UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
COORDENAÇÃO DE INFORMÁTICA
CURSO SUPERIOR DE TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

VICTOR HUGO SANTOS SCAPIN

AVALIAÇÃO DO DESEMPENHO DE BIBLIOTECAS PARA PARALELIZAÇÃO EM ARQUITETURAS MULTICORE: PTHREAD E OPENMP

TRABALHO DE CONCLUSÃO DE CURSO

CORNÉLIO PROCÓPIO

2013

VICTOR HUGO SANTOS SCAPIN

AVALIAÇÃO DO DESEMPENHO DE BIBLIOTECAS PARA PARALELIZAÇÃO EM ARQUITETURAS MULTICORE: PTHREAD E OPENMP

Monografia de trabalho de conclusão de curso apresentada ao Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas da Universidade Tecnológica Federal do Paraná – Campus Cornélio Procópio.

Orientador: Prof. Henrique Yoshikazu Shishido

CORNÉLIO PROCÓPIO

2013

TERMO DE APROVAÇÃO

VICTOR HUGO SANTOS SCAPIN

AVALIAÇÃO DO DESEMPENHO DE BIBLIOTECAS PARA PARALELIZAÇÃO EM ARQUITETURAS MULTICORE:
PTHREAD E OPENMP

Monografia aprovada como requisito parcial para obtenção do grau de Tecnólogo no Curso Tecnologia em Análise e Desenvolvimento de Sistemas, Universidade Tecnológica Federal do Paraná, pela seguinte banca examinadora:

Orientador: _____
Profª. Henrique Yoshikazu Shishido

_____
Profª. Francisco Pereira Junior

_____
Profª. Pedro Henrique Bugatti

CORNÉLIO PROCÓPIO

2013

DEDICATÓRIA

Agradeço a Deus por estar sempre ao meu lado em todos os momentos.

À minha família e minha namorada Luisa por todo o suporte dado durante esta caminhada.

Aos meus amigos pelo incentivo e apoio.

Ao pessoal da Igreja Cristã Shekinah pelas constantes orações.

Aos docentes da UTFPR por todo o aprendizado fornecido, em especial ao prof. Henrique Shishido pela orientação e tempo disposto auxiliando a conclusão deste trabalho.

EPÍGRAFE

“Light up the darkness (Ilumine a escuridão)” – Bob Marley

RESUMO

Esta monografia tem por foco ressaltar e avaliar duas bibliotecas de paralelização dos dias de hoje, *pThreads* e *OpenMP*. Ambas foram utilizadas para paralelizar um algoritmo sequencial de processamento de imagem digital. Com os avanços na área da tecnologia computacional, a acessibilidade à informação na Internet e de poder usufruir de máquinas com processadores *multicore* no conforto de sua casa ou onde estiver, tornou-se possível aplicar a computação paralela como um aliado para melhorar o desempenho de certas aplicações. Inserido nesse contexto, este trabalho avalia e analisa o desempenho das duas bibliotecas citadas anteriormente. O algoritmo utiliza o processo de convolução de *pixels* para ressaltar bordas e texturas, para analisar o índice de fragmentação da imagem processada. A execução da versão sequencial do algoritmo levou cerca de 43 minutos de processamento em um exemplo utilizando máscara de entrada de valor 15X15 *pixels* sobre uma imagem de 8460x9530 *pixels.* Em um contexto mais complexo, poderiam ser utilizadas dezenas ou centenas de imagens como esta, e o tempo gasto para processá-las poderia levar dias. Foram implementadas sete versões paralelas, entre as quais uma atingiu um resultado 6 vezes mais rápido comparado à versão sequencial.

Palavras-chave: bibliotecas de paralelização, OpenMP, pThread, computação paralela, convolução, processamento de imagem.

ABSTRACT

This monograph aims to highlight and to evaluate two of the most populars parallelization libraries nowadays, pThreads and OpenMP. Both of them were used to parallelize a digital image processing algorithm. With the advances in area of computational technology, acessibility to information on the internet and be able to use multicore machines in the confort of your home or wherever you are, it becames possible to apply parallel computing as an ally for improvement in processing time of certain applications. Inserted in this context, this work evaluate and analyse the performance of both libraries previously citted. The algorithm uses a pixel convolution process to enhance borders and textures patterns, to analyse the image fragmentation index of the processed image. The execution time of the serial version of the algorithm tooked about 43 minutes of processing in a example using a input mask value of 15X15 pixels over and 8460x9530 pixels image. In a more complex context, it could be used dozens or hundreds of images like these, and the time spent to process them could take days. Were implemented seven parallel versions, which one of them reached the result that was 6 times faster compared to the serial version processing time.

Keywords: parallelization libraries, OpenMP, pThread, parallel computing, convolution, image processing.

