danniel Cavalcante Lopes
Examinador Externo

Natal, RN
Jan/2014

A Deus e aos meus pais.

Agradecimentos

Agradeço a Deus, pelo seu Amor incondicional na minha história, por ter me concedido a oportunidade de conhecer pessoas que me ajudaram de forma direta ou indireta a concluir mais uma etapa na minha vida.

Aos meus pais Wilder César Coutinho e Josiane Magalhães, por sempre me apoiarem, por sempre estarem do meu lado, por sempre acreditarem em mim, e principalmente agradeço por todo esforço realizado para que tivesse a base moral e emocional adequada para vencer os desafios da vida.

Agradeço, especialmente, a Ayla Márcia, por todo seu carinho e zelo por mim, por ser tão paciente e compreensiva comigo, pelas correções ortográficas desse texto e o respaldo necessário para a conclusão desta jornada.

Ao meu orientador Samuel Souza, excelente profissional, pelo seu voto de confiança, por ter gasto várias horas auxiliando-me nesse trabalho, sem sua ajuda este trabalho não teria sido possível.

Ao meu coorientador Jorge Melo, por ter me acompanhado nessa jornada e pelos ensinamentos que me foram concedidos.

Ao Prof. Daniel Aloise por ter me ajudado com a versão acadêmica do CPLEX® e por suas orientações na área de otimização.

Ao Prof. Luiz Affonso, por ter me concedido um espaço no laboratório do LII.

À Viviane Medeiros, por sempre ter me ajudado na parte burocrática, nunca negando um pedido de socorro.

Ao meu colega de trabalho e de dissertação, Diego Cirilo, por ter esclarecido vários pontos da norma da ABNT, por ter me socorrido em vários problemas do Latex, por todas as madrugadas que passamos escrevendo os nossos textos.

À Direção do IFRN câmpus Pau dos Ferros, nas pessoas de Antônia Francimar da Silva e Amélia Cristina Reis e Silva, por todo apoio e auxílio concedido para concluir esse trabalho.

À CAPES e CNPq pelo apoio financeiro.

"Feliz aquele que se compraz no serviço do Senhor e medita sua lei dia e noite.
Ele é como a árvore plantada na margem das águas correntes:
dá fruto na época própria, sua folhagem não murchará jamais.
Tudo o que empreende, prospera.
(Bíblia Sagrada, Salmos 1, 2-3)

Resumo

Este trabalho apresenta uma implementação paralela escalável e eficiente do algoritmo Simplex padrão em arquitetura de processadores *multicore* para resolver problemas de programação linear de grande escala. Apresenta-se um esquema geral explicando como foi paralelizado cada passo do algoritmo simplex padrão, apontando pontos importantes da implementação paralela. Foram realizadas análises de desempenho através da comparação dos tempos sequenciais utilizando o Simplex tableau e Simplex do CPLEX® da IBM. Os experimentos foram realizados em uma máquina de memória compartilhada com 24 núcleos. A análise de escalabilidade foi feita com problemas de diferentes dimensões, encontrando evidências de que a implementação paralela proposta do algoritmo simplex padrão tem melhor eficiência paralela para problemas com mais variáveis do que restrições. Na comparação com CPLEX®, o algoritmo proposto paralelo obteve uma eficiência de até 16 vezes maior.

Palavras-chaves: Simplex, CPLEX®, Eficiência Paralela, Escalabilidade Paralela, Programação Linear.

Abstract

This work presents a scalable and efficient parallel implementation of the Standard Simplex algorithm in the multicore architecture to solve large scale linear programming problems. We present a general scheme explaining how each step of the standard Simplex algorithm was parallelized, indicating some important points of the parallel implementation. Performance analysis were conducted by comparing the sequential time using the Simplex tableau and the Simplex of the CPLEX® IBM. The experiments were executed on a shared memory machine with 24 cores. The scalability analysis was performed with problems of different dimensions, finding evidence that our parallel standard Simplex algorithm has a better parallel efficiency for problems with more variables than constraints. In comparison with CPLEX®, the proposed parallel algorithm achieved a efficiency of up to 16 times better.

Key-words: Simplex, CPLEX®, Parallel Efficiency , Parallel Scalability, Linear Programming.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

API	*Application Programming Interface*
CPU	Unidade Central de Processamento
CM-2	*Connection Machine-2*
COIN-OR	*Computational Infrastructure for Operations Research*
f.o.	Função Objetivo
FSB	*Front Side Bus*
MP	MasPar
MIMD	*Multiple Instruction, Multiple Data streams*
MISD	*Multiple Instruction, Single Data streams*
MIT	Massachusetts Institute of Technology
MPI	*Message Passing Interface*
NUMA	*Non-uniform memory access*
OpenMP	*Open Multi-Processing*
OpenMP ARB	*OpenMP Architecture Review Board*
PL	Programação Linear
SBV	Solução Básica Viável
SIMD	*Single Instruction, Multiple Data streams*
SISD	*Single Instruction, Single Data streams*
SMP	*Symmetric MultiProcessors*
VB	Variáveis Básicas
VLSI	*Very Large Scale Integration*
VNB	Variáveis Não Básicas

Lista de símbolos

A	Matriz com os coeficientes das restrições.
B	Submatriz quadrada de A formada pelas colunas correspondentes às variáveis básicas.
b	Vetor de constantes do lado direito.
c^t	Vetor transposto de dos coeficientes da função objetivo.
c^B	Subvetor de c formado pelos coeficientes das variáveis básicas da função objetivo.
c^N	Subvetor de c formado pelos coeficientes das variáveis não básicas da função objetivo.
E	Eficiência Paralela.
I	Matriz identidade.
N	Submatriz de A formada pelas colunas correspondentes às variáveis não básicas.
S	*Speedup*.
T_P	Tempo Paralelo.
T_S	Tempo Sequencial.
x	Vetor das variáveis de decisão.
x_B	Subvetor de x formado pelos coeficientes das variáveis básicas.
x_N	Subvetor de x formado pelas coeficientes das variáveis básicas.

Sumário

1 Introdução

Esse trabalho propõe uma implementação paralela escalável e eficiente do algoritmo simplex padrão na arquitetura *multicore*(múltiplos núcleos) para problemas de programação linear de grande escala. O algoritmo proposto foi submetido a uma análise de eficiência paralela para problemas com diferentes dimensões. O objetivo é verificar a escalabilidade do algoritmo, investigando em que situação é mais apropriado resolver problemas com a melhor quantidade de variáveis e restrições.

O Simplex é um dos métodos mais utilizados para resolver os problemas de programação linear (HALL, 2010; GOLDBARG; LUNA, 2000). Programação linear (PL) é uma forma matemática de alcançar o melhor resultado para problemas de otimização que tenham seus modelos representados por expressões lineares. Esses problemas são sujeitos a restrições que são modeladas por equações ou inequações lineares. O método Simplex é um procedimento iterativo desenvolvido por George B. Dantzig (1947) (PASSOS, 2009). Foi uma contribuição primordial na área da Pesquisa Operacional, que acabava de se desenvolver. O Simplex tem sido citado como um dos 10 algoritmos com a maior influência sobre o desenvolvimento e prática da ciência e da engenharia no século 20 (DONGARRA; SULLIVAN, 2000). O rápido crescimento do poder computacional, aliado a seu custo decrescente, proporcionou um grande número de implementações computacionais do método, resultando em uma economia de bilhões de dólares para a indústria. O número de operações aritméticas necessárias para encontrar uma solução ótima de um problema linear de larga escala pelo método Simplex é muito grande e não teria êxito sem utilizar ferramentas computacionais de alta velocidade (CANTANE, 2009).

A evolução das ferramentas computacionais sofreu recentemente talvez a maior mudança nos últimos 40 anos. A obtenção de maior desempenho passa, desde a década passada, a não depender somente de um maior poder de processamento do hardware. O constante aumento de velocidade que os *chips single core*[1] foram submetidos, os fez atingir seu limite de desempenho. Esse limite foi causado pelo aumento de aquecimento dos *chips*. Assim, a solução encontrada pela indústria, foi a produção de *chips* multiprocessados, ou seja, processadores com vários núcleos. Com o aumento de núcleos, pôde-se diminuir o ritmo de crescimento da velocidade interna do *clock*, amenizando o problema de aquecimento. A partir de então, tem-se esforçado para dobrar a quantidade de núcleos a cada geração, chamando esse novo tempo de *Era Multicore*(BORKAR, 2007). Processadores multicores permitem que frações do problema sejam executadas simultaneamente pelos diferentes núcleos. Dessa forma, torna-se fundamental a exploração do paralelismo de forma eficiente.

[1] um único núcleo.

Nesse contexto, esse trabalho visa apresentar uma implementação paralela do algoritmo Simplex padrão para processadores *multicore*, afim de solucionar problemas de programação linear de grande escala. Além de mostrar os efeitos causados na escalabilidade paralela ao modificar a quantidade de restrições e variáveis. Assim como, verificar em que momento pode ser mais eficiente resolver problemas com mais variáveis que restrições. Também é realizada uma comparação de desempenho do Simplex padrão com o Simplex do CPLEX®.

1.1 Motivação e Justificativa

O tempo de processamento para grandes problemas de PL é uma grande preocupação para a solução de qualquer sistema matemático de grande escala (PLOSKAS; SAMARAS; SIFALERAS, 2009). A exigência para cálculos mais rápidos e mais eficientes em aplicações científicas tem aumentado significativamente nos últimos anos. Por isso, necessita-se tanto de software, como hardware com desempenhos melhores. A obtenção de alta performance não depende somente de utilizar dispositivos de hardware mais rápidos, mas também em melhorias na arquitetura dos computadores e técnicas de processamento. Por isso, como resultado da demanda por maior desempenho, menor custo, e alta produtividade, nos últimos anos se assistiu a uma cada vez mais crescente aceitação e adoção de processamento paralelo, tanto para a computação de alto desempenho científico, quanto para aplicações de uso geral (TASOULIS et al., 2004; THOMADAKIS; LIU, 2006).

Hoje, o foco dos fabricantes é produzir mais processadores multiprocessados, como uma das principais soluções para amenizar o problema da muralha de energia. A muralha de energia trata-se da proporcionalidade entre o crescimento do consumo de energia dos dispositivos eletrônicos e a sua frequência de operação. Ou seja, o consumo de energia aumenta quadraticamente com a frequência de operação (KOCH, 2005). Para agravar o problema, o número de transistores por *chip* cresceu drasticamente, conforme Moore observou (MOORE, 1965), o que aumenta densidade de componentes eletrônicos além de consumir mais energia, produz mais calor, precisando de mais energia para ser resfriado.

Assim, propõe-se nesse trabalho paralelizar o algoritmo Simplex Padrão aplicado a vários tamanhos de problemas de PL, modificando a quantidade de restrições e variáveis. E, verificar em que momento pode ser mais viável resolver problemas com mais variáveis que restrições, ou vice-versa. Além disso, se propõe também realizar a análise de escalabilidade do algoritmo Simplex Padrão. Isso decorre diante da necessidade de obtenção de melhores desempenhos nas aplicações científicas na área de programação linear, aplicados a problemas de grande escala, juntamente com o avanço de softwares e hardwares paralelos, bem como da necessidade de reduzir o aquecimento dos *chips* pelo aumento do número de núcleos de processamento.

1.2 Objetivos

O objetivo principal deste trabalho consiste em analisar a Escalabilidade e a Eficiência Paralela do Algoritmo Simplex Padrão na Arquitetura *Multicore*, verificando, em que momento pode ser mais viável resolver problemas com mais variáveis que restrições, ou vice-versa. Essa metodologia será desenvolvida através da implementação da versão serial e paralela do algoritmo Simplex padrão na forma tabular. Em seguida, será comparado o desempenho desses algoritmos com o CPLEX® e finalmente será verificada a escalabilidade dos problemas PL aplicados ao simplex paralelo, tendo como base o CPLEX® e o Simplex Padrão.

1.3 Organização do texto

O presente trabalho está dividido em seis capítulos, organizados da seguinte forma: capítulo 1 - introdução; capítulo 2 - fundamentação teórica, onde se explica o funcionamento do algoritmo do Simplex (seção 2.1), e também é descrita uma breve explanação sobre as arquiteturas paralelas (seção 2.2), assim como aborda conceitos sobre as métricas de análise da escalabilidade paralela (seção 2.3). Capítulo 3 - revisão bibliográfica, onde além de mostrar alguns trabalhos correlatos, também discorre sobre a ferramenta otimização CPLEX®. No capítulo 4, descreve-se em detalhes a implementação do algoritmo paralelo do Simplex padrão. O capítulo 5 são apresentados os resultados experimentais obtidos, a análise de escalabilidade e de *speedup* do algoritmo Simplex paralelo com os dados de problemas de programação linear de grande escala. E finalmente, o último capítulo aborda as considerações finais.

2 Fundamentação Teórica

Este capítulo apresenta o algoritmo do simplex padrão e tabular, as arquiteturas paralelas utilizadas em trabalhos correlatos, o conceito de escalabilidade e as métricas utilizadas para medir escalabilidade paralela.

2.1 Simplex

A Programação linear (PL) baseia-se em encontrar a melhor solução para problemas que sejam modelados por expressões lineares. Os problemas de PL são problemas de otimização os quais a partir de uma função linear, denominada Função Objetivo (f.o.), procura-se encontrar o máximo ou o mínimo desta, respeitando um sistema linear de restrições.

Esse tipo de modelagem é importante para os profissionais tomarem decisões, procurando estabelecer maneiras mais eficientes de utilizar os recursos disponíveis para atingir determinados objetivos. Uma vantagem desse modelo está na eficiência dos algoritmos de solução atualmente existentes, os quais disponibilizam alta capacidade de cálculo, podendo ser facilmente implementados, até mesmo através de planilhas e com o auxílio de microcomputadores pessoais.

O método Simplex é um procedimento iterativo que fornece a solução de qualquer modelo de PL em um número finito de iterações. O rápido crescimento do poder computacional aliado a seu custo decrescente produziu um grande número de implementações computacionais e resultou em uma economia de bilhões de dólares para a indústria.

O Simplex pode ser representando em uma forma tabular, em que todos os dados são colocados em uma tabela, chamada de *Tableau*. Para ter uma melhor compreensão do algoritmo Simplex na forma tabular, será mostrado primeiramente alguns aspectos matemáticos essenciais do método Simplex padrão, e logo após, será exposto como esses aspectos estão organizados na tabela. Ressaltando, que o método do Simplex padrão na forma tabular foi utilizado para a paralelização nesse trabalho.

A seguir, são apresentadas as duas formas que devem ser utilizadas para modelar os problemas de PL, logo em seguida, os conceitos matemáticos do método Simplex, como é feito a organização do Simplex na forma *tableau* e, por último, uma breve explicação sobre o método Simplex Revisado.

2.1.1 Forma Padrão e Canônica

O problema de PL pode ser modelado de duas maneiras: na forma padrão ou canônica. A forma padrão é aquela em que as restrições estão expressas por meio de equações lineares:

$$\begin{aligned}
&\text{Otimizar } z = c_1x_1 + c_2x_2 + \cdots + c_nx_n \\
&\text{sujeito a} \quad a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1, b_1 \geq 0 \\
&\qquad\quad a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2, b_2 \geq 0 \\
&\qquad\quad \vdots \qquad \vdots \qquad\qquad \vdots \qquad \vdots \\
&\qquad\quad a_{m_1}x_1 + a_{m_2}x_2 + \cdots + a_{mn}x_n = b_m, b_m \geq 0 \\
&\qquad\qquad\qquad\qquad x_1 \geq 0, x_2 \geq 0, \ldots, x_n \geq 0
\end{aligned} \tag{2.1}$$

Em termos de matrizes, pode-se representar a forma padrão da programação linear como:

$$\begin{aligned}
&\textbf{Forma Padrão} \\
&\text{Otimizar} \quad Z = c^t x \\
&\text{sujeito a} \quad Ax = b, b \geq 0 \\
&\qquad\qquad\quad x \geq 0
\end{aligned} \tag{2.2}$$

onde:

$\mathbf{A} \in \mathbb{R}^{m \times n}$, $m < n$, uma matriz com os coeficientes das restrições.

$\mathbf{x} \in \mathbb{R}^n$, um vetor das variáveis de decisão.

$\mathbf{b} \in \mathbb{R}^m$, vetor de constantes do lado direito.

$\mathbf{c}^t$ é o vetor transposto de c, o qual contém os coeficientes da função objetivo, onde $c \in \mathbb{R}^n$

O termo *otimizar* está sendo usado para representar a possibilidade de maximizar ou minimizar uma função objetivo. A forma padrão é particularmente importante porque permite resolver sistemas lineares, ao invés de inequações, e, além de ser, a forma utilizada pelo método Simplex. Contudo, é normal encontrar problemas que sejam mais facilmente modelados na forma canônica:

$$\begin{aligned}
&\textbf{Forma Canônica} \\
&\text{Otimizar} \quad Z = c^t x \\
&\text{sujeito a} \quad Ax \left\{ \begin{array}{c} \leq \\ \geq \end{array} \right\} b \\
&\qquad\qquad\quad x \geq 0
\end{aligned} \tag{2.3}$$

Um mesmo modelo de PL, pode ser reescrito em qualquer uma das formas básicas apresentadas. Esse processo de tradução é realizado através das seguintes operações elementares:

Mudança no Critério de Otimização: A transformação do problema de maximização para minimização, ou vice-versa, consiste na seguinte relação:

Maximizar $(f(x))$ = Minimizar $(-f(x))$ e
Minimizar $(f(x))$ = Maximizar $(-f(x))$.

Ocorrência de Desigualdades nas Restrições: A transformação de desigualdade em igualdade consiste em analisar 2 casos:

- Caso em que a restrição é uma inequação de menor ou igual, deve-se adicionar uma variável de folga x_{n+1} não negativa. Exemplo:

$$x_1 + x_2 + \cdots + x_n \leq b$$
$$x_1 + x_2 + \cdots + x_n + x_{n+1} = b,\ x_{n+1} \geq 0$$

- Caso em que a restrição é uma inequação de maior ou igual, deve-se subtrair uma variável de folga x_{n+1} não negativa. Exemplo:

$$x_1 + x_2 + \cdots + x_n \geq b$$
$$x_1 + x_2 + \cdots + x_n - x_{n+1} = b,\ x_{n+1} \geq 0$$

Ocorrência de Igualdade nas restrições: Uma restrição com igualdade é equivalente a duas desigualdades simultâneas:

$$x_1 + x_2 + \cdots + x_n = b \begin{cases} x_1 + x_2 + \cdots + x_n \leq b \\ x_1 + x_2 + \cdots + x_n \geq b \end{cases}$$

Ocorrência de variáveis livres: Transformar uma variável livre ($x_j \in \mathbb{R}$), em variável não-negativa. Nesse caso, a mudança exigirá a substituição da variável em transformação por duas variáveis auxiliares, ambas maiores ou iguais a zero, mas cuja a soma é igual à variável original, ou seja:

$$x_n = x_n^1 - x_n^2 \ e\ x_n^1 \geq 0, x_n^2 \geq 0.$$

Constante do Lado Direito Negativa: Neste caso multiplica-se toda a equação por (-1) trocando os sinais de todos os coeficientes. No caso de desigualdade, a multiplicação por (-1) inverte o seu sentido. Por exemplo:

$$x_1 - x_2 + x_3 \leq -b \xrightarrow{-1} -x_1 + x_2 - x_3 \geq b$$

2.1.2 O método Simplex

O Simplex é um algoritmo que se baseia em ferramentas da álgebra linear para determinar, por meio de um método iterativo, a solução ótima de um Problema de Programação Linear. Em termos gerais, o algoritmo parte de uma solução inicial válida do sistema de equações que constituem as restrições do problema de PL. A partir da solução inicial são identificadas novas soluções viáveis de valor igual ou melhor que a corrente.

Existem duas etapas na aplicação do método Simplex, após obter o modelo de PL na forma padrão:

Etapa A: Teste de otimalidade da solução ou identificação de uma solução ótima;

Etapa B: Melhoria da solução ou obtenção de solução básica viável (SBV) melhor que a atual.

Em seguida Será descrito o método Simplex para um problema de minimização, o qual, é análogo ao de maximização, com uma diferença apenas no critério de otimalidade. É importante ressaltar que o problema deve estar na forma padrão.

2.1.2.1 Equações Básicas

Considere-se o problema de Programação Linear de minimização na forma padrão apresentada em (2.25), onde o posto da matriz $\mathbf{A}$ é m. Agora, particione a matriz $\mathbf{A}$ em duas submatrizes $\mathbf{B}$ e $\mathbf{N}$

$$\mathbf{A} = [\mathbf{B} \quad \mathbf{N}] \tag{2.4}$$

onde $\mathbf{B}$ é uma submatriz quadrada de ordem m e inversível de $\mathbf{A}$, formada por vetores colunas linearmente independentes. As variáveis associadas a essas colunas denominam-se *variáveis básicas* (VB). A submatriz $\mathbf{N}$ de ordem $m \times (n-m)$ de $\mathbf{A}$ constitui o restante das colunas da matriz $\mathbf{A}$. As variáveis associadas as colunas de $\mathbf{N}$ são denominada *variáveis não básicas* (VNB).

O vetor $\mathbf{x}$ também é particionado nos vetores $\mathbf{x}_B$ e $\mathbf{x}_N$

$$\mathbf{x} = \begin{pmatrix} \mathbf{x}_B \\ \mathbf{x}_N \end{pmatrix} \tag{2.5}$$

onde $\mathbf{x}_B$ é um vetor de ordem $m\ x\ 1$ formado pelos coeficientes das variáveis básicas e $\mathbf{x}_N$ é um vetor de ordem $(n-m)\ \times\ 1$ formado pelos coeficientes das variáveis não básicas.

Da mesma forma o vetor $\mathbf{c}$ é particionado nos vetores $\mathbf{c}_B$ e $\mathbf{c}_N$

$$\mathbf{c} = [\mathbf{c}_B \quad \mathbf{c}_N]^t. \tag{2.6}$$

onde $\mathbf{c}_B$ são os coeficientes das variáveis básicas da função objetivo de ordem $m\ \times\ 1$, e $\mathbf{c}_N$ são os coeficientes das variáveis não básicas da função objetivo de ordem $(n-m)\ \times\ 1$.

Com as partições descritas anteriormente, o problema de minimização pode ser reescrito na forma:

$$\begin{array}{ll} \text{Minimizar} & Z = c_b^t x_B\ +\ c_N^t x_N \\ \text{sujeito a} & Bx_B\ +\ Nx_N = b, b \geq 0 \\ & x_B \geq 0, x_N \geq 0 \end{array} \tag{2.7}$$

Da equação (2.7), tem-se que:

$$Bx_B\ +\ Nx_N = b\ \Rightarrow\ x_B = B^{-1}b\ -\ B^{-1}Nx_N \tag{2.8}$$

Esta equação mostra que as variáveis não básicas $\mathbf{x}_N$ determinam os valores das variáveis básicas $\mathbf{x}_B$, de modo que o sistema seja satisfeito. Uma solução particular é dita **básica**, quando o valor das variáveis não básicas é zero. Assim:

$$x_B = B^{-1}b,\ x_N = 0. \tag{2.9}$$

quando $\mathbf{x}_B$ é não negativo, a solução é dita Solução Básica Viável (SBV).

Substituindo a equação (2.8) na função objetivo (2.7), tem-se:

$$z = c_b^t(B^{-1}b\ -\ B^{-1}Nx_N)\ +\ c_N^t x_N = c_b^t B^{-1}b\ -\ c_b^t B^{-1}Nx_N\ +\ c_N^t x_N, \tag{2.10}$$

logo:

$$z = c_b^t B^{-1}b\ +\ (c_N^t\ -\ c_b^t B^{-1}N)x_N \tag{2.11}$$

Em uma SBV, ou seja, quando $\mathbf{x}_N = 0$, a equação (2.11) se torna

$$\overline{z} = c_b^t B^{-1}b \tag{2.12}$$

que representa o valor da função objetivo para qualquer SBV.

Reescrever a função objetivo em função das variáveis não básicas é importante para mostrar como a função se comporta para cada troca de base. Seja $\mathbf{Y} = B^{-1}N$, logo a equação (2.8) pode ser escrita como:

$$x_B = B^{-1}b \ - \ Yx_N = B^{-1}b \ - \ \sum_{j \in K} y^j x_j \tag{2.13}$$

onde y^j é a coluna j da matriz $\mathbf{Y}$ e $\mathbf{K}$ é o conjunto de índices das VNB. Tomando a função objetivo novamente e substituindo a equação (2.13) nela. Tem-se:

$$Z = c_b^t x_B \ + \ c_N^t x_N \ = \ c_b^t(B^{-1}b \ - \ \sum_{j \in K} y^j x_j) \ + \ \sum_{j \in K} c_j x_j \ = \ c_B^t B^{-1} b \ + \ \sum_{j \in K}(c_j \ - \ c_b^t y^j)x_j. \tag{2.14}$$

Por (2.12), obtêm-se:

$$Z = \overline{z} \ + \ \sum_{j \in K}(c_j \ - \ c_b^t y^j)x_j. \tag{2.15}$$

E, definindo a componente:

$$d_j = c_j \ - \ c_b^t y^j, \tag{2.16}$$

y^j pode ser obtido por $y^j = B^{-1}a^j, \forall j \in K$, então tem-se que:

$$d_j = c_j \ - \ c_b^t B^{-1} a^j, \tag{2.17}$$

onde a^j é a coluna j de $\mathbf{A}$. Com isso obtém-se:

$$Z = \overline{z} + \sum_{j \in K} d_j x_j \tag{2.18}$$

A equação (2.18) permite escrever a função objetivo em função das VNB. Como para uma SBV, $x_j = 0, \forall j \in K$, então o valor da função objetivo é Z = $\overline{z}$.

2.1.2.2 Teste de otimalidade - Entrada na Base

O teste de otimalidade serve para verificar se uma determinada solução básica viável é ótima ou não. Caso seja ótima, o método termina. Se não, procura-se uma VNB para entrar na base, de modo que o valor da função objetivo diminua.

Em uma SBV em que todas as variáveis não básicas são nulas, o método Simplex escolhe somente uma para assumir valor positivo. Dessa forma, a f.o. pode ser reescrita sem o somatório (2.18), assim:

$$Z = \overline{z} \ + \ d_j x_j. \tag{2.19}$$

Pela equação (2.19) pode-se estabelecer um dos critérios para melhoria da função objetivo. Se o valor do termo d_j for estritamente negativo, então, quando x_j assume um valor positivo, o valor da da f.o. fica reduzido de $d_j x_j$. Logo, x_j é uma candidata a entrar na base. No entanto, se $d_j > 0$ e x_j elevar o seu valor, isso significa que o valor de Z aumentaria, e isto não é conveniente para o problema de minimização e neste caso, x_j não é candidata para entrar na base. Então, se $d_j \geq 0, \forall j \in K$, não é possível melhorar o valor de Z. Neste caso, a SBV atual é ótima, com valor ótimo de f.o. sendo $Z = \overline{z}$.

O termo $d_j = c_j - c_b^t B^{-1} a^j$ é chamado de coeficiente de *custo reduzido*, pois ele fornece uma medida de decréscimo de Z quando x_j assume um valor positivo.

Um critério bastante utilizado para se escolher uma VNB para entrar na base é tomar a componente mais negativa do custo reduzido, isto é:

$$d_k = min_{j \in K}\{d_j, d_j < 0\}, \tag{2.20}$$

uma vez que a VNB associada, x_k, é a que mais contribui para diminuir o valor da função objetivo na SBV atual.

2.1.2.3 Teste da Razão - Saída da Base

O teste de razão serve para verificar o quanto a variável x_k pode crescer sem interferir na viabilidade da solução, ou seja, todas as variáveis básicas devem permanecer não negativas. Para isso, o método Simplex determina qual VB deve deixar a base, tornando-se não básica. Além disso, o Simplex troca uma coluna da matriz **N** por uma coluna da matriz **B** em cada iteração.

Como somente a variável x_k irá entrar na base e, continuando assim, as demais VNB nulas, somente x_k afetará no valor atual das VB, ou seja, reescrevendo a equação (2.13):

$$x_B = B^{-1} b - y^k x_k, , \forall k \in K. \tag{2.21}$$

A expressão (2.21) mostra que se existir alguma componente $y_i^k < 0$, onde I é o conjunto de índices das VB, então o x_B^i associado pode crescer indefinidamente com o valor de x_k. Se existir, $y_i^k > 0$, então o valor de x_B^i decresce a medida que o valor de x_k é elevado. Ressalta-se que uma solução básica é dita viável quando nenhum dos componentes de x_B é negativo, isto é:

$$B^{-1} b_i - y_i^k x_k \geq 0 \Rightarrow B^{-1} b_i \geq y_i^k x_k \Rightarrow x_k \leq \frac{B^{-1} b_i}{y_i^k} \tag{2.22}$$

a nova variável básica x_k só poderá crescer até que alguma componente x_B^i seja reduzida a zero, o que corresponde ao mínimo entre todos os $\frac{B^{-1}b}{y_i^k}, y_i^k > 0$. Seja x_s^k essa componente, então:

$$x_B^s = min_{y_i^k} \left\{ \frac{B^{-1}b}{y_i^k}, y_i^k > 0 \right\}, \forall i \in I. \tag{2.23}$$

2.1.2.4 Atualização - Obter uma nova SBV

Pelo teste da otimalidade, sabe-se que variável x_k deve entrar na base e, pelo teste da razão, a variável x_s, deve sair da base. A seguir demonstra-se como proceder com a atualização dos elementos usados pelo Simplex.

- A nova matriz $\mathbf{B}$ é obtida da $\mathbf{B}$ anterior trocando a coluna associada a variável de saída, x_B^s, pela coluna da variável que entrou na base, x_k, da matriz $\mathbf{A}$;
- A nova matriz $\mathbf{N}$ é obtida da anterior, com a coluna associada a variável, x_k, substituída pela coluna da variável x_B^s;
- Os novos componentes $\mathbf{c}_B$ e $\mathbf{c}_N$ seguem as mesmas trocas de $\mathbf{B}$ e $\mathbf{N}$ isto é, o coeficiente c_B^s é substituído por c_k em $\mathbf{c}_B$ e o coeficiente c_k de $\mathbf{c}_N$ é substituído por c_B^s;
- Os novos componentes $\mathbf{x}_B$ e $\mathbf{x}_N$ seguem o mesmo princípio, isto é, x_B^s é substituído por x_k em $\mathbf{x}_B$ e x_k e substituído por x_B^s em $\mathbf{x}_N$;
- $x_B = B^{-1}b \Leftrightarrow Bx_B = b$, ou seja, o novo valor das VB pode ser obtido pela resolução do sistema linear. Contudo, x_B também pode ser obtido diretamente da equação (2.21), com o valor de x_k calculado pelo teste da razão;
- O novo valor de Z é obtido diretamente da equação (2.12), com os novos valores de c_B e das variáveis básicas x_B.

2.1.2.5 O Algoritmo Simplex

O algoritmo Simplex descreve uma sequência de ações que devem ser realizadas para obter a solução de um problema de PL, baseados nos critérios estabelecidos anteriormente. Resumidamente, as etapas são:

1. Determinar uma SBV inicial;
2. Aplicar o teste de otimalidade: se a solução for ótima, então o algoritmo deve parar; se não, deve-se determinar uma variável não básica x_k que deverá entrar na base;
3. Aplicar o teste da razão para determinar qual variável básica x_B^s deve sair da base;

4. Obter uma nova SBV e retornar para a segunda etapa.

2.1.3 O Simplex *Tableau*

Quando os cálculos do método Simplex, para problemas pequenos, precisam ser feitos manualmente é desejável seguir um procedimento organizado. Os dados são colocados em uma tabela, chamada de *tableau*, onde são mostradas, em cada iteração, a solução básica viável, as varáveis básicas e as variáveis não básicas, bem como o valor da f.o..

Nesta tabela são efetuadas as operações de pivoteamento: trata-se de encontrar a VNB que deve entrar na base (teste de otimalidade) e encontrar a VB que deve sair da base (teste da razão). Para isso, é necessário que o problema esteja no formato padrão, que exista uma matriz identidade como matriz básica B e que a função objetivo esteja em função das VNB, pois as VB são zeros. O quadro inicial do Simplex *Tableau* pode ser visto na tabela 1.

Tabela 1: Quadro Inicial do Simplex *Tableau*

	VNB	**VB**	**b**
$\mathbf{c}_B$	**Y**	I	$\bar{b}$
Z	$-d$	0	$\overline{z}$

No quadro inicial foi usado o termo $-d$ pois a equação da f.o. foi rescrita como, $z - c^t x = 0$. As outras notações usadas são:

- $d_j = c_j - c_b^t y^j, j \in K$, onde y^j é a coluna j de $Y = B^{-1}N$ ($d_j = 0$ para toda VB x_j).
- $\bar{b} = B^{-1}b \rightarrow$ valor da SBV atual.
- $\overline{z} = c_t^b \bar{b} \rightarrow$ valor da f.o. atual.
- I é a matriz identidade que é, também, a matriz base.

Considere o seguinte problema de maximização de PL:

$$\begin{aligned} &\text{Maximizar } Z = c_1x_1 + c_2x_2 + \cdots + c_nx_n \\ &\text{sujeito a} \quad a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \leq b_1 \\ &\qquad\qquad a_{21}x_2 + a_{22}x_2 + \cdots + a_{2n}x_n \leq b_2 \\ &\qquad\qquad \vdots \qquad\qquad\qquad\qquad \vdots \\ &\qquad\qquad a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \leq b_m \\ &\qquad\qquad x_1 \geq 0, x_2 \geq 0, \ldots, x_n \geq 0 \end{aligned} \tag{2.24}$$

Colocando-o na forma padrão utilizando as operações elementares vistas na seção 2.1.1, tem-se:

$$\begin{aligned}
&\text{Maximizar } Z = c_1x_1 + c_2x_2 + \cdots + c_nx_n + 0x_{n+1} + 0x_{n+2} + \cdots + 0x_{n+s} \\
&\text{sujeito a} \quad a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n + x_{n+1} + 0 + 0 + \cdots + 0 = b_1 \\
&\qquad\quad a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n + 0 + x_{n+2} + 0 + \cdots + 0 = b_2 \\
&\qquad\quad \vdots \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
&\qquad\quad a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n + 0 + 0 + 0 + \cdots + x_{n+s} = b_m \\
&\qquad\quad x_1 \geq 0, x_2 \geq 0, \ldots, x_{n+s} \geq 0
\end{aligned} \tag{2.25}$$

Observe que, o SBV inicial é:

$$\overline{x} = (x_1 \;\; x_2 \;\; \cdots \;\; x_n \;\; x_{n+1} \;\; x_{n+2} \;\; \cdots \;\; x_{n+s})^t = (0 \;\; 0 \;\; \cdots \;\; 0 \;\; b_1 \;\; b_2 \;\; \cdots \;\; b_m)^t$$

onde $x_N = (x_1 \;\; x_2 \;\; \cdots \;\; x_n)^t = (0 \;\; \cdots \;\; 0)^t$ e $x_B = (x_{n+1} \;\; x_{n+2} \;\; \cdots \;\; x_{n+s})^t = (b_1 \;\; b_2 \;\; \cdots \;\; b_m)^t$, pois a matriz identidade é obtida com os coeficientes relativos às variáveis $x_{n+1}, x_{n+2}, \ldots, x_{n+s}$. A matriz N é obtida com os coeficientes relativos às variáveis não básicas $x_1, x_2, \ldots, x_n$. Note, também que:

$$c = (c_1 \;\; c_2 \;\; \cdots \;\; c_n \;\; c_{n+1} \;\; c_{n+2} \;\; \cdots \;\; c_{n+s})^t$$

onde $c_B = (0 \;\; 0 \;\; \cdots \;\; 0)^t$, para toda variável básica, e $c_N = (c_1 \;\; c_2 \;\; \cdots \;\; c_n)^t$. Logo, $\overline{z} = 0$ e $d_j = c_j, \forall j \in K$, pois $c_B = 0$. Observe, também que:

$$x_b = \overline{b} = B^{-1}b = Ib = b; Y = B^{-1}N = N.$$

Na linha da f.o. coloca-se a equação em função das VNB, isto é:

$$Z - c_1x_1 - c_2x_2 - \cdots - c_nx_n - 0x_{n+1} - 0x_{n+2} - \cdots - 0x_{n+s} = 0.$$

A tabela 2 2 mostra o quadro inicial do Simplex *tableau* mais detalhado. Perceba que a primeira linha é formada pelos os coeficientes d_j da f.o., os coeficientes das variáveis básicas tem valor zero, e o valor da função é inicialmente zero. Para as colunas referentes às VB note a matriz identidade que representa a matriz básica **B**. As colunas associadas as VNB possuem valores referentes aos coeficientes $a_{ij}, c_j, \forall i = \{1, 2, \cdots, m\}, j = \{1, 2, \cdots, n\}$, e o conjunto dos elementos a_{ij} fazem parte da matriz **N**.

2.1.3.1 Teste de Otimalidade

Relembrando a equação (2.19):

$$Z = \overline{z} \; + \; d_jx_j. \tag{2.26}$$

Tabela 2: Quadro Inicial mais detalhado do Simplex *Tableau*.

	Z	$\mathbf{x}_1$	$\mathbf{x}_2$	$\cdots$	$\mathbf{x}_n$	$\mathbf{x}_{n+1}$	$\mathbf{x}_{n+2}$	$\cdots$	$\mathbf{x}_{n+s}$	**b**
Z	1	$-c_1$	$-c_2$	$\cdots$	$-c_n$	0	0	$\cdots$	0	0
$\mathbf{x}_{n+1}$	0	a_{11}	a_{12}	$\cdots$	a_{1n}	1	0	$\cdots$	0	b_1
$\mathbf{x}_{n+2}$	0	a_{21}	a_{22}	$\cdots$	a_{2n}	0	1	$\cdots$	0	b_2
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\mathbf{x}_{n+s}$	0	a_{m1}	a_{m2}	$\cdots$	a_{mn}	0	0	$\cdots$	1	b_m

foi visto que no problema de minimização o critério para uma VNB entrar na base é que $d_j < 0$, logo para o problema de maximização esse critério seria o oposto, ou seja, $d_j > 0$. Se o coeficiente de custo reduzido for estritamente positivo fará que aumente o valor da f.o..

Assim para o problema no Simplex *Tableau* tem-se que:

$$Z = \overline{z} \ + \ d_j x_j \ \Rightarrow \ Z \ - \ \overline{z} \ - \ d_j x_j = 0. \tag{2.27}$$

portanto, para ser candidata a entrar na base na forma *Tableau* é necessário que $-d_j < 0$. Se o problema fosse de minimização, esse critério seria $-d_j > 0$. Se existirem várias candidatas, pode-se escolher a que possuir coeficiente mais negativo, isto é:

$$-d_k = min_{j \in K}\{-d_j, -d_j < 0\}. \tag{2.28}$$

A VNB x_k associada é a escolhida para entrar na base e a coluna referente a essa variável é chamada de **coluna pivô**. O método do Simplex em sua forma *tableau* de maximização termina quando $-d_j \geq 0, \forall j \in K$ ou $d_j \leq 0$, pois qualquer $x_j \geq 0$ iria piorar o valor de **Z**.

2.1.3.2 Teste da Razão

Para escolha da VB que deve sair da base, relembre a expressão do teste de razão:

$$x_B^s = min_{y_i^k}\left\{\frac{B^{-1}b}{y_i^k}, y_i^k > 0\right\}, \forall i \in I \mid I = [1 \ \ m] \tag{2.29}$$

Esse teste deve ser realizado em todas as linhas imediatamente depois da linha correspondente a f.o.. Os elementos da razão são dados pelos valores da última coluna b pelos elementos positivos correspondentes da coluna do pivô. O menor desses quocientes diz qual a variável básica associada que deve sair. A linha referente a essa variável é chamada de **linha pivô**.

2.1.3.3 Obtenção de uma nova SBV

Para obter um novo quadro, indicando as novas variáveis básicas e seus valores, o novo valor da f.o. e as novas colunas y^j é necessário realizar as operações elementares sobre linhas, descritas nas etapas a seguir:

Etapa 1. Localizar o elemento que está na intersecção da linha pivô l_p e coluna pivô c_p. Esse elemento é o pivô y_p;

Etapa 2. Multiplicar a linha pivô por $\frac{1}{y_p}$, transformado o elemento pivô em 1;

Etapa 3. Fazer a operação elementar (2.30) em todas as outras linhas, inclusive a linha correspondente à f.o., de modo a transformar em zeros todos os outros elementos da coluna pivô, exceto o elemento pivô que continua sendo 1.

$$l_i = l_i \ - \ y_p l_p, \forall i \in I \mid I = [1 \ \ m]. \tag{2.30}$$

As etapas 2 e 3 são operações elementares sobre as linhas do quadro Simplex e correspondem ao método de Gauss-Jordan para resolver sistemas lineares. O objetivo é transformar a coluna referente a variável que irá entrar na base em coluna identidade e retirar a antiga VB modificando os valores, deixando-a sem a coluna identidade.

2.1.3.4 Algoritmo do Simplex *Tableau*

O procedimento para usar o método Simplex *Tableau* é descrito nos passos descritos a seguir:

Passo 1. Com o problema na forma padrão, fazer o quadro inicial, colocando os dados conforme o modelo que foi explicado;

Passo 2. Realizar o teste de otimalidade:

- Se todos os $-c_j \geq 0, \forall j \in K$, então a solução ótima foi alcançada e o método deve parar (**Critério de parada**).
- Caso contrário escolhe-se o menor $-c_j < 0$, por exemplo, c_k, escolhe-se, assim, a coluna pivô associado a x_k para entrar na base;

Passo 3. Realizar o teste da razão para escolher a linha pivô:

$$x_B^s = min_{y_i^k} \left\{ \frac{B^{-1}b}{y_i^k}, y_i^k > 0 \right\}, \forall i \in I \mid I = [1 \ \ m] \tag{2.31}$$

O teste deve ser realizado para todas as linhas, exceto a linha da f.o. A linha associada ao menor quociente deve ser a linha pivô.

Importante: Se nenhum dos elementos abaixo da linha correspondente a f.o. na coluna pivô for positivo, o problema não tem solução finita ótima, assim, o método deve parar.

Passo 4. Multiplicar a linha pivô por $\frac{1}{y_p}$, transformado o elemento pivô em 1;

Passo 5. Fazer a operação elementar :

$$l_i = l_i \ - \ y_p l_p, \forall i \in I \mid I = [1 \ \ m]. \tag{2.32}$$

em todas as outras linhas, exceto a linha do pivô e inclusive a linha correspondente à f.o., de modo a transformar em zeros todos os outros elementos da coluna pivô, isto é, todos exceto o elemento pivô que continua sendo 1.

2.1.4 Simplex Revisado

O método simplex revisado consiste em uma maneira de aplicarmos o método a um problema de programação linear, de forma a reduzir a quantidade de operações realizadas. Se um problema possuir um número de variáveis muito maior que o de restrições, o total de operações em cada iteração é menor, pois trabalha-se com tabelas cuja dimensão é determinada pelo número de restrições. O método revisado tem mais vantagens em modelos, onde há muitos coeficientes nulos nas restrições.

Simplex revisado utiliza dos mesmos princípios do Simplex Padrão, contudo não necessita atualizar toda a tabela em cada iteração. O método simplex revisado consiste nos seguintes passos:

Passo 1 Converter um problema de programação linear do formato canônico para o padrão, como apresentado na seção 2.1.1.

Passo 2 Considerando a equação descrita em 2.11, calcule o coeficiente de custo relativo $r = c_N^t - c_b^t B^{-1} N$. Se $r \geq 0$, então a solução é ótima. Caso contrário:

Passo 3 Selecionar a coluna a^k para entrar na base selecionando o coeficiente de custo relativo mais negativo. Calcular $y^k = B^{-1} a^k$ que expressa o novo vetor coluna a_k na base nova.

Passo 4 Se não existe nenhum $y_i^k \geq 0$ pare, pois o problema é ilimitado. Caso contrário, calcule os quocientes $\frac{B^{-1}b}{y^k}$, para determinar qual variável irá sair da base. Ou seja:

$$x_B^s = min_{y_i^k} \left\{ \frac{B^{-1}b}{y_i^k}, y_i^k > 0 \right\}, \forall i \in I. \tag{2.33}$$

onde I é o conjunto de índices das variáveis básicas.

Passo 5 Atualize B^{-1}, efetuando pivoteamento gaussiano em torno e y_s^k. Calcule a nova solução corrente $x_B = B^{-1}b$ e retorne ao passo 2.

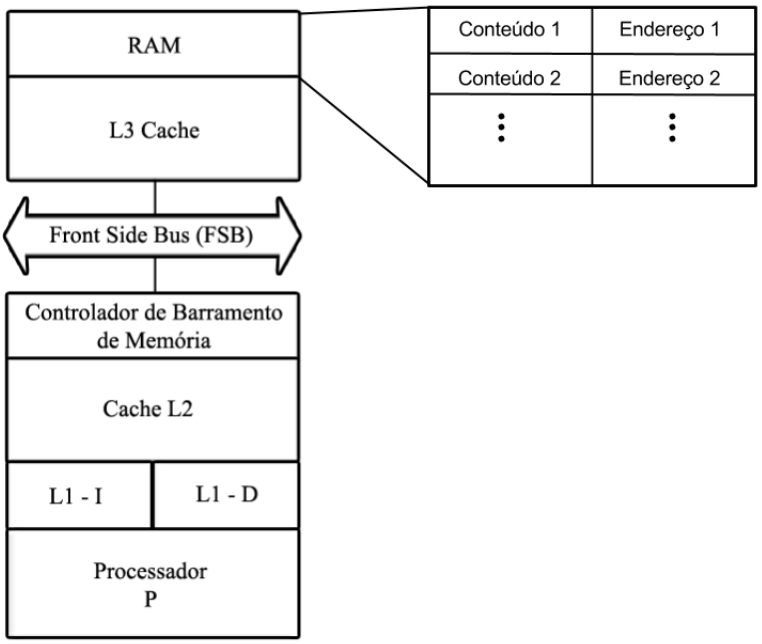
2.2 Arquiteturas Paralelas

O Hardware e software paralelos foram desenvolvidos tendo como base o tradicional hardware e software sequencial[1]. Assim, para entender melhor o estado atual de sistemas paralelos, será feita uma breve revisão em alguns aspectos dos sistemas sequenciais, como também, uma sucinta descrição sobre hardware e software paralelo e, posteriormente, será mostrado como avaliar o desempenho de um programa paralelo.

2.2.1 Arquitetura de Von Neumann

A arquitetura *clássica* de von Neumann consiste em memória principal, uma unidade central de processamento (CPU), e uma interligação entre a memória e a CPU (PACHECO, 2011). A memória principal consiste de um conjunto de locais, onde cada um é capaz de armazenar instruções e dados. Cada local consiste de um endereço, o qual é usado para acessar à localização e o conteúdo, ou seja, as instruções ou os dados armazenados no local (ver figura 1).

Figura 1: A arquitetura de Von Neumann.



fonte: Adaptado de (LIN; SNYDER, 2008).

[1] hardware e software que executam uma única tarefa de cada vez.

As instruções e os dados são transferidos entre a CPU e a memória através de um barramento chamado FSB (*Front Side Bus*). O FSB consiste em um conjunto de vias em paralelo e um hardware específico para controlar o acesso às vias. Na máquina de von Neumann, uma única instrução é executada de cada vez, e esta opera em apenas algumas partes dos dados.

Essa separação da CPU com a memória é, frequentemente, chamada de gargalo de Von Neumann, que consiste na diferença de *clock* entre a CPU e a memória. A taxa de comunicação de dados entre a CPU e a memória, é muito menor do que a taxa com que o processador pode trabalhar. Isso limita a velocidade eficaz de processamento, principalmente quando a CPU é exigida para realizar o processamento de grandes quantidades de dados. Ela é constantemente forçada a esperar por dados que precisam ser transferidos a partir da memória, ou com destino a ela. Como a velocidade da CPU e o tamanho da memória aumentam muito mais rapidamente que a taxa de transferência entre eles, o gargalo se torna mais um problema cuja gravidade aumenta com cada geração de CPU.

Os primeiros computadores eletrônicos digitais foram desenvolvidos na década de 1940, desde então, cientistas e engenheiros de computação têm proposto melhorias para a arquitetura básica de von Neumann. Muitas delas têm o objetivo de reduzir o problema do gargalo de von Neumann, e outras são direcionadas para simplesmente tornar as CPUs mais rápidas. Pode-se citar como exemplo dessas melhorias: a separação dos barramentos de instrução e de dados, que permitem a referência de dados e de instrução paralelamente, sem interferência mútua;a criação de *pipelining*, que é um conjunto de elementos de processamento ligados em série, onde a saída de um elemento é a entrada do próximo. Esses elementos são executados em paralelo, enquanto, por exemplo, uma instrução atual está sendo decodificada, a escrita dos resultados é passada para a memória; O uso de técnicas de especulação, onde os processadores replicam unidades funcionais e tentam executar diferentes instruções de um programa simultaneamente. Ou seja, os processadores sequenciais já possuíam paralelismo intrínseco.

Outra melhoria desenvolvida, muito significativa para amenizar o gargalo, é a memória cache. Uma breve explicação da memória cache será descrita a seguir, visto que apresenta uma grande influência na melhoria da arquitetura de Von Neumann, como também, influenciou diretamente esse trabalho.

2.2.1.1 Memória Cache

A cache é um conjunto de memórias que a CPU pode acessar mais rapidamente do que a memória principal. Ela pode estar localizada no mesmo *chip* com o processador ou pode ser localizada separadamente. A memória cache pode ser acessada muito mais rápido do que um *chip* de memória comum (PACHECO, 2011).

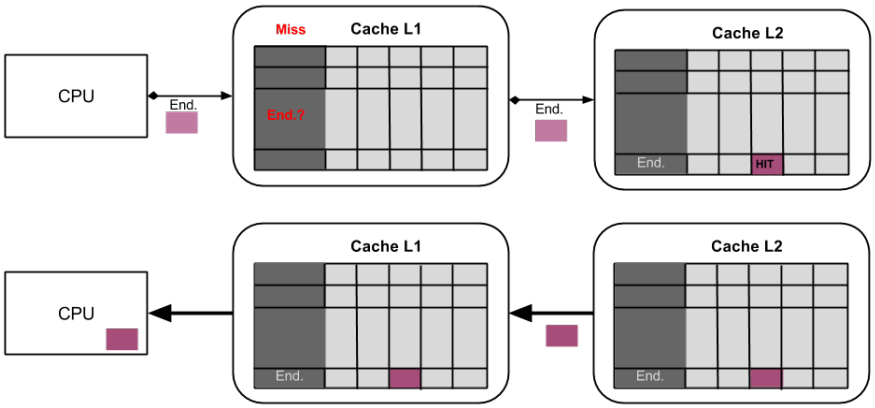
A memória cache segue o princípio de **localidade**, a qual ao acessar uma posição

de memória (instrução ou dados) um programa acessará em uma localização próxima (localidade espacial) no futuro próximo (localidade temporal). A fim de explorar o princípio da localidade, o sistema utiliza efetivamente uma ampla interconexão para acessar os dados e instruções. Isto é, um acesso à memória que irá operar em blocos de dados e instruções, ao invés de instruções individuais e itens de dados individuais. Esses blocos são chamados blocos de cache ou **linhas de cache**.

Conceitualmente, muitas vezes é conveniente pensar em uma cache da CPU como uma única estrutura monolítica. Na prática, porém, o cache é normalmente dividido em níveis: o primeiro nível (L1) é o menor e mais rápido, e níveis mais elevados (L2, L3, ...) são maiores e mais lentos. A maioria dos sistemas atualmente, tem pelo menos dois níveis, e o fato de alguns possuírem três níveis já é bastante comum.

Quando a CPU precisa acessar uma instrução ou dados, ela trabalha o seu caminho para baixo na hierarquia do cache: Inicialmente, verifica o cache de nível 1, em seguida o de nível 2, e assim por diante. Finalmente, se a informação necessária não é encontrada em qualquer um dos caches, ela acessa a memória principal. Quando um cache tem a informação e esta, está disponível, então é chamado de *cache hit* ou apenas *hit*, mas se a informação não estiver disponível, é chamado de *cache miss* ou somente *miss*. *Hit* ou *miss*, são muitas vezes modificadas pelo nível, por exemplo, quando a CPU tenta acessar uma variável, ela pode ter uma L1 *miss* e L2 *hit* (ver figura 2).

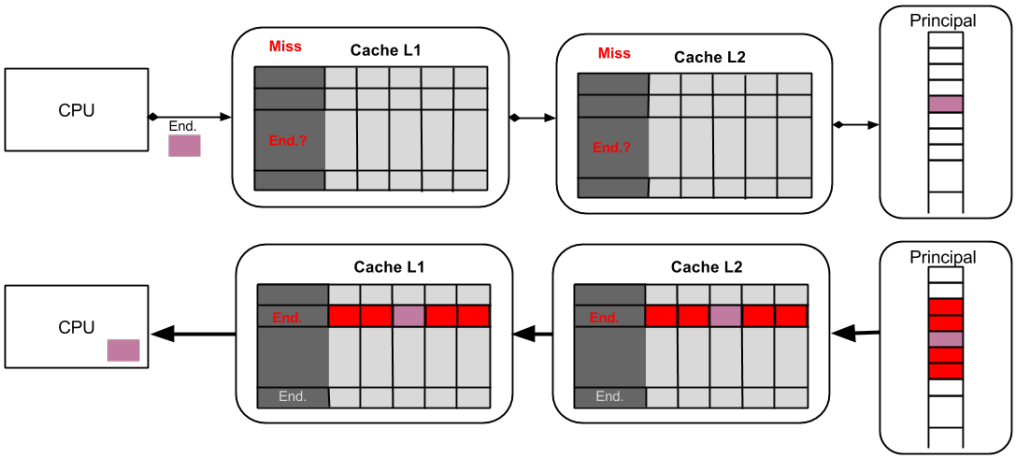
Figura 2: Exemplo de acesso de dados e instruções por meio de memória cache que ocorre um *cache-miss* na *cache* L1 e um *cache-hit* na *cache* L2.



Fonte: Autoria própria.

Quando a CPU tenta ler dados ou instruções, ocorrendo uma leitura-miss em todos os níveis da cache, então será lido da memória principal o bloco de dados ou instruções (a linha de cache). Esse bloco contém as informações necessárias e estas serão armazenadas na cache (ver figura 3). Isso pode parar o processador, enquanto aguarda a memória mais lenta. Ele pode interromper a execução de instruções do programa atual até que os dados ou as instruções necessárias sejam obtidas na memória principal.

Figura 3: Acesso de dados e instruções por meio de memória cache que o dado ou instrução não se encontra na *cache* L1 ou L2.



Fonte: Autoria própria.

2.2.2 Classificação de Computadores Paralelos

Todas essas melhorias feitas na arquitetura de Von Neumann podem claramente ser consideradas hardware paralelo, uma vez que as unidades funcionais são replicadas e os dados/instruções são executados simultaneamente. No entanto, uma vez que esta forma de paralelismo geralmente não é visível para o programador, as melhorias feitas nessa arquitetura, tratam-se de extensões para o modelo básico von Neumann. Para o propósito deste trabalho, hardware paralelo será limitado ao hardware que é visível para o programador. Em outras palavras, se a melhoria pode facilmente modificar seu código fonte para explorá-la, então será considerado o hardware paralelo.

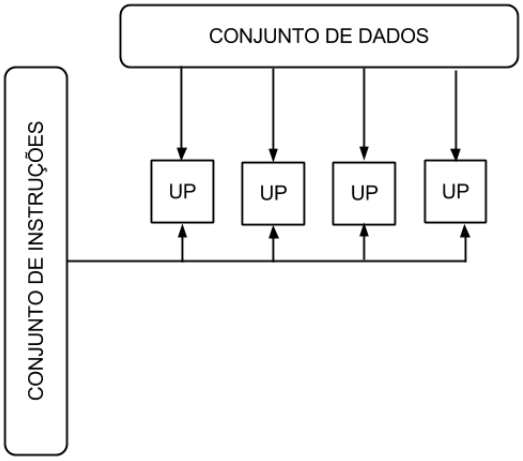
A taxonomia de Flynn (1972) é uma maneira comum de se caracterizar arquiteturas de computadores paralelos. As classes de arquiteturas foram propostas de acordo com a unicidade ou multiplicidade de dados e instruções, conforme descrições a seguir:

SISD - *Single Instruction, Single Data streams*[2] refere-se aos computadores sequenciais com apenas um processador (a clássica máquina seqüencial de Von Neumann), onde há um controle sobre o que está ocorrendo exatamente em cada instante de tempo e é possível de reproduzir o processo passo a passo.

SIMD - *Single Instruction, Multiple Data streams*[3] refere-se aos sistemas paralelos de natureza síncrona, ou seja, todas as unidades de processamento recebem a mesma instrução no mesmo instante de tempo de modo que todas possam executá-la de forma simultânea, mas contendo dados diferentes. A figura 4 apresenta um esquema para esse tipo de arquitetura paralela.

[2] Instrução Única, Dado Único.
[3] Instrução única, Dados múltiplos.

Figura 4: Arquitetura paralela SIMD.



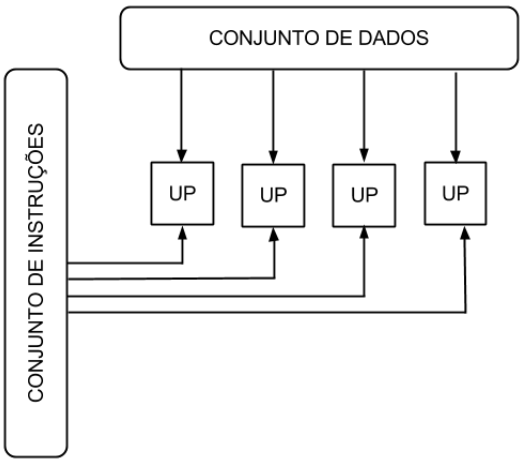
Fonte: Autoria própria.

MISD - *Multiple Instruction, Single Data*[4] são conceitualmente definidas como de múltiplos fluxos de instruções que operam sobre os mesmos dados. Além da dificuldade de implantação deste tipo de sistemas, há ausência de uma plataforma ou paradigma de programação que permita utilizar este tipo de arquitetura eficientemente. Na prática, não é comum encontrar implementações desse modelo.

MIMD - *Multiple Instruction, Multiple Data streams*[5] refere-se aos sistemas paralelos de natureza assíncrona, ou seja, as instruções são geradas por unidades de controle diferentes e cada processador executa sua instrução no seu próprio dado. Eles possuem tanto paralelismo de dados quanto de instruções. É a categoria onde grande parte dos multiprocessadores atuais se encontram. A figura 5 mostra um modelo para essa arquitetura paralela.

[4] Instruções Múltiplas e Dados Únicos
[5] Instrução múltipla, Dados múltiplos.

Figura 5: Arquitetura paralela MIMD.



Fonte: Autoria própria.

2.2.3 Arquiteturas de memória em Computadores Paralelos

A arquitetura MIMD da classificação de Flynn é muito genérica para ser utilizada na prática. Essa arquitetura, teoricamente, poderia executar programas destinados a qualquer outra arquitetura de Flynn, ou seja, todos os outros casos seriam uma sub-classificação do MIMD. Assim, ela é, geralmente decomposta de acordo com a organização de memória. Existem três tipos de sistemas paralelos, diferindo quanto a comunicação, são eles: memória compartilhada, memória distribuída e memória híbrida. Esses sistemas são descritos a seguir:

2.2.3.1 Memória Compartilhada

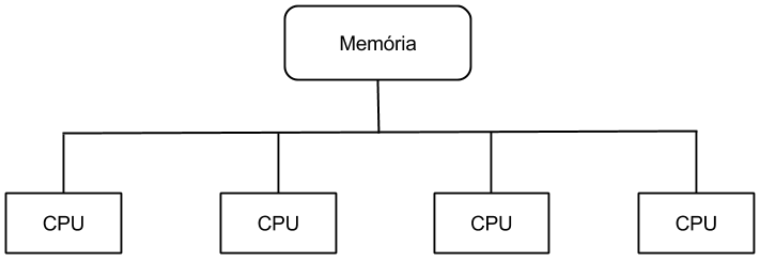
Um sistema de Memória Compartilhada é uma memória que pode ser acessada simultaneamente por múltiplos programas com a intenção de prover comunicação entre eles ou para evitar cópias redundantes. Assim, por exemplo, um processador A desejando comunicar-se com um B deve escrever em uma posição da memória para que B leia desta mesma posição. A seguir será descrito duas das principais classes do sistema de memória compartilhada, SMP e NUMA.

SMPs (*Symmetric MultiProcessors* (Multi-Processadores Simétricos)) é uma classe muito comum desse tipo de sistema, onde dois ou mais processadores idênticos são conectados a uma única memória principal compartilhada. O SMP têm pleno acesso a todos os dispositivos de I/O, além do mais são controlados por uma única instância do sistema operacional, na qual todos os processadores são tratados da mesma forma.

Os processadores trabalham sozinhos compartilhando os recursos de hardware, geralmente eles são iguais, similares ou com capacidades parecidas. Por tratar todos os

processadores de forma igualitária, no multiprocessamento simétrico, qualquer unidade de processamento pode assumir as tarefas realizadas por qualquer outro processador. As tarefas são divididas e também podem ser executadas de modo concorrente em qualquer processador que esteja disponível. Os acessos dos processadores aos dispositivos de entrada e saída, e à memória são feitos por um mecanismo de intercomunicação constituído por um barramento único (ver figura 6).

Figura 6: Arquitetura SMP.



Fonte: Autoria própria.

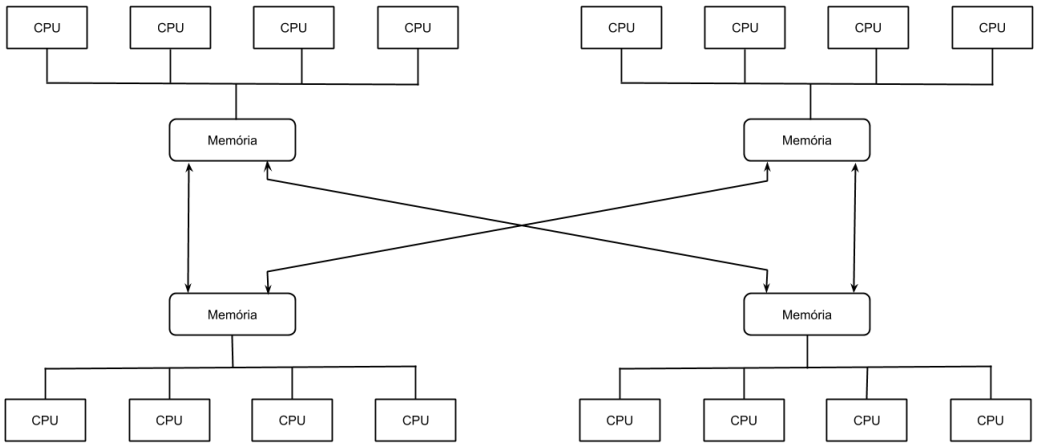
A memória principal da máquina é compartilhada por todos os processadores através de um único barramento que os interliga, de modo que essa comunicação com a memória é nativo. Um dos gargalos desse sistema é o acesso à memória principal ser realizado através de um único barramento, e esse acesso ser serial. O sistema fica limitado a passagem de apenas uma instrução de cada vez pelo barramento, abrindo uma lacuna de tempo entre uma instrução e outra. Contudo, memórias caches junto aos processadores diminuem o tempo de latência entre um acesso e outro à memória principal e ajudam também a diminuir o tráfego no barramento.

Antigamente para utilizar SMP era preciso um hardware específico, placas-mãe com dois ou mais soquetes de CPU e grandes estruturas de servidores conectados. Atualmente com a tecnologia mullticore, as fabricantes já integram tudo isso em apenas um dispositivo físico, também conhecido como processador mullticore.

NUMA (*Non-uniform memory access* (Acesso não Uniforme à Memória)) é um sistema de memória compartilhada em que cada nó desse sistema possui sua própria memória local que pode ser acessada por todos os outros processadores dele. O custo de acesso à memória local é menor que o custo de acesso a dados presentes na memória local de outros processadores, sendo estimado em termos de tempo de acesso e latência. Entretanto, alguns blocos de memória podem ficar fisicamente mais próximos com certos processadores e são naturalmente associados a eles. Dessa forma, o tráfego gerado no meio de comunicação é diminuído, permitindo que mais processadores sejam adicionados ao sistema.

Sistemas de arquitetura NUMA herdam um problema existente nos sistemas de arquitetura SMP: a coerência da memória principal e da memória cache. Em um sistema

Figura 7: Arquitetura NUMA.



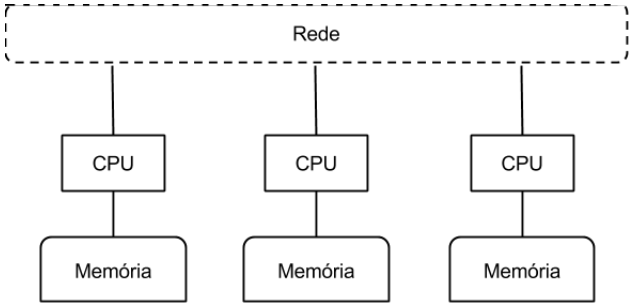
Fonte: Autoria própria.

NUMA com coerência de cache, quando um processador inicia um acesso à memória, se a posição requerida não está na cache deste processador, uma busca na memória é realizada. Se a linha requerida está na porção local da memória principal, ela é obtida por meio do barramento local, se a posição está em uma memória remota, é feita uma requisição de busca à memória remota através do sistema de conexão que interliga os processadores. O dado é então entregue ao barramento local, que é em seguida entregue à cache e ao processador requisitante.

2.2.3.2 Memória Distribuída

Um sistema de Memória Distribuída são sistemas compostos por vários computadores, capazes de operar de forma totalmente desacoplada, e que se comunicam através de operações de entrada e saída (CULLER; GUPTA; SINGH, 1997). Esses sistemas geralmente são compostos por diversas unidades de processamento conectadas por uma rede de dados, onde cada processador possui uma área de memória local apenas. As unidades de processamento podem operar independentemente entre si e se precisarem acessar endereços de memória não-local, esses dados geralmente são transferidos pela rede de uma área de memória para outra. A figura 8 apresenta um exemplo representativo de um computador de memória distribuída.

Figura 8: Arquitetura de memória distribuída.

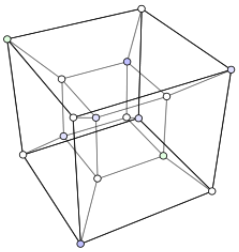


Fonte: Autoria própria.

Em memória distribuída, a ligação entre os processadores é representada através de uma rede de interconexões. Essa rede é composta por: vértices, os quais representam um par processador-memória e as arestas representam as ligações diretas entre estes pares. Existem várias topologias, ou seja, diferentes formas nas quais se pode organizar a interligação entre cada um dos vértices do garfo, duas dessas topologias serão descritas a seguir:

Hipercubo Essa topologia é bastante usada em sistemas paralelos, devido ao seu baixo número de vértices (processadores), o que diminui a latência de memória[6] em relação a outras topologias. A figura 9 apresenta um esquema de topologia em hipercubo.

Figura 9: Topologia em hipercubo.

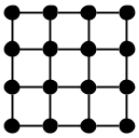


Fonte: Autoria própria.

Malha Essa topologia possui uma boa tolerância a falhas, devido ao grande número de caminhos alternativos entre dois processadores. A figura 10 apresenta um exemplo de topologia em malha.

[6] Latência de memória é o tempo que um processador espera para que um dado requisitado que não esteja em sua memória local fique disponível.

Figura 10: Topologia em malha.



Fonte: Autoria própria.

Dependendo da topologia e da tecnologia utilizadas para interconexão dos processadores, a velocidade de comunicação pode ser tão rápida quanto em uma arquitetura de memória compartilhada. A princípio, não existe uma área de memória compartilhada. Cada computador possui a sua própria memória principal, constituindo assim, uma hierarquia de memória distribuída. Caso um processador necessite de uma informação que está na memória principal de uma outra unidade de processamento, esta informação deve ser explicitamente transferida entre os processadores utilizando-se de um mecanismo de *Message Passing*[7].

A passagem de mensagem é um modelo de programação paralela amplamente utilizado. Seus programas criam múltiplas tarefas, cada uma delas encapsulando dados locais. Cada tarefa é identificada por um nome próprio e interage com outras através de envio e recepção de mensagens entre elas.

Cada tarefa pode realizar operações de leitura ou escrita em sua memória local; operações aritméticas, chamadas de funções locais e, enviar ou receber mensagens. Normalmente, o envio de uma mensagem é instantâneo e ocorre de forma assíncrona, enquanto o recebimento ocorre de forma síncrona e bloqueia o fluxo de execução da tarefa até que a mensagem seja recebida.

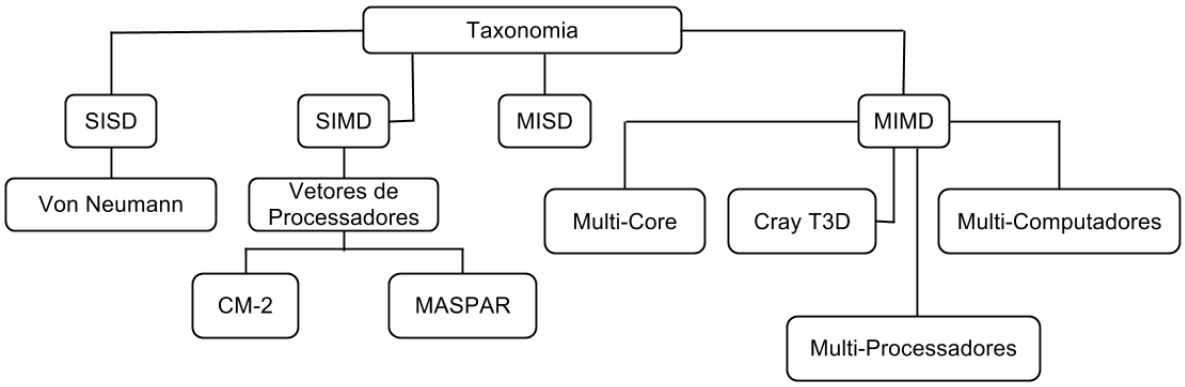
2.2.4 Máquinas Paralelas

O computador paralelo é definido por Almasi e Gottlieb (1989) como uma coleção de elementos de processamento que cooperam e se comunicam para resolver grandes problemas de forma rápida. As máquinas paralelas começaram a surgir com o objetivo de resolver algumas aplicações científicas de maior porte, elas eram projetadas tendo como foco a resolução dos problemas da forma mais rápida possível, sem preocupações com padronização entre arquiteturas.

O diagrama da figura 11 mostra a taxonomia de Flynn e suas ramificações. Algumas máquinas foram criadas baseadas messa taxonomia, como por exemplo, o CM-2 e o MASPAR, ambos constituídos de vetores de processadores, e baseados na arquitetura SIMD.

[7] Passagem de Mensagens

Figura 11: Taxonomia de Flynn e suas ramificações.



Fonte: Autoria própria.

2.2.4.1 CM-2 (*Connection Machine-2*)

O *Connection Machine* (CM) foi uma série de supercomputadores que cresceu a partir da pesquisa Danny Hillis no início de 1980 no MIT. Ele foi originalmente planejado para aplicações em inteligência artificial e processamento simbólico, mas as versões posteriores encontraram maior sucesso no campo da ciência computacional.

O CM-2 era configurável de 4.092 até 65.536 processadores, e com até 512 MB de DRAM, trata-se de uma máquina SIMD com memória distribuída. Seus processadores estão organizados em um hipercubo, onde cada vértice equivale ao que é chamado de *sprint-node*, os quais são formados por 32 processadores não poderosos (1 bit). Assim, por exemplo, utilizando uma estrutura com 65.536 processadores, logo há 2.048 *sprint-nodes* organizados em um hipercubo de 11 dimensões.

2.2.4.2 MP (MasPar)

MasPar Computer Corporation é uma empresa fornecedora de mini-supercomputador que foi fundada em 1987 por Jeff Kalb. O MP-1 era configurável de 1.024 a 16.384 processadores em duas séries (MP1100 a MP1200), sendo uma máquina SIMD de memória distribuída. Os processadores são customizados e estão organizados em uma topologia em malha, com grupos de 16 processadores em malha 4 x 4. A largura de banda desta rede cresce quase linearmente com o número de processadores. O MASPAR (MP) versão 1 utiliza DRAMs de 16 Kbytes por processador, o que dá uma memória total de 256 Mbytes.

2.2.4.3 Cray T3D

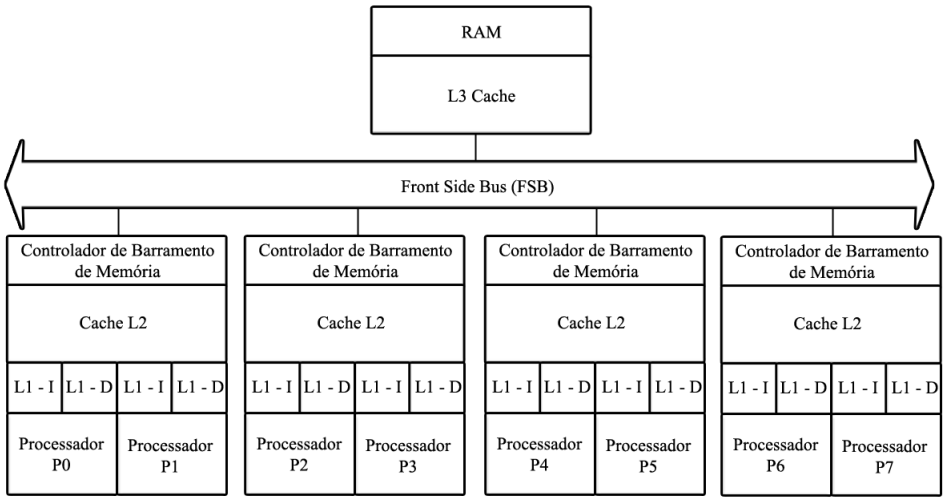
Outra máquina produzida na arquitetura proposta por Flynn foi O Cray-T3D. Ela é considerada a primeira máquina maciçamente paralela da empresa *Cray Research* lançada em 1993. O Cray T3D é uma máquina MIMD de memória compartilhada NUMA.

Nela o processador pode ler ou escrever em memórias não-locais sem conhecimento prévio dos processadores locais a estas, por isso é dita compartilhada. O tamanho da memória é dado pelo número de processadores, multiplicado pelo tamanho da memória local a cada processador, que varia de 16 a 64 Mbytes. Logo, uma máquina com 1024 processadores possui uma memória de até 64 Gbytes.

2.2.4.4 Multicore

A tecnologia *multicore* vem se solidificando no mercado atual cada vez mais. Essa tecnologia vem da arquitetura MIMD de Flynn (ver figura 11). Um processador *multicore* é um componente de computação com duas ou mais unidades independentes de processamento, chamados núcleos. São unidades que lêem e executam as instruções do programa, usando memória compartilhada para a comunicação entre si. Os núcleos podem ou não executar múltiplas instruções ao mesmo tempo, aumentando a velocidade geral da computação paralela. A figura 12 mostra um exemplo de uma arquitetura de múltiplos núcleos. A arquitetura é geralmente um SMP, ou seja, onde dois ou mais processadores idênticos são ligados a uma única memória principal, implementado em um circuito VLSI - *Very Large Scale Integration*[8].

Figura 12: Exemplo de uma arquitetura *multicore*.



fonte:(LIN; SNYDER, 2008)

Em processadores de múltiplos núcleos o sistema operacional trata cada um desses núcleos como um processador diferente. Na maioria dos casos, cada unidade possui seu próprio cache e alguns casos realizam acesso direto e independente à memória principal. Possibilita-se, assim, que as instruções de aplicações sejam executadas em paralelo, ou

[8] VLSI é o processo de criação de circuitos integrados através da combinação de milhares de transistores em um único chip. VLSI surgiu na década de 1970, quando as tecnologias de semicondutores e de comunicação complexos estavam sendo desenvolvidos.

seja, cada processador realiza os cálculos de que é requisitado concorrentemente com o outro, ganhando desempenho. Adicionar novos núcleos de processamento a um processador possibilita que as instruções das aplicações sejam executadas em paralelo, como se fossem 2 ou mais processadores distintos.

Dois ou mais núcleos não somam a capacidade de processamento, mas dividem as tarefas entre si. Por exemplo, um processador de dois núcleos com *clock* de 1.8 Ghz não equivale a um processador de um núcleo funcionando com *clock* de 3.6 Ghz, e sim dois núcleos 1,8 Ghz operando em paralelo.

Os processadores *multicore* tornaram-se fundamental devido à necessidade de solucionar alguns problemas. Dentre eles pode-se destacar: a missão cada vez mais difícil de resfriar processadores *single core* com *clocks* cada vez mais altos; A concentração cada vez maior de transistores em um mesmo circuito integrado. Além dessas e outras limitações dos processadores *single core*, que serão detalhados logo adiante na sessão 2.3.

2.2.4.5 Multi-Computadores/Processadores

A figura 11 apresenta sistemas ramificados pelo MIMD, dois deles são: o multi-processador e o multi-computador. O multi-processador é um conjunto de unidade de processamento, utilizando memória compartilhada; já o multi-computador é um sistema distribuído, ou seja, dois ou mais computadores interconectados, comunicando-se por mensagem em que cada computador pode realizar uma tarefa distinta.

Essas máquinas paralelas foram produzidas a partir da década de 70 e eram caracterizadas pelo custo elevado e pela dificuldade de programação. O programador necessitava ter o conhecimento específico de cada máquina paralela para a qual seriam implementadas as aplicações paralelas, o que aumentava a complexidade do desenvolvimento do programa. Além disso, o rápido decréscimo na relação preço-desempenho de projetos de microprocessadores convencionais levou ao desaparecimento desses supercomputadores no final de 1990.

2.2.5 Software Paralelo

Para resolver um problema, tradicionalmente, o software tem sido escrito como um fluxo serial de instruções, já a programação paralela, por outro lado, faz uso de múltiplos elementos de processamento para resolvê-los. Isso é possível ao quebrar um problema em partes independentes de forma que cada elemento de processamento possa executar sua parte do algoritmo simultaneamente com outros.

Após mostrar as arquiteturas paralelas de hardware, se faz necessário também estudar as arquiteturas paralela de software, pois sem eles não há como utilizar o hardware para realizar trabalho útil. Nesse capítulo, será visto dois softwares que podem ser utilizados

para computação paralela: o MPI baseado em memória distribuída e o OpenMP, utilizado nesse trabalho, baseado em memória compartilhada.

2.2.5.1 MPI

MPI (*Message Passing Interface*) é um sistema de transmissão de mensagens padronizadas, desenvolvido por um grupo de pesquisadores da academia e da indústria para funcionar em uma ampla variedade de computadores paralelos. O padrão define a sintaxe e a semântica de um núcleo de rotinas de biblioteca úteis para uma ampla gama de usuários. Os programas de transmissão de mensagens portáteis são escritos em Fortran 77 ou na linguagem de programação C. Existem várias implementações bem testadas e eficientes de MPI, incluindo algumas que são gratuitas ou de domínio público.

MPI é uma API(*Application Programming Interface*) o qual foi desenvolvido para permitir que os usuários criem programas que podem ser executados de forma eficiente na maioria das arquiteturas paralelas. O processo de projeto é incluído fornecedores, como: IBM, Intel, TMC, Cray, Convex; autores de bibliotecas paralelas e especialistas.

O MPI executa, inicialmente, um número fixo de processos, tipicamente, cada processo é direcionado para diferentes processadores. Os processos podem usar mecanismos de comunicação ponto a ponto, que são operações para enviar mensagens de um determinado processo a outro. Um grupo de processos pode invocar operações coletivas (*collective*) de comunicação para executar operações globais. O MPI é capaz de suportar comunicação assíncrona e programação modular, através de mecanismos de comunicadores (*communicator*) que permitem ao usuário MPI definir módulos que encapsulem estruturas de comunicação interna.

2.2.5.2 OpenMP

OpenMP (Open Multi-Processing) é uma API multiplataforma para multiprocessamento, usando memória compartilhada. Ela consiste de um conjunto de diretivas para o compilador, funções de biblioteca e variáveis de ambiente as quais especificam a implementação de um programa paralelo em C/C++.

A inserção adequada de recursos OpenMP em um programa seqüencial vai permitir que muitas, talvez a maioria, das aplicações se beneficiem da arquitetura paralela de memória compartilhada com o mínimo de modificação no código. Na prática, muitas das aplicações têm considerável paralelismo que podem ser exploradas.

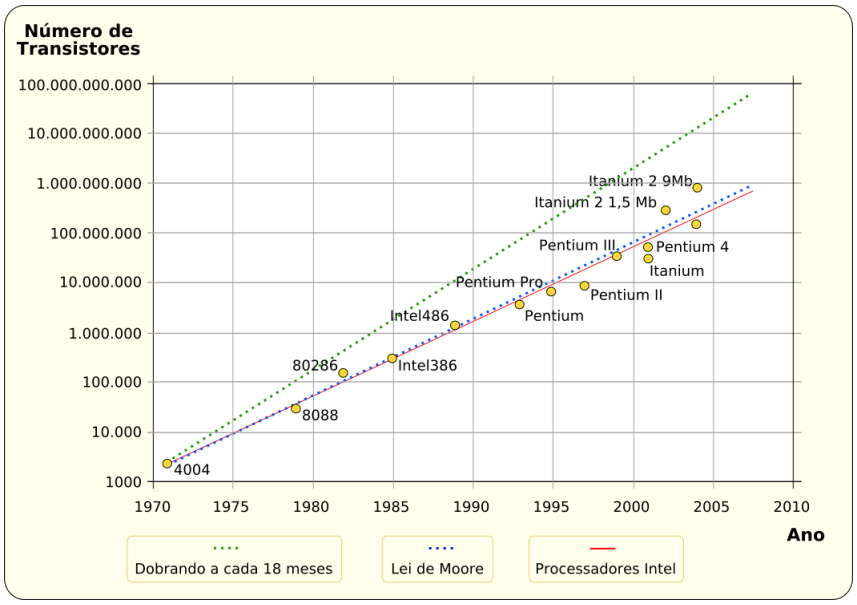
A OpenMP é gerenciada por uma associação sem fins lucrativos, a OpenMP ARB (OpenMP Architecture Review Board), definida por um grupo das principais empresas de hardware e software, incluindo AMD, IBM, Intel, Cray, HP, Fujitsu, Nvidia, NEC, Microsoft, Texas Instruments, Oracle Corporation e outros (ARB, 2013). Os integrantes

da OpenMP ARB compartilham o interesse de uma abordagem portátil, de fácil utilização e eficiente para programação paralela com memória compartilhada.

2.3 Paralelismo e Escalabilidade

Em meados de 1965, o co-fundador da Intel®, Gordon Moore (MOORE, 1965), observou que o número de transistores dos chips teria um aumento de 60%, pelo mesmo custo, a cada período de 18 meses. Inicialmente a lei de Moore não passava de uma observação, mas acabou tornando-se um objetivo para as indústrias de semicondutores. Muitos recursos foram gastos para poder alcançar as previsões de Moore no nível de desempenho. A figura 13 mostra a evolução na quantidade de transistores nos processadores Intel®.

Figura 13: Lei de moore.



fonte: <http://pt.wikipedia.org/wiki/Lei_de_Moore>

A dedicação das grandes empresas em pesquisar e desenvolver processadores com freqüência de *clock* cada vez maiores, fez com que mantivesse por mais de 40 anos a duplicação de quantidade de transistores a cada ano e meio. Contudo, essa duplicação e o aumento da freqüência de operação dos processadores têm sido mais difíceis de serem feitos por diversos motivos.

Um desses motivos diz respeito ao tamanho dos transistores. O funcionamento dos transistores são baseados no uso de eletricidade para controlar o fluxo de elétrons entre os dois extremos (pólos). O *gate*, localizado no centro do transistor, é responsável por permitir ou bloquear o fluxo de elétrons de acordo com o estado lógico do transistor. Depois dos 10 nm, os elétrons passam aleatoriamente por ele, mesmo que o *gate* não esteja

ativo, fazendo com que ele deixe de ser confiável, isso é chamado de *tunneling*. Ao chegar aos 5 nm o problema se torna crônico e a quebra da barreira passa a ocorrer em 50% das vezes, eliminando a possibilidade do uso de qualquer sistema de correção. Em uma escala tão pequena, o *tunneling* ocorre independentemente do material usado, o que elimina a possibilidade de que algum novo material possa estender o processo por mais gerações (MORIMOTO, 2012).

Um dos maiores problemas é a muralha de energia. Existem 3 fatores que estão intimamente ligados: freqüência de operação dos *chips*, energia e a temperatura. Ao aumentar a freqüência de operação faz-se necessária mais energia para poder operar, e conseqüentemente, há maior liberação de calor. Com a miniaturização do transistor, os *chips* atuais são mais densos, e como consequência, aumenta-se os 3 fatores. Em 1993, por exemplo, os processadores Intel® Pentium® possuiam aproximadamente 3 milhões de transistores, enquanto que o processador Intel® Itanium® 2 possuíam cerca de 1 bilhão de transistores. Segundo Koch (2005), se essa taxa continuar, os processadores Intel®, logo produzirão mais calor por centímetro quadrado que a superfície do sol. Por isso que o problema do calor já estabelece difíceis limites para o aumento da frequência.

Nos últimos anos, vem crescendo a quantidade de dispositivos portáteis, como smartphones, tablets, laptops, netbooks e outros. Todos esses dispositivos possuem processadores e memória para processar os dados. Logo, o consumo de tais aparelhos deve ser o mínimo possível para prolongar o tempo da bateria. Dessa forma, o novo desafio da engenharia é trabalhar na criação de dispositivos elétricos que consumam menos energia e produzam menos calor. Freqüências muito acima dos 4 ou 4.5 GHz demandam *water-coolers*[9] ou soluções exóticas de resfriamento, mas dificilmente serão usados pela grande massa de usuários devido ao custo e à complexidade.

Os *chips multicore* podem operar em freqüências menores comparados com os processadores de um núcleo, já que podem realizar mais trabalhos a cada ciclo. Como aumentar a operação, também eleva o consumo de energia, os chips multicores são um dos meios para a solução dos problemas de energia e temperatura. Assim, tem-se esforçado para dobrar a quantidade de núcleos a cada geração, chamando esse novo tempo de *Era Multicore* (BORKAR, 2007) (KOCH, 2005).

Embora a nova era paralela da computação beneficie o processamento de diversas tarefas ao mesmo tempo, a aceleração de uma tarefa ou algoritmo isolado requer, desta vez, um esforço adicional. Para isso é necessário que os programas mudem, e como consequência, os programadores também devem. Como resultado, atualmente, a grande maioria dos algoritmos não suporta paralelismo. Portanto, dentre os muitos desafios da *era multicore*, um dos mais urgentes é verificar se os algoritmos possuem problemas de escalabilidade

[9] *water-coolers* ou sistemas de refrigeração a líquido, são sistemas de resfriamento, o qual utiliza água para amenizar altas temperaturas causados por certos processadores.

de desempenho. Isso significa, a grosso modo, saber se o programa é capaz de usar progressivamente uma maior quantidade de processadores, implicando que os avanços da tecnologia de silício e o maior número de núcleos farão o programa se manter em bom desempenho.

2.3.1 Conceitos

A seguir, serão abordados alguns temas e conceitos de paralelismo relevantes para este trabalho, e posteriormente, serão discutidas as métricas básicas utilizadas na análise de algoritmos paralelos.

2.3.1.1 Threads e Processos

Um processo é basicamente um programa em execução. Associado com cada processo está seu espaço de endereçamento e uma lista de locais da memória a partir de um mínimo até um máximo que o processo pode ler e gravar. O espaço de endereçamento contém o programa executável, os dados do programa e sua pilha de chamadas. Também associado com cada processo está um conjunto de registradores, incluindo o contador de programa, o ponteiro da pilha e outros registradores de hardware e todas as demais informações necessárias para executar o programa (TANENBAUM; WOODHUL, 2000). Vários processos podem ser associados com o mesmo programa, por exemplo, abrindo várias instâncias do mesmo programa geralmente significa mais que um processo está sendo executado. Os processos normalmente se comunicam entre si pelo envio de mensagens.

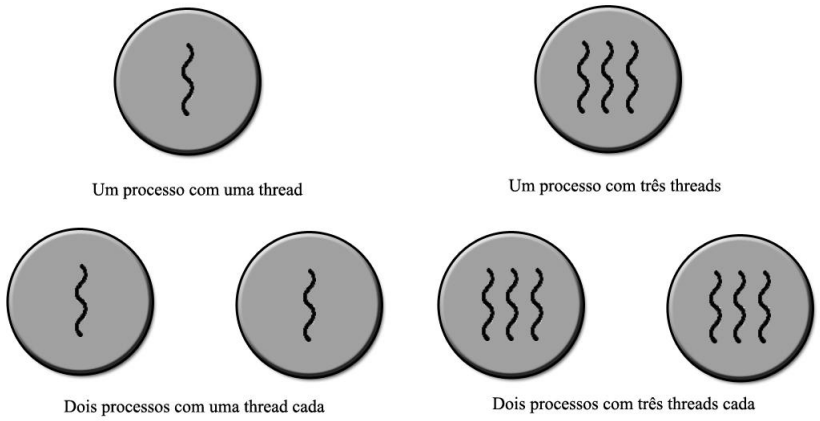
Threads é uma forma de um processo dividir a si mesmo em duas ou mais tarefas que podem ser executadas concorrentemente. Uma *thread* possui todas as características de um processo, menos o espaço de endereço privado. Eles também compartilham o acesso à memória, e dessa forma, podem se comunicar com outras threads lendo ou escrevendo diretamente na memória. Esta diferença entre *threads* e processos está ilustrada na figura 14.

A programação com *threads* é conhecida como programação paralela de memória compartilhada. Em contra partida, a programação paralela com o uso de processos é geralmente referida como programação paralela por passagem de mensagens (LIN; SNYDER, 2008).

2.3.1.2 Barreira

Um grupo de *threads* ou processos é iniciado igualmente em um ponto do código-fonte, porém isso não garante que todos irão alcançar o fim do código de modo igual. Isso acontece devido à alguns fatores que podem ser: a carga de trabalho dos núcleos, os quais uns podem estar mais sobrecarregados com processos de programas ou do próprio sistema

Figura 14: Ilustração demonstrando a diferença entre processos e *threads*.



Fonte (ROSARIO, 2012)

operacional que outros; a carga de trabalho repartida entre as *threads* ou processos; ou o próprio algoritmo de escalonamento que pode ocasionar mais vantagens a uns indivíduos. Contudo, muitas vezes é necessário garantir que alguns valores de dados não sejam lidos antes de serem computados por outras *threads* ou processos. Por isso, necessita-se de um mecanismo de sincronização entre eles.

Uma barreira é um tipo de método sincronização para um grupo de *threads* ou processos. Qualquer *thread*/processo deve parar em algum ponto específico do código e não pode prosseguir até que todos os outros alcancem esse ponto. É utilizada em aplicações onde todas as *threads* devem completar um estágio antes de irem todas juntas para a próxima fase.

O uso de barreiras gera um tempo ocioso entre as *treads*/processos, pois as primeiras têm que esperar as outras alcançarem o ponto de parada (barreira), antes de continuar. Se por exemplo, núcleos distintos têm diferentes cargas de trabalho, alguns deles podem ficar ociosos durante pelo menos uma parte do tempo em que os outros núcleos estão trabalhando no problema paralelo. Isso ocorre nos casos em que núcleos têm que sincronizar a execução de tarefas no código. Se por acaso algum dos núcleos ainda não tiver terminado sua respectiva parte da tarefa, aqueles que terminarem antes, deverão esperar em um estado ocioso pelo núcleo atrasado ou mais atarefado. Isso é um caso de *Overhead*, que será abordado logo mais adiante.

2.3.1.3 Região Crítica

Sempre que dois ou mais processos/*threads* estão acessando um recurso compartilhado na memória, por exemplo uma variável, é possível que um processo interfira em outro. Isso se chama **condição de corrida**.

Por exemplo, suponha-se que duas *threads* estão tentando incrementar a mesma variável x localizada na memória. Ambas têm a seguinte linha no código-fonte:

```
x := x + 1.
```

Uma maneira para cada *thread* executar esta declaração é ler a variável, e em seguida, adicionar um valor unitário, e depois, escrevê-lo de volta. Suponha-se que o valor de `x` é 3. Se ambas as *threads* lerem `x`, ao mesmo tempo, então receberão o mesmo valor 3 dentro do escopo de memória de cada um. Se for adicionado o valor 1 à variável `x`, as *threads* terão o valor 4, e tão logo terão que escrever 4 de volta na variável compartilhada `x`. O resultado é que ambas as *threads* incrementam `x`, mas seu valor é 4, ao invés de 5.

Para as *threads* executarem corretamente, deve-se assegurar que apenas uma delas execute a instrução de cada vez. Um conjunto de instruções que pode ter apenas um processo/*thread* e executá-lo em um momento é chamado de **seção crítica**.

Existem algumas soluções para esse problema, as duas mais básicas são: a possibilidade de independência do acesso de diversas *threads* a uma variável, isto é, se é possível substituir uma variável compartilhada por uma variável local e independente para cada *thread*. A outra é o uso de semáforos, ou seja, para entrar em uma seção crítica, uma *thread* deve obter um semáforo, o qual é liberado quando esta sai da seção. As outras *threads* são impedidas de entrar, enquanto o semáforo não for liberado. Contudo, são livres para ganhar o controle do processador e executar outro código, incluindo outras seções críticas que são protegidas por diferentes semáforos.

Embora o semáforo seja uma forma eficiente de garantir a exclusão mútua, seu uso pode reduzir severamente o desempenho, principalmente se a verificação de disponibilidade de acesso estiver dentro de um laço realizada com certa frequência. A independência de acesso entre *threads* nem sempre pode ser implementada, mas quando é possível acabará sendo mais eficiente.

2.3.1.4 OverHead

A melhoria de desempenho de um programa computacional é o principal objetivo da computação paralela. Porém, existem diversos problemas que digridem a potencialidade da computação paralela e infelizmente, esses problemas são inerentes da mesma. Idealmente, um programa que leva um tempo T para ser executado em um processador *single core*, deveria ser executado em um tempo $^{T}/_{P}$, sendo P o número de processadores.

Quando se utiliza o dobro de recursos de hardware, muitos podem esperar que um programa seja executado duas vezes mais rápido. No entanto, este raramente é o caso, devido a uma variedade de *overheads* gerais associados ao paralelismo (GRAMA et al., 2003).

Os *overheads* são provenientes da interação entre processos, *threads*, ociosidade e

excesso de computação. Eles são uma combinação direta ou indireta de tempo excessivo do uso de computação, de memória, de largura de banda, ou de outros recursos durante algum objetivo computacional. Ou seja, *overhead* é a quantidade de tempo requerido para coordenar as tarefas em paralelo, ao invés de fazer o trabalho útil. *Overhead* paralelo pode incluir fatores como:

- Tempo de criação de um processo ou *thread*;
- Sincronizações;
- Comunicações de Dados;
- *Overhead* de Software imposta por línguas paralelas, bibliotecas, sistema operacional, etc;
- Tempo de destruição de um processo ou *thread*.

2.3.2 Métricas de Escalabilidade paralela

O algoritmo paralelo reduz o tempo de execução sequencial, porém é importante estudar o desempenho dos programas paralelos, entendendo até que ponto é vantajoso utilizá-los. Esse tipo de análise tem o objetivo de determinar os benefícios do paralelismo aplicados a um problema considerado, e, também, o quanto o algoritmo é capaz de continuar eficiente.

Antes de discutir as métricas, é importante perceber que todo algoritmo possui uma fração seqüencial e paralela. A porção seqüencial é a parte do código a qual não pode ser executado concorrentemente e limita a aceleração dos algoritmos paralelos. Dessa forma, um algoritmo sequencial ótimo, ou seja, o seu tempo de execução é mínimo, provavelmente possua uma fração seqüencial muito alta, portanto, não tenha perfil de algoritmo paralelo. Isto é, não tendo um bom desempenho paralelo ou até mesmo difícil ou impossível de paralelizar. Enquanto que muitos códigos sequenciais ditos como lentos ou com menor desempenho possam ser mais atrativos, possuindo uma fração paralela muito maior.

Portanto, uma série de análises deve ser realizada para que sejam efetuados ajustes para ampliar a parte paralela ou compreender se um determinado algoritmo é escalável ou não. A discussão sobre as principais métricas de algoritmos paralelos é descrita na próxima subseção.

2.3.2.1 Speedup e Eficiência

Para a análise de escalabilidade, *speedup* e eficiência, são consideradas as principais métricas. O ***speedup*** S, é definido como a razão entre o tempo sequencial [10], T_S e o tempo

[10] Tempo serial é o tempo de execução do algoritmo implementado sequencialmente.

paralelo T_P.

$$S = \frac{T_S}{T_P} \tag{2.34}$$

O *speedup* diz quantas vezes o algoritmo paralelo é mais rápido que o algoritmo serial. Idealmente, dividindo o trabalho igualmente entre os núcleos, sem acréscimo de trabalho inicial, obtém-se no programa paralelo um tempo P vezes mais rápido que o programa serial. Ou seja, $S = P$, logo, $t_P = \frac{T_S}{P}$, quando isso ocorre, chama-se de *speedup* linear. Quando $S < P$, diz que o *speedup* é sublinear, e $S > P$, o *speedup* é super linear. A super linearidade não é algo comum, porém pode ocorrer (LIN; SNYDER, 2008).

Na prática, para obter um *speedup* linear existe várias complicações, porque como foi descrito na seção anterior, o uso de vários processos/*threads* introduz *overhead*. Os programas de memória compartilhada quase sempre terão seções críticas, o que exigirá o uso de algum mecanismo de exclusão mútua, como um semáforo. As chamadas para as funções do semáforo possuem um *overhead* que não estava presente no programa serial, e o seu uso força o programa paralelo a serializar a execução da seção crítica. Os programas paralelos de memória distribuída quase sempre necessitam transmitir dados através da rede, que normalmente é muito mais lento do que o acesso à memória local. Programas sequenciais, por outro lado, não tem esses custos. Assim, será muito incomum que programas paralelos tenham *speedup* linear ou super linear. Além disso, é provável que o *overhead* sofra acréscimos à medida que aumentar o número de *threads*, ou seja, mais *threads* provavelmente vai significar mais acesso a seção crítica. E, mais processos, significará provavelmente mais dados a serem transmitidos através da rede.

Muitas vezes, *speedups* reais tendem a saturar com o aumento do número de processadores. Isso ocorre porque com o tamanho do problema fixo e com o aumento de número de processadores, há uma redução da quantidade de trabalho por processador. Com menos trabalho por processador, custos como *overhead* podem se tornar mais significativos, de modo que a relação entre o tempo serial e o paralelo não melhora de forma linear (GRAMA et al., 2003). Em outras palavras, significa que S/P provavelmente vai ficar cada vez menor, a medida que P aumenta. Este valor, S/P, é chamado de **eficiência** do programa paralelo. Se substituirmos a fórmula de S, a eficiência é

$$E = \frac{S}{P} = \frac{\frac{T_S}{T_P}}{P} = \frac{T_S}{PT_P} \tag{2.35}$$

A eficiência é uma medida normalizada de *speedup* que indica o quão efetivamente cada processador é utilizado. Um *speedup* com valor igual a P, tem uma eficiência igual 1, ou seja, todos processadores são utilizados e a eficiência é linear. E, valores acima de 1 é chamado de eficiência super linear. No entanto, como dito anteriormente, existem vários fatores que degradam o desempenho de um programa paralelo, logo a eficiência

será tipicamente inferior a 1, e sempre diminui à medida que o número de processadores aumenta.

Está claro que o *speedup* e a eficiência dependem do número de P, número de *threads* ou processos. Precisa-se ter em mente que o tamanho do problema também tem influência nessas métricas. Por exemplo, o autor Pacheco (2011, pg. 123) em seu livro intitulado *An Introduction Parallel Programming*, cita dados de um experimento realizado, utilizando MPI para paralelizar uma função sequencial que multiplica uma matriz quadrada por um vetor. Os testes foram realizados utilizando uma matriz de ordem 1024, 2048, 4096, 8192, 16384 e com 2, 4, 8, 16 processos. A tabela 3 mostra o tempo sequencial e paralelo em milissegundos.

Tabela 3: Tempos de execução serial e paralelo da multiplicação de matriz por um vetor.

Processos	**Ordem da Matriz**				
	1024	2048	4096	8192	16384
1	4.1 ms	16.0 ms	64.0 ms	270 ms	1100 ms
2	2.3 ms	8.5 ms	33.0 ms	140 ms	560 ms
4	2.0 ms	5.1 ms	18.0 ms	70 ms	280 ms
8	1.7 ms	3.3 ms	9.8 ms	36 ms	140 ms
16	1.7 ms	2.6 ms	5.9 ms	19 ms	71 ms

Adaptado da fonte: (PACHECO, 2011)

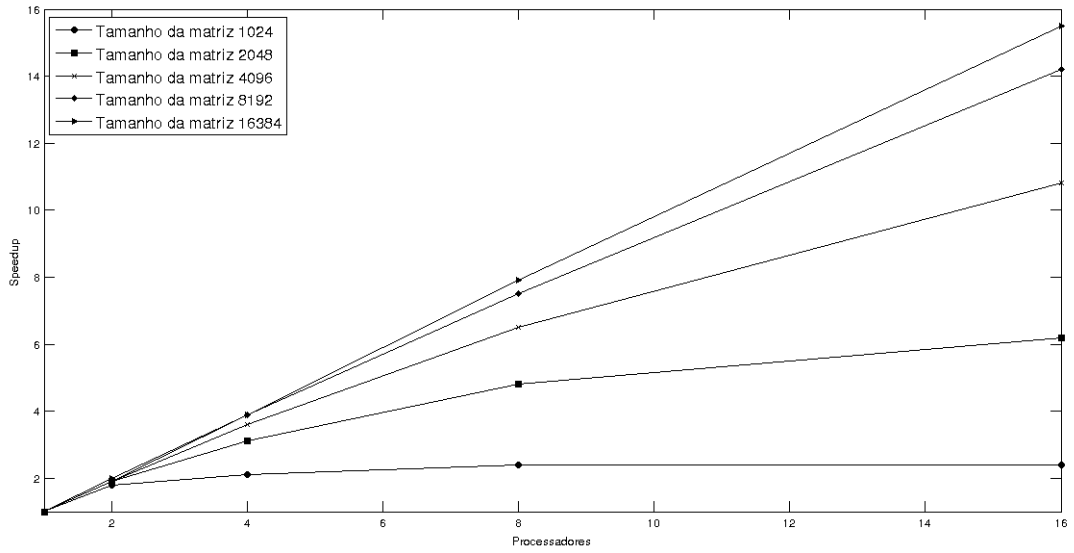
As figuras 15 e 16 mostram o *speedup* e eficiência, respectivamente, do programa paralelo da multiplicação de matriz por um vetor para diferentes tamanhos de problema e quantidades de processos. Como pode ser visto nas figuras, quando aumenta-se o tamanho do problema, o *speedup* e a eficiência também aumentam. Percebe-se, também, que o comportamento dos gráficos apresentados nas figuras 15 e 16 são muito comuns em programas paralelos. O *speedup* cresce com o aumento da quantidade de processos/*threads*, contudo tende a saturar após certo ponto. A eficiência tende a diminuir com o aumento da quantidade de processos/*threads*, isso ocorre devido a própria definição de eficiência mostrada na equação 2.35 ($E = S/P$).

Muitos programas paralelos são desenvolvidos para dividir o programa sequencial entre as *threads*/processos, contudo, deve ser lembrado o tempo de *overhead* paralelo, como exclusão mútua ou comunicação. Então, T_O, denotando o tempo de *overhead*, tem-se que:

$$T_P = T_S/P + T_O. \tag{2.36}$$

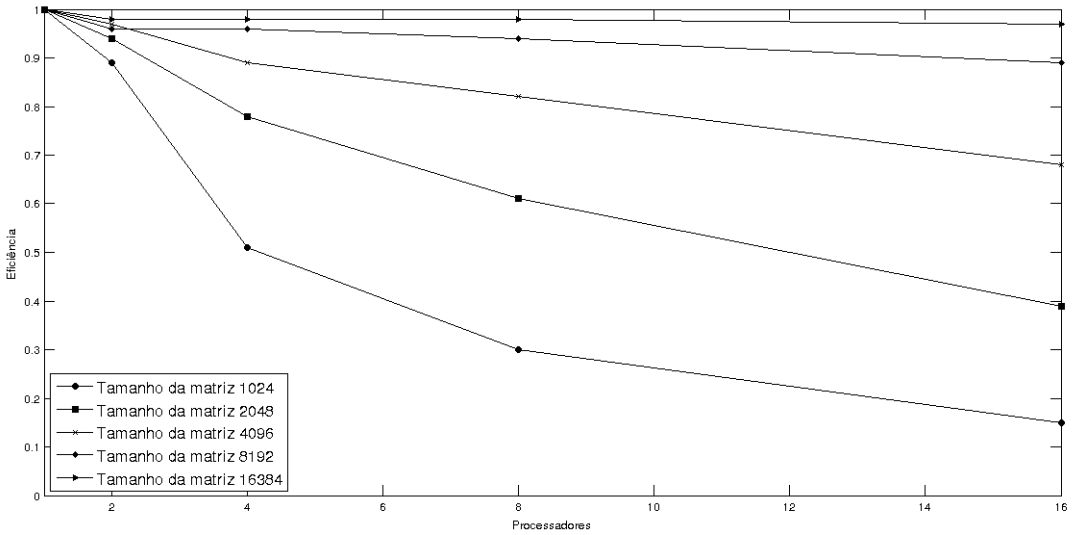
Além disso, quando o tamanho do problema é aumentado, o T_O frequentemente cresce com uma taxa bem menor que o tempo sequencial. Quando isso acontece o *speedup*

Figura 15: *Speedup* do programa paralelo de diferentes tamanhos do problema da multiplicação de matriz por um vetor.



Adaptado da fonte:(PACHECO, 2011)

Figura 16: Eficiência do programa paralelo de diferentes tamanhos do problema da multiplicação de matriz por um vetor.



Adaptado da fonte:(PACHECO, 2011)

e a eficiência também aumentam, devido ao aumento da quantidade de trabalho por processador/núcleo.

Para ter melhor entendimento sobre escalabilidade, será detalhado a lei de Amdahl, a qual, mesmo tendo sido elaborada antes da *era multicore*, ainda possui grande influência na área de paralelismo.

2.3.3 Lei de Amdahl

Na década de 1960, Gene Amdahl, fez uma observação que se tornou conhecida como a lei de Amdahl. Ela diz, resumidamente que, a menos que praticamente todo um programa serial seja paralelizado, o possível aumento do *speedup* vai ser muito limitado, independentemente do número de núcleos disponíveis. A consequência disso é que haverá um limite onde o processamento paralelo pode ser aplicado, mesmo utilizando uma quantidade enorme de processadores/núcleo.

O tempo de execução de um problema paralelo ideal, sem *overhead*, é definido como a soma do tempo da porção paralela do código com o tempo da porção sequencial, assim como mostra a equação abaixo:

$$T_P = (1 - P)T_S \ + \ P\left(\frac{T_S}{N}\right) \tag{2.37}$$

onde, T_P é o tempo paralelo, T_S é o tempo sequencial, P é a fração paralela do código, $(1 - P)$ é fração sequencial e N é o número de processadores.

Ao manipular a equação 2.37 e sabendo da fórmula de *speedup* apresentada em 2.34, tem-se que:

$$T_P = T_S\left[(1 - P) \ + \ \frac{P}{N}\right] \Rightarrow S = \frac{T_S}{T_P} = \frac{1}{(1 - P) \ + \ \frac{P}{N}} \tag{2.38}$$

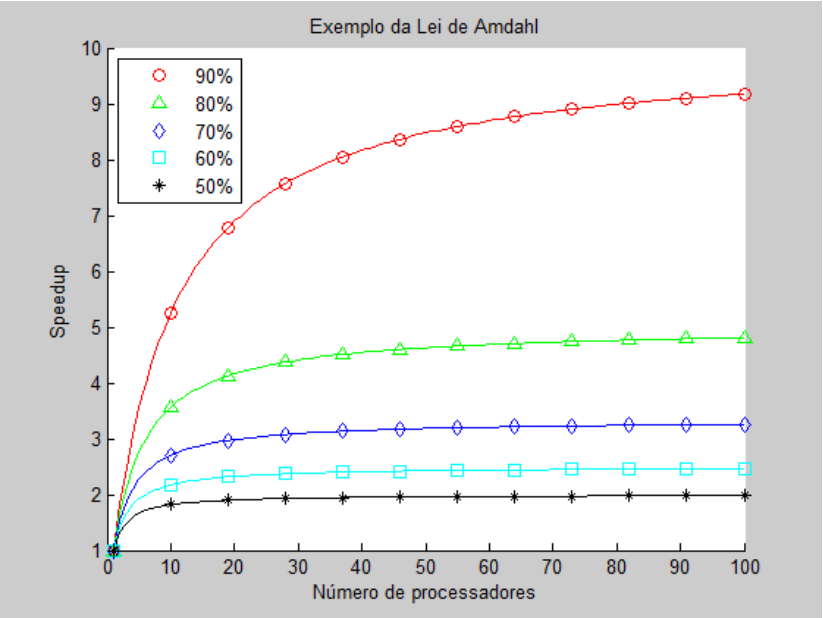
A equação 2.38 diz que, ao aumentar o número de processadores/núcleos a uma quantidade exorbitante, ou seja, $N \to \infty$, a parte paralelizável será reduzida a zero e o *speedup* será:

$$S = \frac{1}{r} \tag{2.39}$$

onde, $r = (1 - P)$, ou seja, a porção serial do código. Logo, a lei de Amdahl diz, que se uma fração r do código do programa sequencial permanece não paralelizável, então o *speedup* nunca será melhor do que $1/r$. Por exemplo, $r = 1 - 0,9 = 1/10$, portanto, o *speedup* não poderá ser melhor do que 10. Portanto, se um r é inerentemente sequencial, isto é, não pode ser paralelizado, então não se pode obter um aumento de velocidade melhor que $1/r$. Assim, mesmo se r é muito pequena, por exemplo, $1/100$ e um sistema com milhares de

núcleos, o *speedup* não será melhor do que 100. O exemplo exibido na figura 17 ilustra a limitação superior aplicado ao *speedup*, em que não importa a adição de processadores e o tamanho da fração paralela, a execução paralela não será melhorada.

Figura 17: Exemplo da lei de Amdahl.



fonte:(ROSARIO, 2012).

A interpretação dessa lei como um argumento de que a melhora do desempenho por paralelismo é limitada, não leva em consideração o tamanho do problema. Para muitos casos, aumentar o tamanho do problema faz com que a fração sequencial do programa diminua. Portanto, a duplicação do tamanho do problema pode ampliar a porção sequencial para um tamanho irrisório, fazendo com que a maior fração do problema seja necessariamente paralela. Outro ponto importante é que existem milhares de programas usados por cientistas e engenheiros que rotineiramente obtém enormes aumentos de velocidade em sistemas de memória distribuída de grande porte (PACHECO, 2011).

2.3.4 Escalabilidade

A palavra escalável tem uma grande variedade de significados em diversas áreas. Essa palavra já foi usada ao longo do texto diversas vezes. Informalmente, a tecnologia é escalável quando ela pode lidar com problemas cada vez maiores sem perdas de desempenho. No entanto, uma definição mais formal é feita por Pacheco (2011), o qual diz que, se puder manter o valor da eficiência constante, aumentando o tamanho do problema com a mesma taxa de crescimento do número de processos/*threads*, então o programa é dito **escalável**.

Será detalhado a seguir os dois tipos de escalabilidades existentes: Escalabilidade de Amdahl e Escalabilidade de Gustafson.

2.3.4.1 Escalabilidade de Amdahl

Suponha um programa paralelo com tamanho do problema e a quantidade de processos/*threads* fixos, assim, obtêm-se uma eficiência E. Admita agora, que aumenta-se a quantidade de processos/*threads* usados pelo programa. Se a eficiência E manter-se constante com esse aumento e com tamanho do problema fixo, então esse programa é dito escalável por Amdahl (PACHECO, 2011).

Percebe-se que para a eficiência, $E = S/P$, manter-se constante para diferentes valores de P, é necessário que o *speedup* S cresça na mesma proporção que P com o tamanho do problema fixo. Para Amdahl, quanto maior for o número de unidades de processamento, para um problema de tamanho fixo, maior será o *speedup*, já que T_P será menor. Todavia, como visto anteriormente, o *speedup* possui um valor máximo devido à fração serial do código.

Segundo a Lei de Amdahl, a velocidade de processamento paralelo é limitada. Ela afirma que, uma pequena porção do programa que não pode ser paralelizada limitará o aumento de velocidade disponível com o paralelismo. A consequência é que haverá um limite onde o processamento paralelo pode ser aplicado. Os algoritmos escaláveis de Amdahl tem o comportamento ilustrado pela figura 17, isto é, há uma melhoria significativa do ganho de paralelização de acordo com o número de unidades de processamento. E, isso depende, da porcentagem da porção paralela que o código possui, quanto maior é essa porção, melhor é o desempenho paralelo, melhor é o *speedup*. Nota-se também que, quanto menor for a porção sequencial do código, menos limitado será o *speedup*, podendo alcançar valores maiores.

2.3.4.2 Escalabilidade de Gustafson

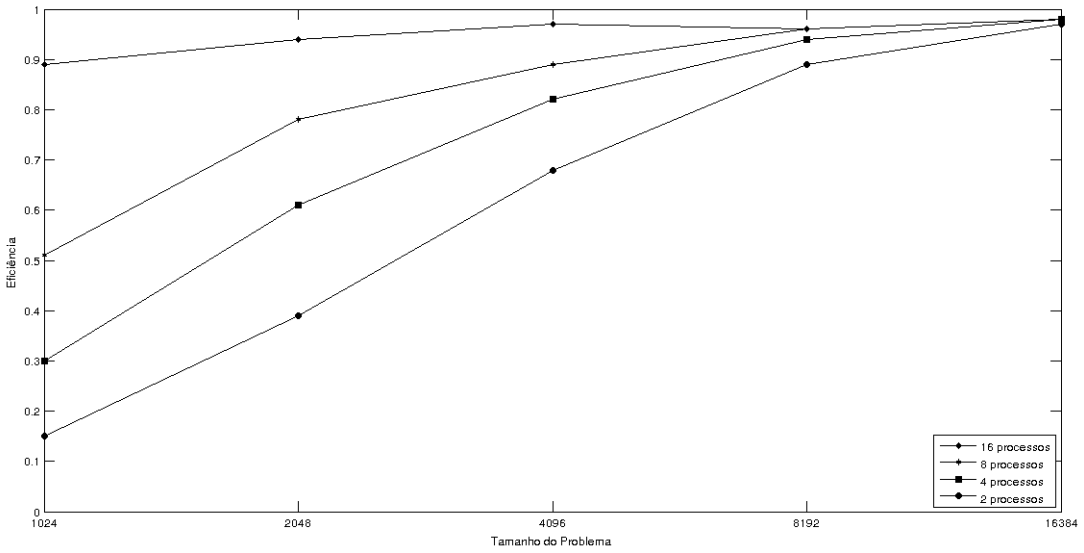
Suponha, agora, um programa paralelo com tamanho do problema e a quantidade de processos/*threads* fixos, assim, obtêm-se uma eficiência E. Admita também, que aumenta-se tanto a quantidade de processos/*threads* usados pelo programa como o tamanho do problema. Se a eficiência E manter-se constante nesse cenário, então esse programa é dito escalável por Gustafson (PACHECO, 2011).

A escalabilidade de Amdahl, não leva em conta o tamanho do problema, como foi explicado anteriormente. Gustafson percebeu isso em seu trabalho *Reevaluating Amdahl's law* (GUSTAFSON, 1988). Comparando experimentos, notou que o *speedup* não estava limitado pela parte serial do algoritmo, na realidade, o paralelismo cresceu de acordo com o tamanho do problema.

Para comprovar a escalabilidade Gustafson de um algoritmo paralelo, faz-se necessário aumentar o tamanho do problema e verificar o aumento da eficiência. Pois, a eficiência de um programa paralelo está intimamente conectada com a porção paralela do

código. Um exemplo de um algoritmo escalável por Gustafson é apresentando na figura 18.

Figura 18: Eficiência do programa paralelo de diferentes tamanhos do problema da multiplicação de matriz por um vetor.



Adaptado da fonte: (PACHECO, 2011).

3 Revisão Bibliográfica

Esse capítulo apresenta a Revisão Bibliográfica sobre a paralelização do método Simplex com informações obtidas a partir da literatura científica. Ele está dividido em 3 seções: a primeira, apresenta os artigos realizados antes da arquitetura *multicore*, usando memória compartilhada; a segunda seção apresenta artigos mais atuais, utilizando memória compartilhada e, por último, uma breve explicação do que é software de otimização e cita alguns exemplos destes.

3.1 Memória Distribuída

Como já descrito na seção 2.1, o método Simplex trata-se de um procedimento utilizado para resolver problemas de Programação Linear, criado em 1947 por Dantzig. Desde então, sofreu diversas modificações e tem sido usado amplamente por programadores e profissionais de diferentes setores da indústria, assim como, nas áreas da saúde, logística e transporte, estatística e gestão financeira.

A aplicação do processamento paralelo com o método Simplex para a programação linear tem sido considerado desde o início da década de 70 e se intensificando ainda mais na década de 90 (PLOSKAS; SAMARAS; SIFALERAS, 2009). Como justificativa para tal comportamento tem-se rápido crescimento do poder computacional, principalmente do desenvolvimento das máquinas paralelas.

Eckstein et al. (1995) descrevem 3 implementações paralelas do Simplex para problemas densos de PL, aplicados a um computador SIMD. A primeira usa o Simplex *Tableau* com a regra do menor custo do valor negativo dos coeficientes da função objetivo para a escolha do pivô; a segunda utiliza o algoritmo *steepest-edge*[1] para escolha do pivô, também para o *Tableau*; e por último, a paralelização do método revisado. Todas as implementações são realizadas na Máquina de Conexão CM-2(*Connection Machine* CM-2). O trabalho descreve a comparação entre essas implementações e também utiliza um software de otimização chamado MINOS versão 5.4 em uma máquina sequencial produzida pela Sun Microsystems chamada Sun SPARCStation 2. A máquina SPARCStation é da mesma época que foi desenvolvido o CM-2 e possui o mesmo poder de processamento de uma única unidade vetorial de processamento da máquina paralela. Os problemas de teste são retirados do Netlib[2], e também são adicionados alguns problemas densos de aplicações reais.

[1] A regra de Dantzig normalizada, ou *steepest-edge*, consiste em escolher uma variável que tenha o coeficiente normalizado mais negativo. Para escolher qual variável irá entrar na base deve-se primeiro normalizar os coeficientes, e depois, escolher a mais negativa. Dessa forma, é possível encontrar em menos iterações a solução ótima.

[2] O repositório Netlib contém software disponível gratuitamente, documentos e bases de dados de

O *tableau* paralelo com a regra *Steepest-edge* obteve o melhor resultado, conseguiu ser 20 vezes mais rápido que as duas outras implementações. O simplex *tableau* paralelo de menor custo foi mais eficiente que o algoritmo Simplex do MINOS. Já o algoritmo revisado paralelo comparado com o *tableau* paralelo sem o *steepest-edge* foi cerca de 1.5 vezes mais rápido. Esse trabalho mostrou que o CM-2 pode render tempos de execução paralela melhor do que uma estação de trabalho que executa o mesmo algoritmo equivalente sequencialmente. Com os resultados, percebe-se que independentemente do algoritmo, problemas difíceis, de grandes escalas e densos, são mais viáveis quando resolvidos paralelamente, pois estes apresentam melhor resultado que seu algoritmo sequencial.

Outro trabalho envolvendo computadores SIMD, mais potentes quando comparados ao CM-2, foi desenvolvido por Thomadakis e Liu (1996). Eles propuseram um algoritmo do simplex paralelo escalável baseado no *tableau* e na regra *steepest-edge* para seleção do pivô. O algoritmo foi desenvolvido, utilizando todas as especificidades das máquinas massivas paralelas MasPar 1 (MP-1) e MasPar-2(MP-2). Uma versão sequencial do simplex *steepest-edge* foi implementada no servidor Unix Sun SPARC 1000 para efeitos de comparação. Em termos de desempenho, os experimentos mostraram que o algoritmo paralelo do simplex proposto executado no MP-2, com 128 x 128 elementos de processamento, alcançou um *speedup* de ordem 1.000 para 1.200 vezes maior que a versão sequencial. Enquanto que, no MP-1, com 64 x 64 elementos de processamento, teve ordem de 100 vezes mais rápido. Este trabalho também analisou a escalabilidade fixando a quantidade de processadores e aumentando o tamanho de linhas ou colunas. Com sucesso, os autores conseguem demonstrar que à medida que aumenta o tamanho do problema o *speedup* cresce, demonstrando a escalabilidade da implementação.

O trabalho desenvolvido por Hall e McKinnon (1998), utiliza um computador MIMD[3]. Hall e McKinnon descrevem um variante assíncrono do método Simplex revisado, o qual é adequado para implementação paralela em um multiprocessador com memória compartilhada ou, em um computador MIMD com uma rápida comunicação entre os processadores. A implementação deste algoritmo é realizada na Cray T3D, uma máquina que possui uma grande taxa de comunicação comparada com a computação. Isso é necessário para alcançar um bom *speedup* quando esse método assíncrono é usado em memória distribuída. Um dos objetivos dessa implementação é permitir que processos operem de uma maneira assíncrona, ou seja, o máximo de independência possível e um resultado dessa execução é que diferentes processos podem trabalhar com diferentes bases ao mesmo tempo. Neste trabalho os experimentos computacionais no Cray T3D foram realizados utilizando quatro problemas do Netlib. O método Simplex sequencial com a regra

interesse para a computação científica, numérica e outras comunidades. O repositório é mantido pela AT & T Bell Laboratories, da Universidade do Tennessee e Oak Ridge National Laboratory.

[3] Relembrando o capítulo 2.2, MIMD Refere-se aos sistemas paralelos de natureza assíncrona, ou seja, as instruções são geradas por unidades de controle diferentes e cada processador executa sua instrução no seu próprio dado.

steepest-edge para escolha do pivô foi implementado em um computador Sun SPARCstation 5, utilizando o ERGOL[4] versão 1.2. Os resultados demonstraram um bom *speedup* entre 2.5 e 4.8 para os quatro problemas. Segundo os autores, é possível implementar esta variante em processadores de memória compartilhada com certa facilidade. Contudo, a performance não pode ser garantida a menos que a memória *cache* seja capaz de armazenar os dados que são acessados a cada iteração.

As máquinas paralelas foram produzidas a partir da década de 70 e eram caracterizadas pelo custo elevado e pela dificuldade de programação. O programador necessitava ter o conhecimento específico de cada máquina paralela para a qual seriam implementadas as aplicações paralelas, o que aumentava a complexidade do desenvolvimento do programa. Além disso, desde meados de 2000, vem acontecendo uma mudança de paradigma, onde os computadores não estão sendo produzidos com apenas um núcleo de processamento. Esta tendência tem sido chamada de *Era Multicore*, detalhado por Kock e Borkar (KOCH, 2005; BORKAR, 2007) e explicado na seção 2.3.

3.2 Memória Compartilhada

Muitas das pesquisas desenvolvidas tiveram como foco o método revisado do simplex, devido a vantagem de resolver problemas esparsos[5] e, sendo, mais eficiente para problemas em que a quantidade de variáveis é bem maior que a quantidade de restrições (YARMISH; SLYKE, 2003). Contudo, o método revisado não é muito adequado para a paralelização (HALL, 2010).

Em 2009, Yarmish e Slyke (2009) apresentaram uma implementação distribuída do Simplex padrão para problemas de grande escala de programação linear. Nos experimentos foi utilizado um laboratório com 7 estações de trabalho independentes conectados pela Ethernet. Para comparar com a aplicação paralela proposta, foi utilizado o método simplex revisado do pacote de otimização MINOS. Um conjunto de testes foi elaborado com 10 problemas de 1000 variáveis e 5000 restrições variando a densidade da matriz de restrições de 5% até 100% e foi constatado que o Simplex padrão paralelo proposto não é afetado pela variação da densidade do problema. Além do mais, quando usa-se 7 processos é comparável com o método revisado sequencial com 10% de densidade, pois quanto menor a densidade, mais eficiente será o método revisado. Assim, pelo modelo matemático desenvolvido neste trabalho, se houvesse a possibilidade da utilização de 17 processos, o algoritmo paralelo proposto seria equivalente ao método revisado sequencial com menos de 10% de densidade. Logo, é possível construir uma versão eficiente paralela do método padrão, considerando que o método revisado é difícil de paralelizar.

[4] ERGOL é um software de otimização da IBM

[5] Um problema é dito esparso, quando a maioria dos elementos de sua matriz de restrições são nulos.

Um outro trabalho produzido no mesmo ano, desta vez, por Ploskas, Samaras e Sifaleras (2009) mostraram a implementação paralela do algoritmo do Simplex, usando um computador pessoal de dois núcleos. Nesta abordagem, a base inversa é calculada em paralelo, a matriz que contém a base é distribuída entre os processos e o cálculo é realizado mais rapidamente em problemas de grande escala de PL. Além da implementação paralela, este trabalho apresenta um estudo computacional que mostra o *speedup* entre a versão serial e o paralelo em problemas PL densos gerados aleatoriamente. A comparação computacional foi realizada em um computador com processador Intel Core 2 Duo T5550, 2 núcleos, 2MB de cache, 1.83 GHz, com 3Gb RAM executando a edição do Windows Professional XP 32-bit SP3. O algoritmo foi implementado usando a edição MATLAB R2009a 32-bit Professional. Foram criados 6 problemas de tamanhos diferentes, para cada problema, 10 instâncias foram geradas, utilizando um número de semente distinta. Todas as instâncias foram geradas aleatoriamente, executando 5 vezes cada instância e todas as ocorrências foram em média 98% densas. O melhor *speedup* adquirido com a paralelização da base inversa foi em torno de 4,72. Devido às matrizes muito densas e muita comunicação, a relação de computação para a comunicação foi extremamente baixa e como resultado a implementação paralela não ofereceu *speedup* para o tempo total.

Nota-se que todos os trabalhos relacionados a esse tema apresentam diferentes formas de paralelizar o Simplex, buscando a melhor eficiência possível. Porém, é necessário que os algoritmos paralelos possam ir além da melhor eficiência, pois devido a *Era Multicore* tem-se a cada ano um aumento exponencial do número de processadores (BORKAR, 2007). Logo, é importante que tais algoritmos acompanhem esse crescimento sem que haja repetidas mudanças no código, ou seja, analisar a escalabilidade com diferentes números de núcleos, variando a dimensão do problema.

3.3 Softwares de Otimização

Um software de otimização é software matemático na área de otimização sob a forma de um programa *desktop* ou fornecendo uma biblioteca contendo as rotinas necessárias para resolver um problema matemático. Alguns tipos de problemas matemáticos que podem ser resolvidos por esses softwares são: programação linear e não-linear, programação inteira, programação quadrática e outros.

Existem vários softwares de otimização, os quais variam: o tipo de licença, proprietário ou código livre, as linguagens de programação para qual os desenvolvedores podem incorporar em seus próprios programas, e os tipos de problemas matemáticos que podem ser resolvidos. Nessa sessão serão apresentados dois softwares, o CPLEX® da IBM (utilizado nesse trabalho) e o COIN-OR(*Computational Infrastructure for Operations Research*), um repositório para projetos de código livre na área de softwares matemáticos.

3.3.1 IBM ILOG CPLEX Optimization

IBM® ILOG® CPLEX® Optimization é uma ferramenta da IBM para resolver problemas de otimização. Cplex foi escrito por CPLEX Optimization Inc., que foi adquirida pela ILOG em 1997 e a ILOG foi posteriormente adquirida pela IBM em janeiro de 2009. Atualmente, continua sendo desenvolvido pela IBM e reconhecido como uma das melhores ferramentas para otimização, conquistando o INFORMS Impact Prize em 2004 (LOWE, 2013).

CPLEX® oferece bibliotecas em C, C++, Java, .Net e Python. Especificamente, resolve problemas de optimização linear ou quadrática, ou seja, quando o objetivo a ser otimizado pode ser expresso como uma função linear ou uma função quadrática convexa. O CPLEX® Optimizer é acessível através de sistemas de modelagem independentes, como AIMMS, AMPL, GAMS, AMPL, OpenOpt, OptimJ e Tomlab.

O IBM® ILOG® CPLEX® consiste nos seguintes otimizadores: otimizador CPLEX® para programação matemática; otimizador IBM® ILOG® CPLEX® CP para a programação com restrições; *Optimization Programming Language* (OPL) (Otimização de Linguagem de Programação), o qual fornece uma descrição matemática natural dos modelos de otimização e uma IDE totalmente integrado (CPLEX..., 2013).

O IBM® ILOG® CPLEX® Optimizer resolve problemas de programação quadrática (PQ) convexa e não-convexa, de programação inteira e problemas de programação linear (PL) de grande escala, utilizando tanto as variantes primal ou dual do método Simplex, ou o método de ponto interior (*barrier*).

O CPLEX® pode utilizar de multiprocessamento para melhorar o desempenho. O otimizador concorrente (concurrent optimizer) dessa ferramenta lança diferentes PL e PQ otimizadores em várias *threads*. Se disparar somente uma *thread*, o CPLEX® funciona de forma automática, da mesma forma que sequencialmente, ou seja, escolhe qual otimizador será utilizado dependendo do tipo do problema. Se uma segunda *thread* estiver disponível, o otimizador concorrente executa o otimizador *barrier*. Se uma terceira ocorrer, todas os três otimizadores serão executados ao mesmo tempo: dual Simplex, primal Simplex e o *barrier*. Acima de três *threads* todas elas são dedicadas à paralelização do método *barrier*.

Percebe-se que o CPLEX® utiliza um algoritmo para otimizar a solução usando vários otimizadores concomitantemente, para obtenção do melhor tempo possível. Contudo, não possui uma rotina que paralelize somente um único método, impossibilitando, assim, uma comparação mais concisa com o algoritmo paralelo proposto.

3.3.2 COIN-OR

COIN-OR significa, em português, Infra-estrutura Computacional para Pesquisa Operacional. Trata-se de um repositório dedicado a software de código aberto para a

comunidade de pesquisa operacional. Hospedado pelo INFORMS, o site COIN-OR é o comporta vinte e cinco projetos e uma comunidade crescente composta de aproximadamente 2.003 assinaturas (COIN-OR, 2007). Ele incentiva novos projetos, fornecendo um extenso conjunto de ferramentas e suporte para o desenvolvimento de projetos colaborativos e desde 2004, tem sido gerido pela fundação COIN-OR sem fim lucrativos.

O repositório COIN-OR foi lançado como um experimento em 2000, em conjunto com o 17º Simpósio Internacional de Programação Matemática em Atlanta, Geórgia. Dentre os projetos tem-se: ferramentas para programação linear, como CLP (*COIN-OR Linear Programming*); programação não-linear como COIN-OR IPOPT(*Interior Point Optimizer*); programação inteira, como CBC (COIN-OR Branch and Cut) e outros (COIN-OR..., 2013).

CLP (COIN-OR LP) é um software de código-aberto para programação linear escrito em C++. Seus principais pontos fortes são seus algoritmos Dual e Primal Simplex. Existem, também algoritmos para resolver problemas não-lineares e quadráticos. O CLP está disponível como uma biblioteca e como um programa *standalone*[6].

Existem também aplicações paralelas no COIN-OR. Algumas são: o projeto CHiPPS (*COIN-OR High Performance Parallel Search Framework*), uma estrutura para a construção de algoritmos de árvore de busca paralelos; BCB, (*Branch-Cut-Price Framework*) que é uma estrutura paralela para a implementação *Branch-Cut-Price* - método para resolver programas inteiros mistos; e PFunc (*Parallel Functions*), que é uma biblioteca leve e portátil que fornece APIs em C e C++ para expressar paralelismo em tarefas. Contudo, nenhum dos projetos ainda, não possui um Simplex padrão paralelo para realizar uma comparação com o algoritmo paralelo proposto.

[6] São chamados *standalone*, os programas completamente auto-suficientes, ou seja, para seu funcionamento não necessitam de um software auxiliar sob o qual terão de ser executados.

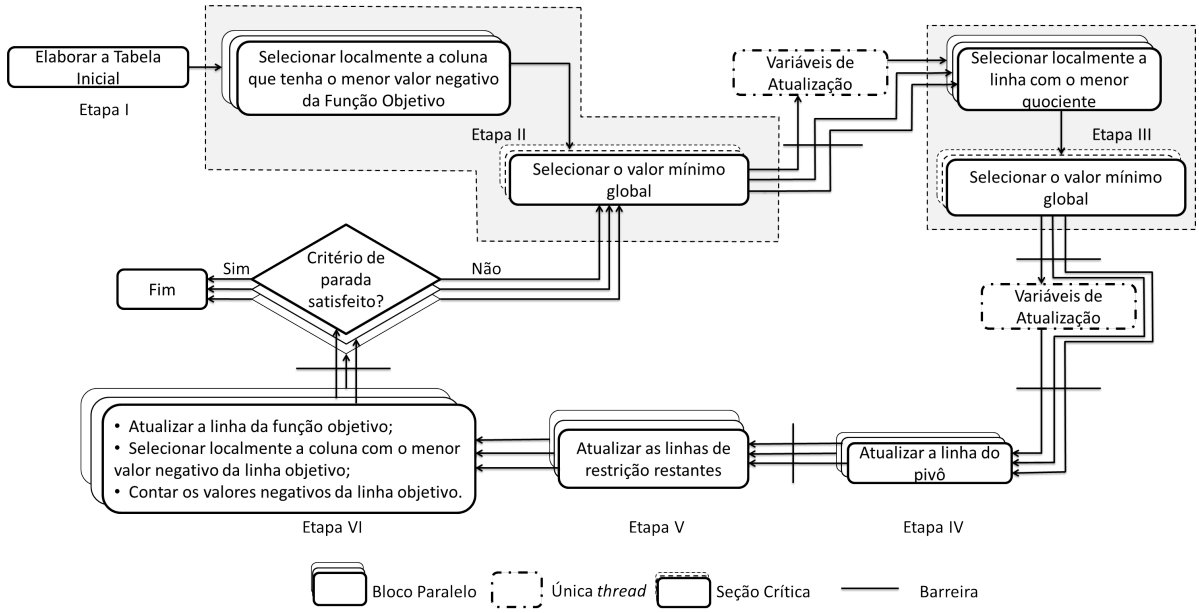
4 Algoritmo Simplex Multicore

Nesse capítulo será apresentado a metodologia utilizada, abordando inicialmente a ideia geral de como o algoritmo simplex padrão na forma tabular (*tableau*) foi paralelizado. Logo em seguida, será apresentada a Implementação do algoritmo Simplex *Multicore* em OpenMp, detalhando cada etapa. Além de mostrar a conexão entre os conceitos de paralelismo e o Simplex *tableau*.

4.1 Esquema Geral da Paralelização

O algoritmo paralelo proposto por esse trabalho é baseado no simplex *tableau* apresentado na seção 2.1.3. Esse algoritmo é composto por seis etapas como pode ser visto na figura 19.

Figura 19: Fluxograma do algoritmo Simplex Multicore.



Fonte: Autoria própria.

A primeira etapa consiste em elaborar a tabela inicial do Simplex com todas as restrições e variáveis, observando que o problema de programação linear deve estar na forma padrão. A segunda etapa deve paralelizar a seleção da coluna que tiver menor valor negativo dos coeficientes da função objetivo - teste de otimalidade. A terceira etapa deve paralelizar a seleção da linha que tiver menor valor do quociente da equação 4.1,

apresentado na seção 2.1.3 - teste da razão.

$$x_B^s = min_{y_i^k}\left\{\frac{B^{-1}b}{y_i^k}, y_i^k > 0\right\}, \forall i \in I \mid I = [1 \;\; m] \tag{4.1}$$

A quarta etapa consiste em paralelizar a multiplicação da linha pivô por $\frac{1}{y_p}$. A quinta etapa paraleliza a operação elementar em todas as linhas das restrições, exceto as linhas do pivô e da função objetivo, como demonstrado na equação 4.2, apresentado na seção 2.1.3:

$$l_i = l_i \; - \; y_p l_p, \forall i \in I \mid I = [1 \;\; m]. \tag{4.2}$$

A sexta etapa é um melhoramento que foi feito no algoritmo, na qual são paralelizadas três operações simultaneamente: a atualização da linha da f.o. a partir da equação 4.2; a realização do teste de otimalidade e a contabilização de quantos coeficientes negativos existem na linha que representa a f.o.

Percebe-se que a etapa I não é paralelizada, porque consiste em estruturar o problema inicial na forma da tabela inicial do simplex. Após ter essa tabela montada, é que são iniciados de fato os passos do simplex a serem paralelizados.

A etapa II contém um laço paralelizado, isto é, cada *thread* é responsável por selecionar localmente a coluna que tenha o menor valor negativo da linha da função objetivo a partir de um conjunto de coeficientes. Ao terminar sua tarefa, cada *tread* terá a coluna que corresponde ao valor negativo mínimo do seu conjunto de trabalho. Depois disso, é necessário selecionar quais das *threads* possuem a coluna com o menor valor negativo. Para isso, é preciso que as *threads* comparem entre si suas colunas selecionadas, fazendo uso de uma variável global ou compartilhada, isto é, uma variável que todas tenham acesso. Essa variável terá o índice da coluna de menor valor entre todas as *threads*, e cada uma deverá comparar se o valor da coluna armazenada internamente é menor que a variável compartilhada, caso positivo deverá escrever seu valor. Observe que, como esclarecido na seção 2.3.1, isso deve ser tratado como uma seção crítica, portanto apenas uma *thread* por vez poderá fazer a comparação e escrever na variável global. Em sequência, existe uma barreira para assegurar que todos as *threads* tenham entrado na seção crítica antes de seguir para o próximo passo.Observa-se que, uma única *thread* é utilizada para atualizar algumas variáveis de controle, liberando, dessa forma, as outras *threads* para seguir para a próxima etapa.

Na etapa III, seleciona-se localmente, a linha que tenha o menor quociente e utiliza-se uma região crítica para selecionar o índice da linha global, da mesma maneira que acontece no passo II. Uma barreira sincroniza as *threads* para assegurar que todas elas tenham entrado na seção crítica antes de continuar as etapas seguintes. Além disso, antes

da etapa IV é necessário definir quem é o pivô, assim, para que as *threads* não avancem sem antes assegurar que todas tenham o pivô correto, uma outra barreira é posta antes da etapa IV. O pivô é definido a partir linha e a coluna global selecionadas nas etapas anteriores.

O laço paralelo da etapa IV atualiza a linha do pivô. Cada *thread* é responsável por atualizar um conjunto de coeficientes da linha do pivô. Para prosseguir, é necessário que esta etapa seja concluída, porque os valores desta linha são utilizados no próximo passo, portanto, precisa-se de uma barreira.

A etapa V é um laço paralelizado para a atualização das linhas de restrição restantes, ou seja, cada *thread* é designada para um conjunto de linhas (exceto a linha do pivô), para modificá-las utilizando a equação 4.2. Ao terminar as tarefas, cada *thread* pode ir para a próxima etapa sem ter que esperar pelas outras, pois o próximo passo é independente desta etapa.

A etapa VI utiliza o mesmo laço paralelizado para realizar três operações, ou seja, cada *thread* terá como carga de trabalho um conjunto de coeficientes da f.o. para executar as três operações. Inicialmente, será atualizada a linha da função objetivo a partir da equação 4.2. Em seguida, é selecionada localmente a coluna com o menor valor negativo da linha objetivo, como é feito na primeira parte do passo II; e finalmente, contabilizar os valores negativos da linha objetivo. Vale ressaltar que o critério de parada do algoritmo é satisfeito quando não existir nenhum valor negativo na f.o.. Logo após essa etapa, para garantir a contagem correta, é necessário uma barreira, se o critério de parada não for alcançado, o ciclo reinicia na seção crítica da segunda etapa para selecionar a coluna global.

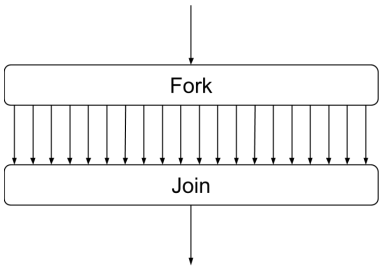
Observe a sincronização entre os passos, pois eles são dependentes uns dos outros. Por exemplo, o passo III só pode ser realizado se o índice da coluna é definido, o que é realizado no passo II. O passo IV, precisa do pivô, isto é, a coluna selecionada no passo II e a linha selecionada no passo III. E o passo V necessita da linha pivô atualizada. Complementando, na sincronização das seções críticas, tem-se muito tempo de *overhead*, o que pode prejudicar a execução paralela do algoritmo proposto. Contudo, note que, todas as sincronizações que foram utilizadas são indispensáveis para a corretude do algoritmo. Na próxima seção, será detalhada a implementação paralela.

4.2 Implementação do Algoritmo Simplex Multicore

A paralelização foi feita em C++ utilizando OpenMP (*Open Multi-Processing*), o qual utiliza um modelo *fork*(bifurcar) e *join*(unir). Existe um fluxo de execução principal chamado *master thread*, quando uma seção paralela é encontrada, *threads* são disparadas (*fork*) conforme necessário. Todas as *threads* começam a executar o código dentro daquela

seção paralela e ao fim da seção é feito um *join*, conforme listra a figura 20.

Figura 20: Diagrama do funcionamento do OpenMp.



Fonte: Autoria própria.

O OpenMp utiliza diretivas de compilador para especificar o comportamento do programa ao processar alguma entrada. A diretiva `#pragma omp parallel` declara uma seção paralela, criando *threads* como mostrado na figura 20. A criação das *threads* requer um tempo extra, o qual foi visto na seção 2.3.1, o qual é chamado de *overhead*. As diretivas utilizadas para dividir o trabalho nessa implementação somente funcionam dentro da seção paralela.

Parte do pseudo-código da implementação paralela simplex padrão pode ser visto nas figuras 21 e 22.

Figura 21: Primeira parte do código paralelo.

```
#pragma omp parallel escopo_de_variáveis
{
  //ETAPA II
  #pragma omp for nowait
  for (j = 0; j <= qtd_col; j++)
    /*Selecionar localmente a coluna que tenha o menor valor
     * negativo da função objeto.*/
  do{

      #pragma omp critical
      /*Selecionar valor mínimo global(pivo_col).*/
      #pragma omp barrier
      #pragma omp single nowait
      //Variáveis de Atualização.
      //ETAPA III
      #pragma omp for reduction(+:invalidos) nowait
      for (i = 0; i < qtd_lin; i++) {
          if(tabela[i][pivo_col] > 0){
            quociente =
              tabela[i][ultima_col] / tabela[i][pivo_col]
            /*Selecionar localmente a linha com o menor
             *quociente.*/
          }else{
            invalidos = invalidos + 1
          }
      }
      #pragma omp critical
      /*Selecionar o valor mínimo global(pivo_lin). */
      #pragma omp barrier
      #pragma omp single nowait
      {
        if (invalidos == qtd_lin) {
          exit(1);
        }
```

Figura 22: Segunda parte do código paralelo.

```
35      pivo = tabela[pivo_lin][pivo_col]
36      #pragma omp barrier
37      //Etapa IV
38      #pragma omp for
39      for (j = 0; j <= qtd_col; j++)
40          /*dividir cada elemento da linha do pivô
41            pelo pivô*/
42      //ETAPA V
43      #pragma omp for nowait
44      for (i = 0; i < qtd_lin; i++)
45        if( i != pivo_lin)
46          for (j = 0; j < qtd_col; j++)
47            tabela[i][j] =
48              tabela[i][j] -
49                (tabela[i][pivo_col] * tabela[pivo_lin][j])
50      //ETAPA VI
51      #pragma omp for reduction(+:conta)
52      for (j = 0; j <= qtd_col; j++){
53          // tabela[ultima_lin][_] -> Função Objetivo
54          tabela[ultima_lin][j] =
55            tabela[ultima_lin][j] -
56              (tabela[ultima_lin][pivo_col] * tabela[pivo_lin][j])
57          if(tabela[ultima_lin][j] < 0){
58             /*Selecionar localmente a coluna com o menor
59               valor negativo da linha objetivo.*/
60              conta = conta + 1
61          }
62      }
63    } while(conta > 0)
64 }
```

Percebe-se que as *threads* são disparadas no início para evitar repetição de *overhead* de criação e destruição de *threads* (linha 1; figura 21). Para efeito de otimização do código, a primeira parte da etapa II é executada antes do começo do laço (linhas 4-7; figura 21). Percebe-se ainda na primeira linha, que é possível selecionar o escopo de cada variável do programa que irá ser paralelizado. O OpenMp tem 6 tipos diferentes de escopo, são eles:

Shared: Os dados dentro de uma região paralela são compartilhados, o que significa visível e acessível por todas as *threads* em simultâneo.

Private: Os dados em uma região paralela são individuais para cada *thread*. Com este recurso cada *thread* terá uma cópia local e será utilizada como uma variável temporária. Uma variável não é inicializada e o valor não é mantido para o uso fora da região paralela;

Firstprivate: É um escopo como o `private`, exceto ao inicializar pelo valor original;

Lasprivate: Também semelhante ao `private`, exceto que o valor original é atualizado depois da construção.

Default: Permite ao programador declarar o escopo padrão para os dados os quais serão utilizados pelas *threads*.

None: Força o programador a declarar cada variável na região paralela utilizando as cláusulas atribuídas ao compartilhamento de dados.

A cláusula `#pragma omp for` (linha 4;figura 21), divide o intervalo de iteração do laço entre as *threads* automaticamente. Esse intervalo do laço se dá a partir do primeiro elemento das colunas até *qtd_col*, isto é, até o último elemento. Por padrão no OpenMp, existe uma barreira implícita que sincroniza todas as *treads* ao final de um laço paralelizado com `#pragma omp for`. Ou seja, somente, depois de todas as *threads* executarem seus trabalhos é que elas continuarão no resto do código do programa. Porém, é possível adicionar uma cláusula na diretiva de tal forma que possa alterar o seu comportamento. A claúsula `nowait`, é usada a fim de eliminar uma barreira. E, no caso da linha 4, não há necessidade de sincronização após a conclusão das tarefas de cada *thread*.

A segunda parte da etapa II é encontrar o índice global da coluna entre as soluções encontradas por cada *thread*. Como já foi explicado anteriormente, esta é uma seção crítica e para resolver esse problema o OpenMp possui uma diretiva chamada `#pragma omp critical` (linha 10; figura 21). Essa diretiva garante que todo o código executado dentro do seu escopo, somente é executado por uma *thread* por vez. Por tanto, irá assegurar que apenas uma *thread* poderá fazer a comparação e escrever na variável global (*pivo_col*) por vez.

A diretiva `#pragma omp barrier` (linha 12; figura 21) cria uma barreira de sincronização para certificar que todas as *threads* tenham entrado na seção crítica antes de continuar. Isso garante que a variável *pivo_col* seja correto. A diretiva `#pragma omp single` é usada para que somente uma *thread* execute o código contido dentro do escopo, nesse caso, para atualizar algumas variáveis de controle (linhas 13-14; figura 21). O `omp single` também possui uma barreira implícita e a cláusula `nowait`, serve para que as outras *threads* não precisem esperar pela única *thread* no interior do `single`.

A terceira etapa divide um conjunto de linhas para cada *thread* o qual encontra, localmente, a linha que tem menor valor do quociente (linhas 17-26; figura 21). O intervalo desse laço começa a partir da primeira linha até a última (*qtd_lin*). Note que a primeira parte dessa etapa não precisa de barreira, igualmente como ocorre na primeira etapa do passo II.

Retomando a equação 4.1, perceba que, como B^{-1} é uma matriz identidade, logo $B^{-1}b = b$. E, y_i^k é o coeficiente da linha i da coluna da variável não-básica k selecionada para entrar na base. Ou seja, é o coeficiente da coluna do pivô, da linha i. Em outras palavras, a equação 4.1, também pode ser reescrita como (linha 19-20; figura 21):

$$x_s^B = min\left\{\frac{\text{tabela}[i][ultima\_col]}{\text{tabela}[i][pivo\_col]}, \text{tabela}[i][pivo\_col] > 0\right\}, \forall i \in I \mid I = [1 \;\; m], \quad (4.3)$$

Onde: *ultima_col* é o índice da última coluna; tabela$[i][ultima_col]$ é o valor da constante do lado direito indexado pelo índice i, também chamado de valor independente; tabela é uma matriz que contem os coeficientes das restrições, cada linha se refere a uma restrição e

cada coluna se refere a uma variável; i é o índice da linha; e *pivo_col* é o índice da coluna do pivô.

A cláusula `reduction` (linha 16; figura 21) faz uma cópia local da variável *invalidos* em cada *thread*. Ao término da tarefa de cada *thread*, os valores das cópias locais são resumidos (reduzido) para uma variável compartilhada global no final do laço. Note que essa variável somente é incrementada se tabela$[i][pivo_col] < 0$, caso contrário deve-se calcular o quociente da linha 19.

Observe o uso da diretiva `#pragma omp critical` (linha 27; figura 21) para encontrar globalmente o índice da linha entre as *threads*. O índice encontrado será a linha do pivô (*pivo_lin*). Em seguida, existe uma barreira (linha 29; figura 21) que serve para garantir que *pivo_lin* tenha o valor correto, não prejudicando o valor final do pivô. O `#pragma omp single nowait` da linha 30 (figura 21) serve para atualizar algumas variáveis e, principalmente, para testar se todos coeficientes da coluna do pivô são negativos ($invalidos == qtd\_lin$ (linha 32; figura 21)), pois, se for o caso, o programa deve parar, indicando que não há solução ótima e o método simplex não pode continuar.

A segunda barreira (linha 36; figura 22) é para garantir que todas as *threads* tenham o mesmo valor de pivô antes de continuar para a próxima etapa. Note que o pivô é o elemento da tabela de restrições indexado por *pivo_lin* e *pivo_col*

A quarta etapa paraleliza a divisão da linha pelo pivô (linhas 38-41; figura 22). Percebe-se que existe uma barreira implícita após a execução do laço, isso é necessário já que, os próximos passos necessitam dessa linha normalizada pelo pivô.

O passo V (linhas 43-49; figura 22) é responsável por atualizar as restrições restantes, utilizando a fórmula apresentada nas linhas 47 à 49 que é a mesma em 4.2. Note a presença da cláusula `nowait` na diretiva `#pragma omp for`.

O último passo consiste em três operações no mesmo laço: a modificação da última linha usando a mesma fórmula utilizada no passo anterior (linhas 54-56; figura 22); se existir algum valor negativo na f.o., então deve-se selecionar a coluna com o elemento negativo de menor valor na linha objetivo (linhas 57-59; figura 22), isto é, a mesma operação que é feita na primeira parte do passo II; e um incremento da variável *conta* (linha 60; figura 22). Note ainda, a ausência da cláusula `nowait`, ou seja, a presença da barreira implícita.

Observe a presença da cláusula `reduction` (linha 51; figura 22) para somar todos os coeficientes negativos da f.o. e atribuir esse valor para a variável *conta*. No final (linha 63) tem-se o teste para o critério de parada. Se houver quaisquer números negativos na linha objetivo, o algoritmo continua, retornando para o `critical` da linha 10 (figura 21). No caso, basta a variável *conta* ter algum número positivo.

Percebe-se que, não é tão simples implementar um algoritmo paralelo. Para que o

programador possa extrair o máximo de desempenho de sua aplicação, ele deverá estar sempre atento aos conceitos de paralelismo, como: regiões críticas, barreiras, *overhead* de sincronização e de criação/destruição de *threads*, regiões de código que possuem ou não dependência, e configuração de escopo de variáveis. Aprofundar-se na linguagem utilizada e nas diretivas de compilador, também é importante para otimizar o código paralelo o máximo possível.

No próximo capítulo serão apresentados os resultados adquiridos com essa aplicação, bem como a análise de *speedup* e de escalabilidade.

5 Experimentos e Resultados

Para realização dos experimentos, o computador usado possui dois AMD Opeteron 6172 com 12 cores 2.1Ghz, 16 GB DDR3 RAM, 12 x 64KB de cache L1, 12 x 512 KB L2 e 2 x 6 MB L3, contendo a versão Ubuntu 12.04.1 LTS. Para gerar o conjunto de problemas de PL utilizados nos experimentos foi necessário montar a tabela inicial representada na tabela 4:

Tabela 4: Tabela Inicial

	VNB + VB	**Lado direito**
VB	A	b
Z	$-c$	**0**

Onde A é a matriz de restrições, c é o vetor de coeficientes da função objetivo e b é o vetor dos valores independentes. A fim de gerar os problemas de teste foi implementado no Octave[1] um método em que os valores de A, c e b eram gerados com números aleatórios. Reforça-se que, a quantidade de linhas de um problema se refere ao número de restrições e a quantidade de colunas, ao número de variáveis. As dimensões dos problemas foram os seguintes: 256, 384, 512, 768, 1024, 1536, 2048, 3072, 4096. O número de linhas e o de colunas dos problemas foram definidos pela combinação desses números e cada problema foi gerado 5 vezes com diferentes dados, assim, obtendo 405 (9 * 9 * 5) problemas. Por exemplo, o problema 256x256 tem 5 diferentes instâncias: 256x256_1, 256x256_2, 256x256_3, 256x256_4, 256x256 _5, e todas elas possuem valores diferentes. O tempo de execução é a média dessas instâncias.

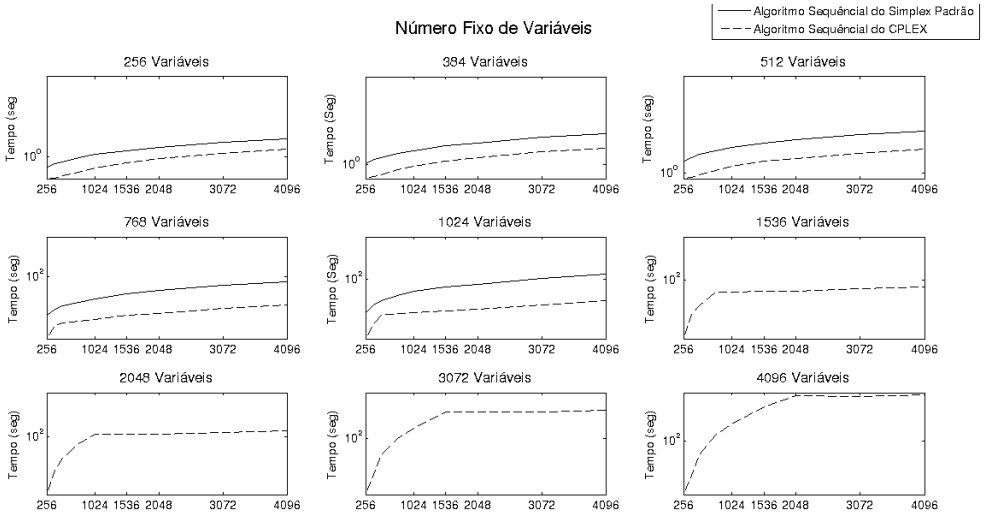
5.1 Análise de Speedup

A figura 23 mostra o tempo sequencial do método Simplex padrão na forma tabular e do algoritmo do Simplex do CPLEX® com o número fixo de variáveis e variando a quantidade de restrições. A quantidade de iterações do algoritmo Simplex padrão teve que ser limitada em 5000, devido ao alto tempo gasto para encontrar a solução dos problemas de grande escala. Acima de 1536 variáveis, o Simplex Padrão chegou ao limite de iterações, logo para não prejudicar a comparação com o CPLEX®, as curvas de desempenho do Simplex Padrão para esses problemas não foram apresentados. Assim, não se exibe o

[1] GNU Octave é uma linguagem de alto nível interpretada, principalmente para cálculos numéricos. Ele fornece recursos para a solução numérica de problemas lineares e não lineares, e também realiza outros tipos de experimentos numéricos. A linguagem Octave é bastante semelhante ao Matlab® fazendo com que alguns programas e scripts de ambas ferramentas sejam compatíveis.

desempenho do Simplex padrão para os gráficos dos problemas com 1536 até 4096 variáveis da figura 23.

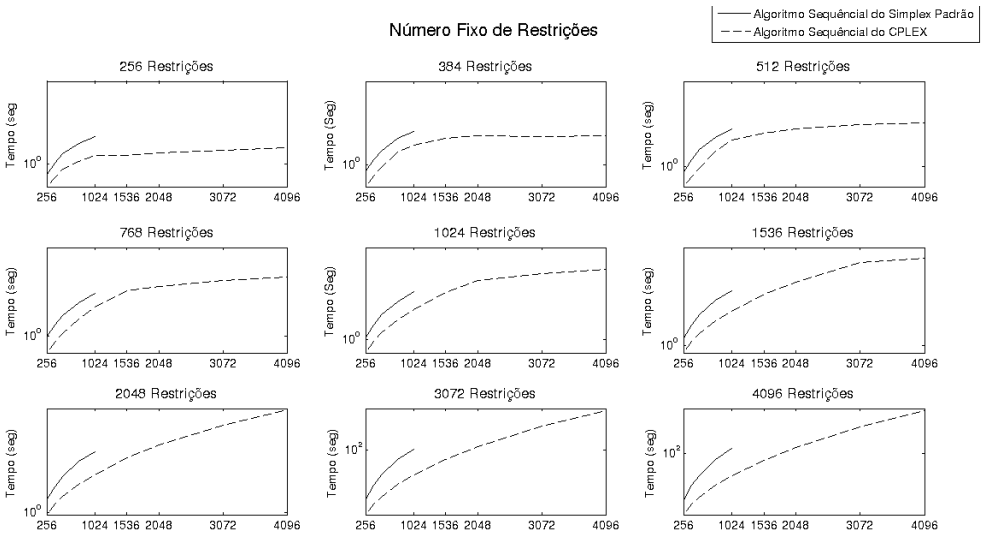
Figura 23: Tempo sequencial do algoritmo Simplex e o Simplex do CPLEX®, fixando-se o número de variáveis.



Note que o CPLEX® é mais rápido que o Simplex Padrão. Como visto na seção 3.3.1, o CPLEX® utiliza de algoritmos mais otimizados e técnicas refinadas para adquirir melhor desempenho.

A figura 24 mostra o tempo sequencial do Simplex padrão e do algoritmo do Simplex do CPLEX® com o número fixo de restrições, variando a quantidade de variáveis.

Figura 24: Tempo sequencial do algoritmo Simplex padrão e o Simplex do CPLEX®, fixando-se o número de restrições.



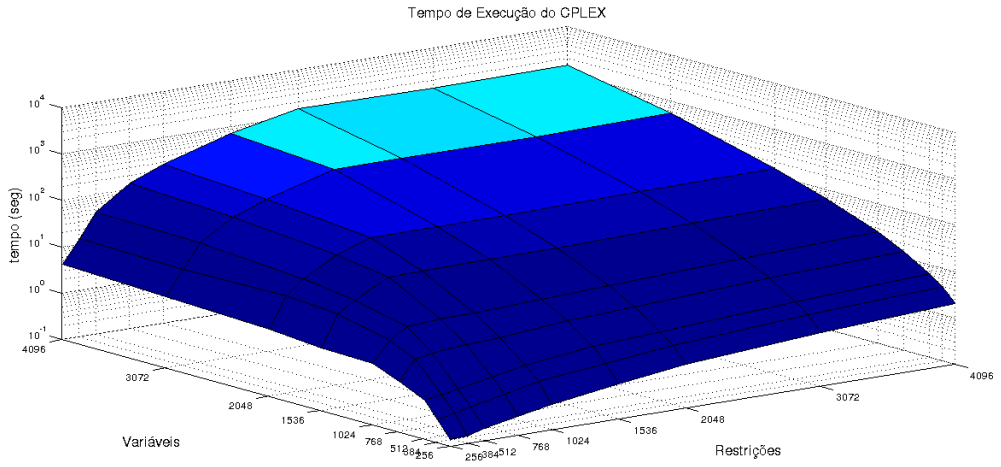
Assim como na figura 23, o CPLEX® tem melhor tempo sequencial em todos os gráficos. Note que a curva de desempenho do algoritmo Simplex padrão está exibida até

1024 variáveis, pois acima desse valor o Simplex alcança o limite de iterações. O CPLEX® se mostra com menor tempo em todos os gráficos.

Um dos motivos para o CPLEX® ter melhor desempenho é que precisa de menos iterações do que o Simplex Padrão para encontrar a solução. O CPLEX® ultrapassou o limite de iterações somente quando a quantidade de variáveis é de 4096.

A figura 25 mostra o tempo sequencial do algoritmo Simplex do CPLEX®. Perceba que para problemas com mais restrições que variáveis o tempo de execução é bem menor. Como por exemplo, o problema com 512 variáveis e 3072 restrições tem tempo de execução menor que seu oposto. Note, também que o CPLEX® resolveu os problemas com poucas restrições e mais variáveis de forma mais eficiente.

Figura 25: Tempo sequencial do algoritmo Simplex do CPLEX®.



O algoritmo Simplex paralelo proposto teve seu desempenho comparado ao do CPLEX®, em virtude deste ter apresentado melhor performance para resolução dos problemas. Isso já era esperado, visto que o CPLEX® trata-se de uma ferramenta para solução de problemas de otimização conhecido comercialmente. Dessa forma, ele servirá de referência para verificar a escalabilidade e eficiência do algoritmo paralelo. Para efeito de complemento, essas comparações de desempenhos também serão feitas nas mesmas condições para o algoritmo Simplex Padrão.

As figuras 26, 27, 28, 29, 30 apresentam os gráficos dos *speedups* de 2, 4, 8, 16 e 24 *threads*, respectivamente. Note que para 2 *threads* existe uma super linearidade ultrapassando 2 unidades. Isso ocorre para alguns problemas acima de 2048 variáveis. Principalmente, para os problemas em que há mais variáveis que restrições. Para 2 *threads*, o *speedup* chegou 5.4, indicando que o algoritmo paralelo proposto chegou a ser 5.4 vezes mais rápido que o CPLEX®.

O gráfico 27 possui curvas semelhantes com as apresentadas na figura 26, contudo

Figura 26: Speedup para 2 *threads*.

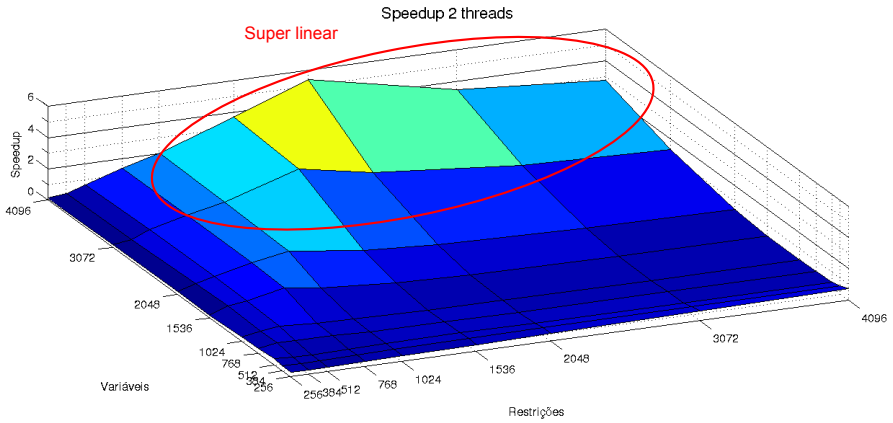


Figura 27: Speedup para 4 *threads*.

Figura 28: Speedup para 8 *threads*.

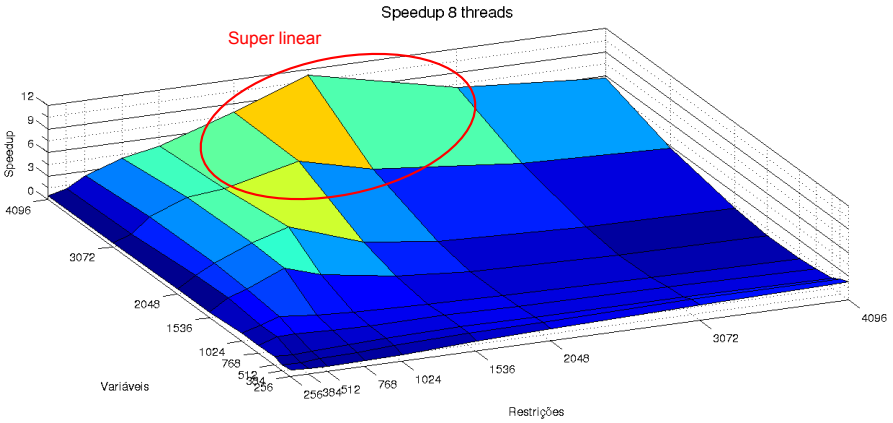


Figura 29: Speedup para 16 *threads*.

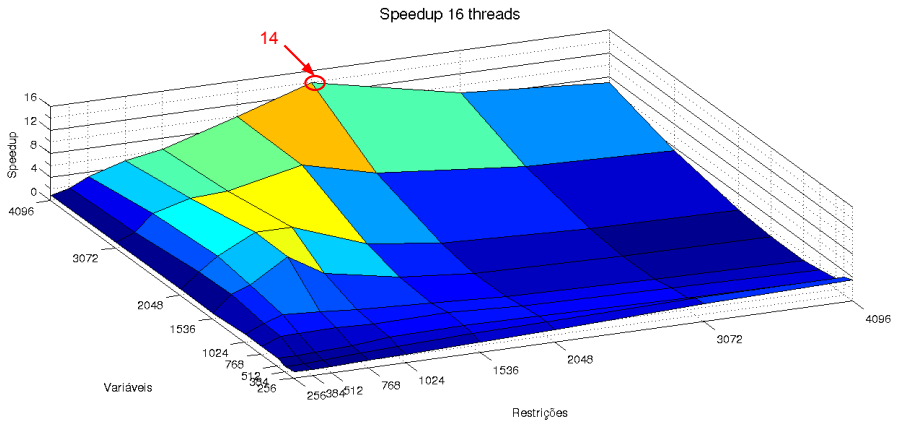
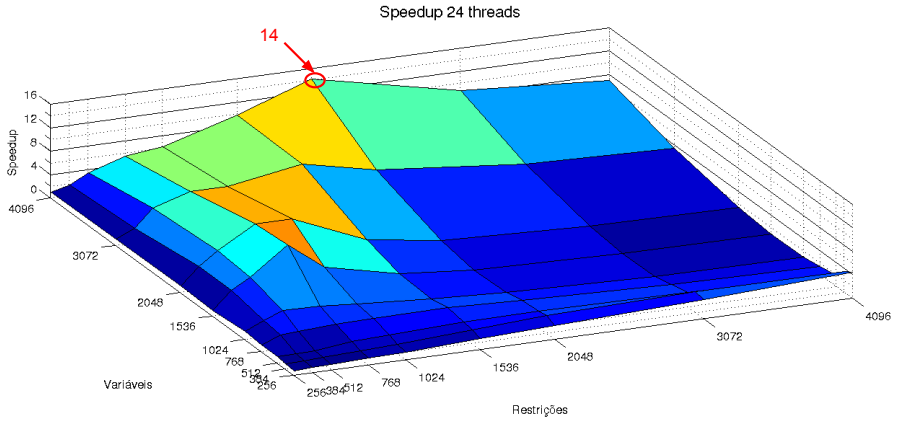


Figura 30: Speedup para 24 *threads*.



possui um *speedup* ainda maior. Para o problema 2048 restrições com 4096 variáveis, obteve-se um *speedup* de 9.5, mostrando que para 4 *threads* o algoritmo paralelo foi 9.5 vezes mais rápido que o CPLEX®. Perceba, também, que para a maioria dos problemas acima de 2048 variáveis, o *speedup* é maior que 4. Isso apresenta a super linearidade para esse gráfico.

O gráfico 28 também apresenta uma super linearidade e consegue ser até 11 vezes mais rápido que o CPLEX®. O gráficos 29 e 30 não tem super linearidade, contudo em ambos os gráficos, o algoritmo paralelo é 14 vezes mais rápido que o CPLEX®.

Nota-se outras características importantes nesses gráficos. Primeiro, que à medida que se aumenta a quantidade de *threads*, o *speedup* também cresce, o que é normal em sistemas paralelos (ver subseção 2.3.2.1). Segundo, para os problemas com mais variáveis que restrições, o CPLEX® não teve uma boa performance, enquanto que o algoritmo paralelo apresentou ser melhor. E, por fim, existe uma queda no *speedup* em todos os gráficos, as possíveis causas dessa queda de desempenho serão discutidas na próxima seção.

5.2 Análise da escalabilidade

Todos os problemas foram executados usando a implementação paralela proposta com 2, 4, 8, 16 e 24 *threads*. O tempo de execução é a média entre os tempos de execução dos 5 problemas do mesmo tamanho.

As figuras 31, 32, 33, 34 e 35, mostra os gráficos para 2, 4, 8, 16 e 24 *threads*, em relação ao Simplex Padrão. Primeiramente, perceba que os gráficos são mostrados até 1024 variáveis, pois acima desse valor o Simplex padrão ultrapassa o limite de iterações. Observa-se que à medida que aumenta o número de *threads*, a eficiência para todos os problemas diminui. Note que para 2 e 4 *threads* os valores de eficiência para todos os problemas são próximos de 1, o que indica uma boa utilização dos processadores.

Figura 31: Eficiência para 2 *threads*, em relação ao Simplex padrão.

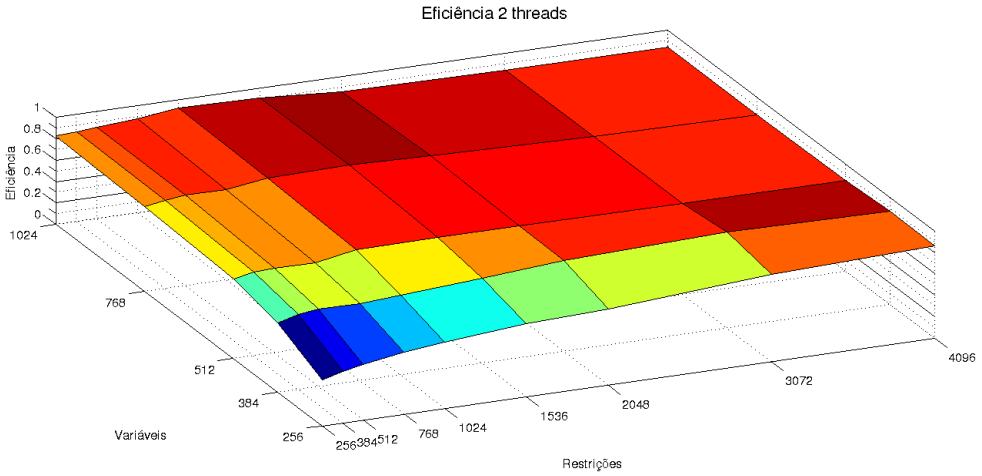
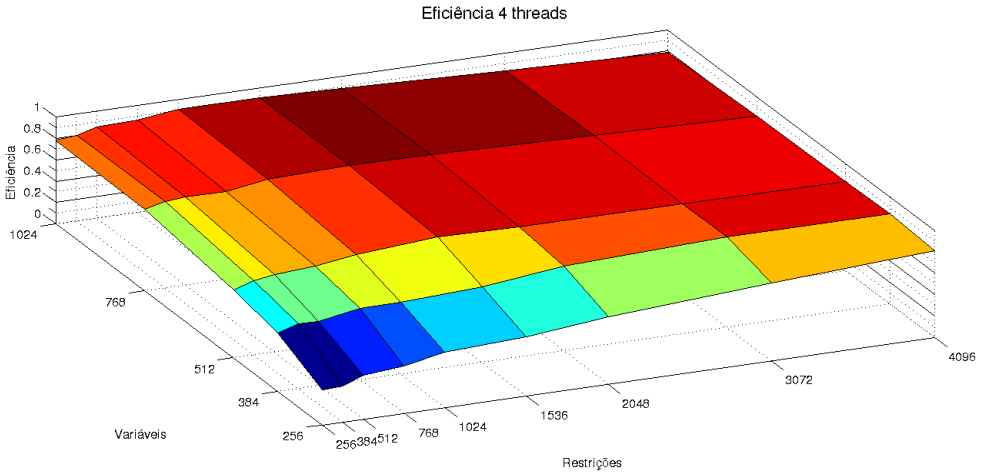


Figura 32: Eficiência para 4 *threads*, em relação ao do Simplex padrão.



Note pelos gráficos de 8, 16 e 24 *threads* que para os problemas com mais variáveis que restrições obteve-se melhor eficiência. A seta vermelha indica a direção em que a

Figura 33: Eficiência para 8 *threads*, em relação ao Simplex padrão.

Figura 34: Eficiência para 16 *threads*, em relação ao Simplex padrão.

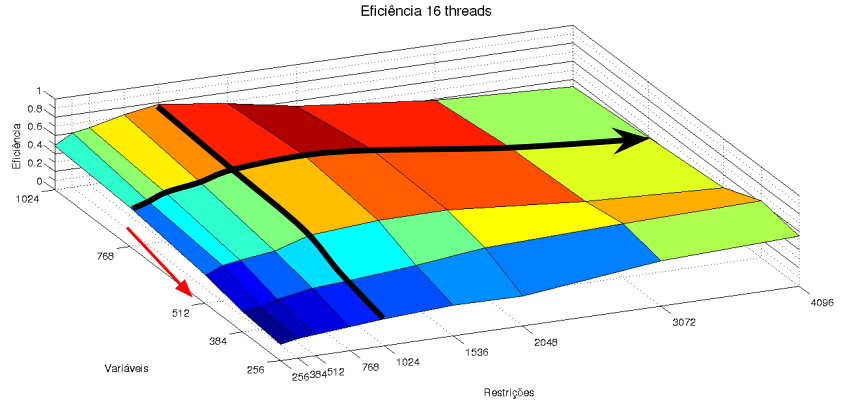
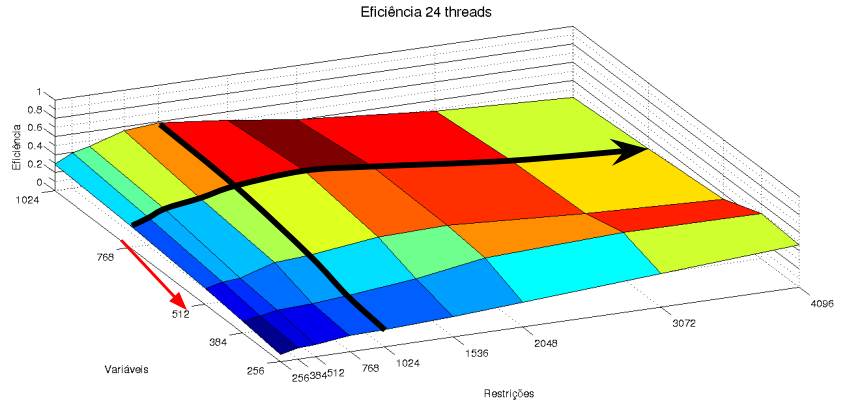


Figura 35: Eficiência para 24 *threads*,em relação ao Simplex padrão.



eficiência cai nesses gráficos. Por exemplo, o problema com 1024 variáveis e 256 restrições nos gráficos 33, 34 e 35 tem maior eficiência que seu oposto. Note a demarcação em 1024 restrições nesses gráficos. Isso mostra o limite para analisar a escalabilidade desses problemas, 1024 variáveis por 1024 restrições. Isso é necessário, devido a limitação do gráfico em 1024 variáveis, como já explicado. Acima desse valor, não há um número de restrições correspondente para realizar a análise.

O algoritmo é escalável para todos os gráficos. Os gráficos de eficiência para 2 e 4 *threads* se mantém próximo de 1, mostrando que o algoritmo é escalável independente do tamanho dos problemas. Já as figuras 33, 34 e 35 o algoritmo é escalável até certo tamanho do problema. Pois, a eficiência diminui com o aumento da quantidade de linhas e de colunas. A seta em cima do plano do gráfico, mostra como esse se comporta ao aumentar a quantidade de restrições. A escalabilidade aumenta até uma certa quantidade de restrições e, logo em seguida, começa a declinar. Esse efeito foi causado pela quantidade limitada de memória *cache* do servidor. Não há espaço suficiente para acomodar todos os valores dos problemas de grande escala, fazendo com que os dados sejam buscados na memória RAM, prejudicando, assim, o desempenho - gargalo de Von Neumann.

A quantidade total de *bytes* do problema de programação linear não pode ser maior que 12 MB, que é o tamanho total da cache L3 do servidor. Os dados da matriz, que representa a tabela do Simplex, foi definida com o tipo primitivo `double`. Esse tipo de dados necessita 8 *bytes* para armazenamento. Logo, tem-se que:

$$V \times R \times 8bytes \leq 12MB \Rightarrow V \times R \leq \frac{12MB}{8B} \Rightarrow V \leq \frac{1.572.864}{R}, \tag{5.1}$$

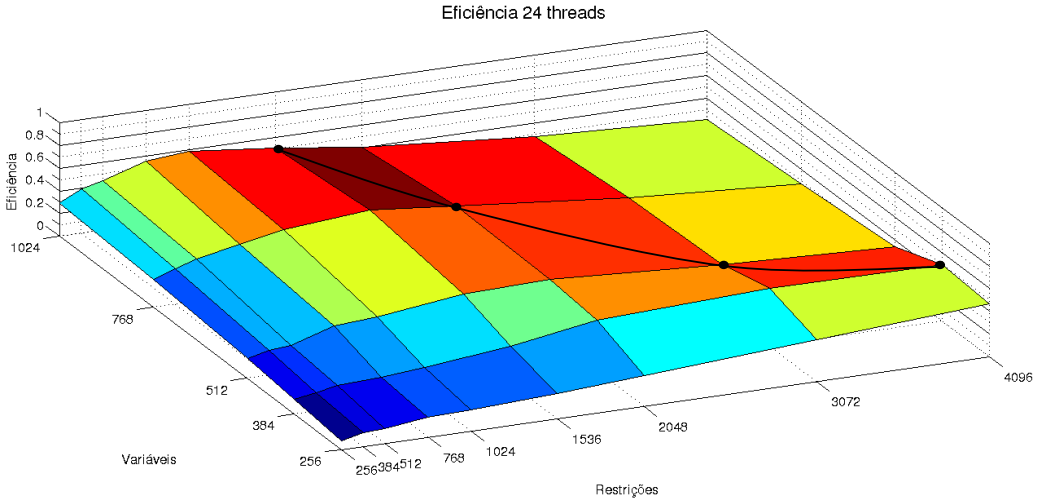
onde R é a quantidade de restrições e V a quantidade de variáveis.

A equação 5.1 demonstra o limite máximo de variáveis e restrições que podem ser usados sem que haja perda de desempenho, pois acima desse limite, acontece o problema da cache citada anteriormente. Substituindo, R pelos valores das restrições usadas nas dimensões dos problemas, isto é: 256, 384, 512, 768, 1024, 1536, 2048, 3072 e 4096; tem-se como valores máximos de variáveis, respectivamente: 6144, 4096, 3072, 2048, 1536, 1024, 768, 512 e 384. Ou seja, para 384 restrições a quantidade de variáveis não pode ultrapassar 4096, caso contrário, a memória cache não irá comportar. Vale ressaltar que 256 restrições tem como limite 6144 variáveis, e no caso, a quantidade máxima de variáveis é 4096. Ou seja, para 256 restrições ainda é possível aumentar a quantidade de variáveis.

A figura 36 apresenta um gráfico de eficiência para 24 *threads*. A linha ponteada mostra os valores de eficiência destacados que correspondem aos problemas limitantes designados pela equação 5.1. Esses valores, indicam onde ocorre a primeira queda de eficiência. Após a demarcação, note que a eficiência começa a diminuir, pois o tamanho do problema ultrapassa o limite máximo de armazenamento da cache. Dessa forma, mostra-se com mais clareza a região em que há a queda de eficiência causada pelo limite de espaço

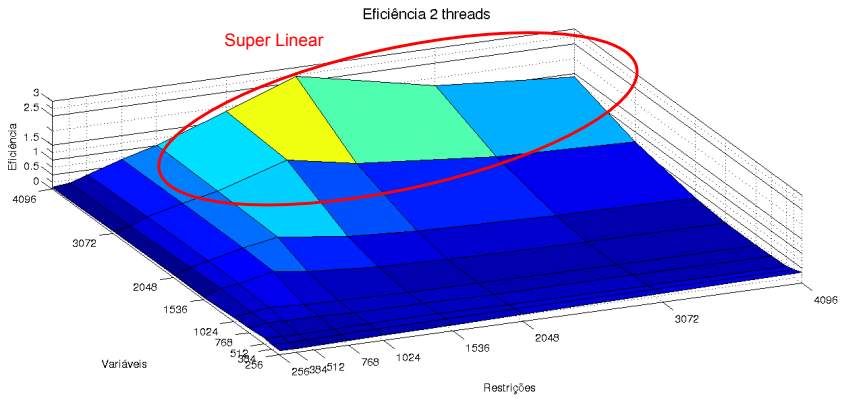
da cache L3. Destaca-se que essa análise foi baseada no gráfico de eficiência de 24 *threads*, porém pode ser estendida para os outros gráficos.

Figura 36: Gráfico de eficiência para 24 *threads*, mostrando o limite dos problemas designados pela equação 5.1.



As figuras 37, 38, 39, 40 e 41 mostram os gráficos de eficiência para 2, 4, 8, 16 e 24 *threads*, em relação ao desempenho do CPLEX®. Inicialmente, observa-se que à medida que aumenta o número de *threads*, a eficiência para todos os problemas diminui. Como foi visto na seção 2.3.2.1, isso é natural nos sistemas paralelos, devido a própria definição de eficiência (E = S/P). Note que para 2, 4 e 8 *threads* os valores de eficiência para alguns problemas são maiores que 1, o que indica uma eficiência super linear. Enquanto, que para 16 e 24 *threads* obteve uma eficiência sublinear, ou seja, valores abaixo de 1. Contudo, a eficiência com 16 *threads* obteve eficiência próximo de 1 e com 24 *threads* próximo de 0.6.

Figura 37: eficiência para 2 *threads*, em relação ao Simplex do CPLEX®.



Note que para todas as *threads*, o valor da eficiência escala com o aumento da magnitude dos problemas, até um certo valor de variáveis e restrições. Dessa forma, pode-se dizer que o algoritmo possui escalabilidade para esses conjuntos.

Figura 38: eficiência para 4 *threads*, em relação ao Simplex do CPLEX®.

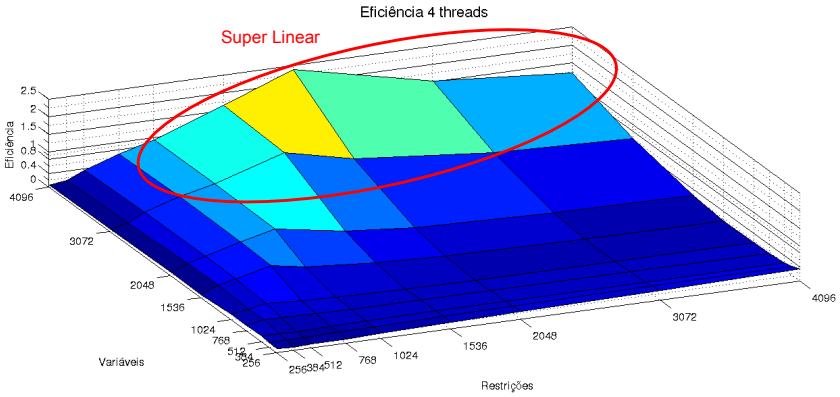


Figura 39: eficiência para 8 *threads*, em relação ao Simplex do CPLEX®.

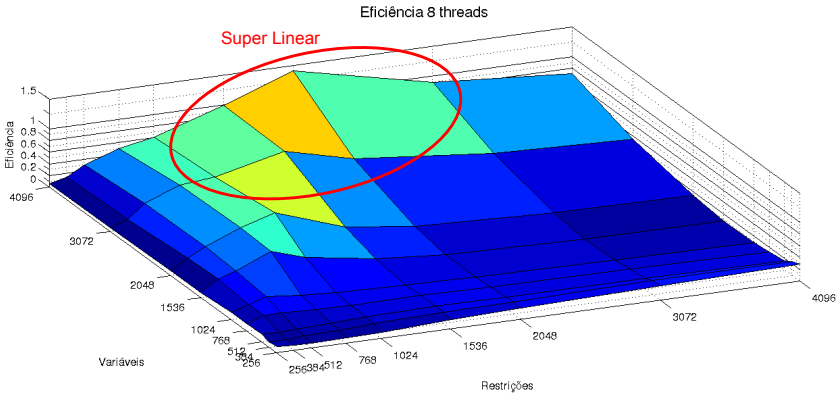


Figura 40: eficiência para 16 *threads*, em relação ao Simplex do CPLEX®.

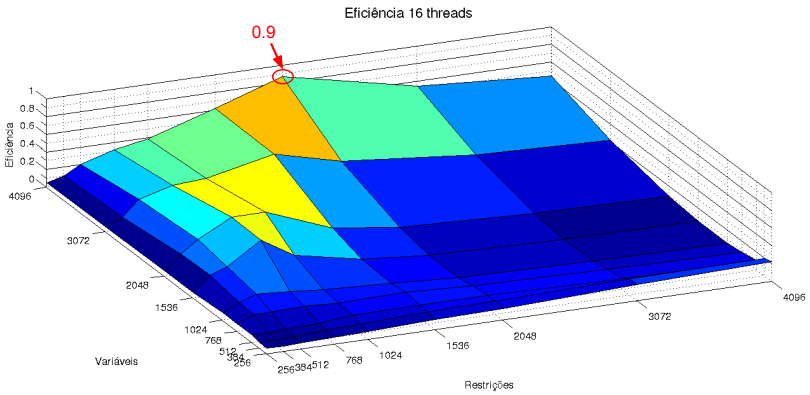
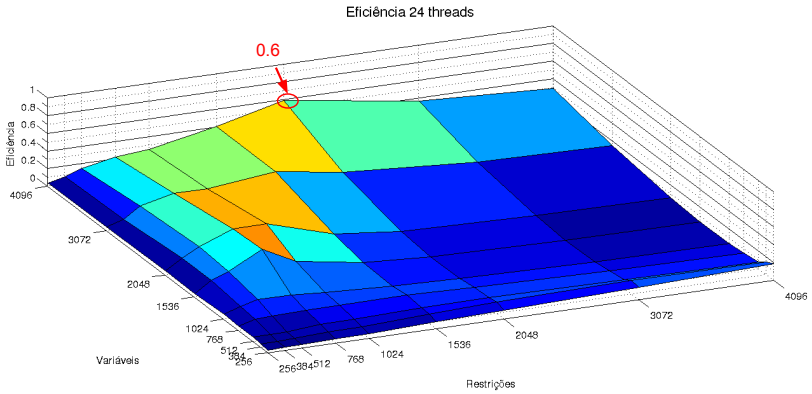
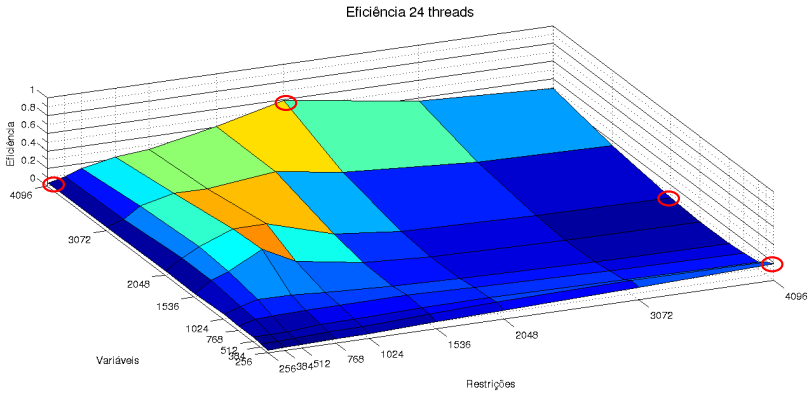


Figura 41: eficiência para 24 *threads*, em relação ao Simplex do CPLEX®.



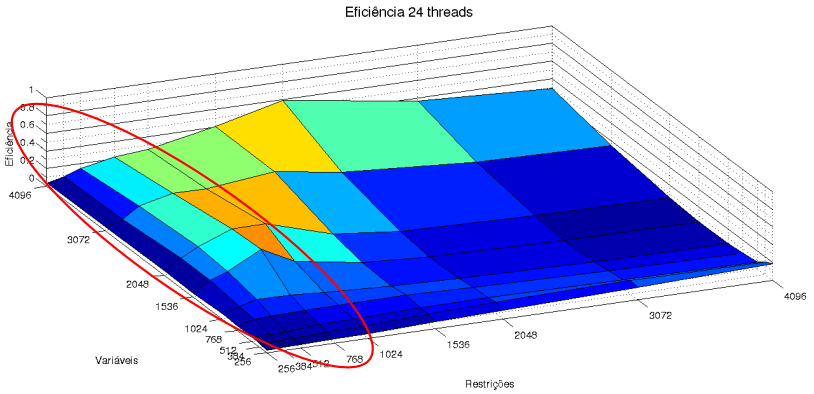
Pode-se analisar na figura 42, a variação de eficiência para problemas de diferentes dimensões. Fazendo uma análise por esse gráfico pode-se identificar em que momento é melhor executar o simplex paralelo. Por exemplo, claramente nota-se que para o problema com 4096 variáveis e 2048 restrições possui maior eficiência que seu oposto. Ou seja, o problema com 2048 variáveis e 4096 restrições é menos escalável. Outro exemplo é o problema com 256 variáveis e 4096 restrições, a eficiência é maior que seu oposto. Logo, é preferível executar, para esse caso, o problema de menor variáveis que restrições. Essa análise pode ser expandido para os outros gráficos com 2, 4, 8 e 16 *threads*, podendo diagnosticar as dimensões dos problemas que são mais escaláveis, ou seja, mais eficientes.

Figura 42: eficiência para 24 *threads*, em relação ao Simplex do CPLEX®.



Importante observar que, o gráfico de eficiência relacionado ao CPLEX®, perde escalabilidade para problemas com poucas restrições, como mostrado no gráfico 43. Isso ocorreu, pois, o tempo de execução do CPLEX® foi menor para os problemas com mais variáveis e poucas restrições, como pode ser visto na figura 25.

Figura 43: eficiência para 24 *threads*, em relação ao Simplex do CPLEX®.



O algoritmo do Simplex Padrão sequencial na sua versão mais simples, sem otimização algorítmica, tem menor desempenho quando comparado ao CPLEX®. Contudo, nossa implementação paralela do algoritmo Simplex Padrão obteve melhor performance quando comparado ao CPLEX®. Isso nos leva crer que otimizações no algoritmo proposto possam levar a desempenho ainda melhores do que os registrados.

Conclusão

Este trabalho apresentou uma implementação paralela escalável e eficiente do algoritmo Simplex padrão na arquitetura de processadores *multicore* para resolver problemas de programação linear de grande escala. Na fundamentação teórica foi descrito o funcionamento do método Simplex Padrão na forma tabular (*tableau*) para facilitar o entendimento da nossa proposta de paralelização. Foi feito uma breve explanação sobre as arquiteturas paralelas as quais foram abordados nos trabalhos correlatos. E ainda, no capítulo de fundamentação, foram apresentados os conceitos de escalabilidade e as métricas utilizadas para medir escalabilidade paralela. Na revisão bibliográfica foi mostrado os trabalhos relacionados ao tema desse texto e algumas ferramentas de otimização, destacando o CPLEX®. Logo em seguida, explicou-se o esquema geral da paralelização, abordando a ideia geral de como o algoritmo Simplex *Tableau* foi paralelizado. Conjuntamente, foi detalhado cada etapa da implementação paralela proposta em OpenMp.

Para realizar as analises, gerou-se problemas de teste, cujas dimensões são combinações de: 256, 384, 512, 768, 1024, 1536, 2048, 3072, 4096. Na solução desses problemas foi utilizado o Simplex paralelo proposto, o CPLEX® e o Simplex Padrão na forma tabular. Na análise do *speedup*, realizou-se a comparação dos tempos sequenciais para a resolução dos problemas através dos algoritmos Simplex Padrão e do CPLEX® com o número fixo de variáveis e alteração na quantidade de restrições e, vice-versa. O CPLEX®, teve melhor desempenho para todos os problemas, principalmente, devido a quantidade de iterações ser bem menor que o Simplex Padrão sequencial. Vale salientar que foi limitado a quantidade de iterações imposta ao método Simplex Padrão. Isso foi feito devido ao tempo não prático para resolução dos problemas de grande porte.

Com relação à análise da escalabilidade, todos os problemas foram executados usando a implementação paralela proposta com 2, 4, 8, 16 e 24 *threads*, considerando o desempenho do CPLEX® e o do Simplex Padrão. Os gráficos da eficiência paralela, considerando o CPLEX® para 2,4 e 8 *threads* apresentaram valores super lineares. Enquanto para 16 e 24 *threads* teve uma eficiência de 0.9 e 0.6, respectivamente. Em nenhum dos gráficos da eficiência paralela comparado ao Simplex Padrão apresentou super-linearidade. Embora que para 2 e 4 *threads* a eficiência é próxima do ideal.

Afirma-se também, com base nos conceitos de Gustafson (GUSTAFSON, 1988), que o algoritmo é escalável, pois, o valor da eficiência aumenta, com o crescimento da magnitude dos problemas. Entretanto, para os gráficos de eficiência do algoritmo paralelo em relação ao simplex Padrão com 8, 16 e 24 *threads*, quando aumenta o tamanho em *bytes* do problema além da capacidade da cache ($Variáveis \times Restrições \times 8B \leq 12MB$),

ocorre uma queda no valor da eficiência. Isso foi causado pelo gargalo de Von Neumann, ou seja, insuficiência da memória *cache* para acomodar todos os valores dos problemas de grande dimensão.

Outra análise importante realizada foi identificar em que situação é melhor executar a nossa aplicação do Simplex paralelo. Os gráficos de eficiência paralela, considerando o desempenho do CPLEX®, mostraram que a maioria dos problemas com mais variáveis que restrições tem melhor desempenho. Isso indica que o tempo sequencial do CPLEX® foi maior para boa parte dos problemas com mais variáveis que restrições. Já a eficiência paralela considerando o Simplex Padrão mostrou que é sempre melhor resolver os problemas com mais variáveis que restrições. Isso aponta que o tempo sequencial do Simplex Padrão foi maior para todos esses tipos de problemas.

Um outro resultado importante deste trabalho consiste no bom desempenho da implementação paralela do Simplex, quando comparado com o CPLEX®. O algoritmo paralelo proposto tem melhor eficiência ao resolver problemas de grande escala com mais variáveis que restrições tanto comparado ao CPLEX® quanto ao Simplex Padrão.

A importância desses resultados pode levar a escolha do problema de Programação Linear na sua forma original ou dual para economizar o tempo de processamento.

Referências

ALMASI, G. S.; GOTTLIEB, A. *Highly Parallel Computing*. Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc., 1989. ISBN 0-8053-0177-1. Citado na página 49.

ARB, O. *The OpenMP*. 2013. Disponível em: <http://openmp.org/wp/about-openmp/>. Citado na página 53.

BORKAR, S. Thousand core chips: A technology perspective. In: *Proceedings of the 44th Annual Design Automation Conference*. New York, NY, USA: ACM, 2007. (DAC '07), p. 746–749. ISBN 978-1-59593-627-1. Citado 4 vezes nas páginas 23, 55, 69 e 70.

CANTANE, D. R. *Contribuição da Atualização da Decomposição LU no Método Simplex*. Tese (Doutorado) — UNICAMP, São Paulo, SP, ago. 2009. Citado na página 23.

COIN-OR. *COIN-OR Annual Report 2007*. [S.l.], 2007. Citado na página 72.

COIN-OR Projects. 2013. Disponível em: <http://www.coin-or.org/projects/>. Citado na página 72.

CPLEX Optimizer. 2013. Disponível em: <http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/>. Citado na página 71.

CULLER, D. E.; GUPTA, A.; SINGH, J. P. *Parallel Computer Architecture: A Hardware/Software Approach*. 1st. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997. ISBN 1558603433. Citado na página 47.

DONGARRA, J.; SULLIVAN, F. Guest editors' introduction: The top 10 algorithms. *Computing in Science and Engg.*, IEEE Educational Activities Department, Piscataway, NJ, USA, v. 2, n. 1, p. 22–23, jan. 2000. ISSN 1521-9615. Citado na página 23.

ECKSTEIN, J. et al. Data-parallel implementations of dense simplex methods on the connection machine cm-2. *INFORMS Journal on Computing*, v. 7, n. 4, p. 402–416, 1995. Citado na página 67.

FLYNN, M. J. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, IEEE Computer Society, Washington, DC, USA, v. 21, n. 9, p. 948–960, set. 1972. ISSN 0018-9340. Citado na página 43.

GOLDBARG, M.; LUNA, H. *Otimização combinatória e programação linear: modelos e algoritmos*. [S.l.]: Campus, 2000. ISBN 9788535205411. Citado na página 23.

GRAMA, A. et al. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. [S.l.]: Addison-Wesley, 2003. ISBN 0201648652. Citado 2 vezes nas páginas 58 e 60.

GUSTAFSON, J. L. Reevaluating amdahl's law. *Commun. ACM*, ACM, New York, NY, USA, v. 31, n. 5, p. 532–533, maio 1988. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/42411.42415>. Citado 2 vezes nas páginas 65 e 93.

HALL, J. Towards a practical parallelisation of the simplex method. *Computational Management Science*, v. 7, n. 2, p. 139–170, 2010. Citado 2 vezes nas páginas 23 e 69.

HALL, J. A. J.; MCKINNON, K. Asynplex, an asynchronous parallel revised simplex algorithm. *APMOD95 Conference*, 1998. Citado na página 68.

KOCH, G. *Discovering multi-core: Extending the benefits of moore?s law*. [S.l.], 2005. Citado 3 vezes nas páginas 24, 55 e 69.

LIN, Y. C.; SNYDER, L. *Principles of Parallel Programming*. Boston, Mass: Pearson/Addison Wesley, 2008. ISBN 978-0321487902. Citado 4 vezes nas páginas 40, 51, 56 e 60.

LOWE, J. *INFORMS Impact Prize*. 2013. Disponível em: <https://www.informs.org/Recognize-Excellence/Award-Recipients/Janet-Lowe>. Citado na página 71.

MOORE, G. E. Cramming more components onto integrated circuits. *Electronics*, IEEE, v. 38, p. 114–117, abr. 1965. Citado 2 vezes nas páginas 24 e 54.

MORIMOTO, C. *O fim da lei de Moore*. 2012. Disponível em: <http://www.hardware.com.br/artigos/end-moore/>. Citado na página 55.

PACHECO, P. *An Introduction to Parallel Programming*. 1st. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN 9780123742605. Citado 7 vezes nas páginas 40, 41, 61, 62, 64, 65 e 66.

PASSOS, A. N. dos. *Estudos em Programação Linear*. Dissertação (Mestrado) — Universidade Estadual de Campinas, Campinas, SP, 2009. Citado na página 23.

ROSARIO, D. A. N. do. *Escalabilidade Paralela de um Algoritmo de Migração Reversa no Tempo (RTM) Pré-Empilhamento*. Dissertação (Mestrado) — Universidade Federal do Rio Grande do Norte, UFRN, Natal, RN, 2012. Citado 2 vezes nas páginas 57 e 64.

TANENBAUM, A. S.; WOODHUL, A. S. *Sistemas Operacionais: Projeto e Implementação*. [S.l.]: BookMan, 2000. Citado na página 56.

TASOULIS, D. K. et al. Parallel differential evolution. In: *In IEEE Congress on Evolutionary Computation (CEC*. [S.l.: s.n.], 2004. Citado na página 24.

THOMADAKIS, E. M.; LIU, J.-C. Progress and challenges in high performance computer technology. *Computer Science and Technology*, 2006. Citado na página 24.

THOMADAKIS, M. E.; LIU, J.-C. An efficient steepest-edge simplex algorithm for simd computers. In: *International Conference on Supercomputing*. [S.l.: s.n.], 1996. p. 286–293. Citado na página 68.

PLOSKAS, N.; SAMARAS, N.; SIFALERAS, A. (Ed.). *A parallel implementation of an exterior point algorithm for linear programming problems*. [S.l.]: University of Macedonia, 2009. Citado 3 vezes nas páginas 24, 67 e 70.

YARMISH, G.; SLYKE, R. V. *retroLP, An Implementation of the Standard Simplex Method*. [S.l.], 2003. Citado na página 69.

YARMISH, G.; SLYKE, R. V. A distributed, scaleable simplex method. *Supercomputing*, 2009. Citado na página 69.