LISTA DE FIGURAS

FIGURA 4.1 – MODELO SISD 18

FIGURA 4.2 – MODELO SIMD 19

FIGURA 4.3 – MODELO MISD 19

FIGURA 4.4 – MODELO MIMD 20

FIGURA 4.5 – MODELO DE MEMÓRIA COMPARTILHADA DE ACESSO UNIFORME A MEMÓRIA 21

FIGURA 4.6 – MODELO DE MEMÓRIA COMPARTILHADA DE ACESSO NÃO UNIFORME A MEMÓRIA 21

FIGURA 4.7 – MODELO DE MEMÓRIA DISTRIBUÍDA 22

FIGURA 4.8 – MODELO DE MEMÓRIA COMPARTILHADA-DISTRIBUÍDA 23

FIGURA 8.1 – GRAFICO GERAL DOS TEMPOS DE EXECUÇÃO TODAS AS VERSÕES 46

FIGURA 8.2 – GRAFICO GERAL DOS VALORES DE SPEEDUP DE TODAS AS VERSÕES 48

FIGURA 8.3 – GRAFICO GERAL DOS VALORES DE EFICIÊNCIA DE TODA AS VERSÕES 49

LISTAGENS

LISTAGEM 7.1 – TRECHO DE CÓDIGO PARA SETAR NÚMERO DE THREADS A SEREM USADAS 37

LISTAGEM 7.2 – TRECHO DE PARALELIZAÇÃO DA VERSÃO 1 – OPENMP 38

LISTAGEM 7.3 – TRECHO DE PARALELIZAÇÃO DA VERSÃO 2 – OPENMP 39

LISTAGEM 7.4 – TRECHO DE PARALELIZAÇÃO DA VERSÃO 3 – OPENMP 40

LISTAGEM 7.5 – TRECHO DE PARALELIZAÇÃO DA VERSÃO 4 – OPENMP 41

LISTAGEM 7.6 – TRECHO DE PARALELIZAÇÃO DA VERSÃO 5 – OPENMP 43

LISTA DE TABELAS

TABELA 6.1 – TABELA DE VALORES DO ÍNDICE NA ESCALA DE CINZA.....35

TABELA 8.1 – TABELA GERAL DOS TEMPOS DE EXECUÇÃO DE TODAS AS VERSÕES.....44

TABELA 8.2 – TABELA GERAL DOS VALORES DE SPEEDUP DE TODAS AS VERSÕES.....47

TABELA 8.3 – TABELA GERAL DOS VALORES DE EFICIÊNCIA DE TODAS AS VERSÕES.....48

SUMÁRIO

1 INTRODUÇÃO 13

1.1 TEMA 13

1.2 ALGORITMOS DE PARALELIZAÇÃO 13

2 JUSTIFICATIVA 15

3 OBJETIVOS 16

3.1 OBJETIVO GERAL 16

3.2 OBJETIVOS ESPECÍFICOS 16

4 REVISÃO BIBLIOGRÁFICA 17

4.1 COMPUTAÇÃO PARALELA 17

4.2 ARQUITETURAS PARALELAS 18

4.3 ORGANIZAÇÕES DE MEMÓRIA EM ARQUITETURAS PARALELAS 21

4.4 MODELOS DE PROGRAMAÇÃO 23

4.4.1 MODELO DE MEMÓRIA COMPARTILHADA SEM THREADS 23

4.4.2 MODELO DE THREADS 24

4.4.3 MODELO DISTRIBUÍDO / MPI 25

4.4.4 MODELO DE DADOS PARALELOS 25

4.4.5 MODELO HÍBRIDO 26

4.5 POSIX THREADS 27

4.6 OPENMP 28

4.7 MÉTRICAS DE DESEMPENHO E EFICIÊNCIA 29

5 TRABALHOS RELACIONADOS 30

6 MATERIAIS E MÉTODOS 34

6.1 ALGORITMO 34

6.2 ESPECIFICAÇÕES TÉCNICAS, FERRAMENTAS E TECNOLOGIAS 35

7 MODELOS / IMPLEMENTAÇÕES PARALELAS 36

7.1 VERSÃO SEQUENCIAL 36

7.2 VERSÃO 1 - OPENMP 37

7.3 VERSÃO 2 - OPENMP 38
7.4 VERSÃO 3 - OPENMP 39
7.5 VERSÃO 4 - OPENMP 40
7.6 VERSÃO 5 - OPENMP 42
7.7 VERSÃO 1 - PTHREAD 43
7.8 VERSÃO 2 - PTHREAD 43
8 RESULTADOS E DISCUSSÃO 44
8.1 TEMPOS 44
8.2 SPEEDUP 47
8.3 EFICIÊNCIA 48
9 CONSIDERAÇÕES FINAIS 50
REFERÊNCIAS BIBLIOGRÁFICAS 51

1 INTRODUÇÃO

Tradicionalmente, *softwares* são escritos para utilizar processamento sequencial. As máquinas continham uma única unidade central de processamento (CPU), fazendo com que as instruções fossem executadas uma após a outra, individualmente. Com os avanços na área da computação, é possível utilizar uma máquina com múltiplas unidades centrais de processamento para usufruir do processamento paralelo.

O processamento paralelo é o uso simultâneo de múltiplos recursos computacionais para resolver um problema (Barney, 2012). Diferente das máquinas com uma única CPU, as que possuem múltiplas CPU ou núcleos podem dividir um problema grande em partes menores para executá-los paralelamente e/ou concorrentemente. Dessa maneira, é possível obter melhorias em tempo de processamento, sobrecarga em *hardware*, dividir problemas grandes para ser resolvido em partes, utilização de recursos externos, entre outros.

1.1 TEMA

Bibliotecas de memória compartilhada: *pThreads* e *OpenMP*.

1.2 ALGORITMOS DE PARALELIZAÇÃO

Nos dias atuais, a computação paralela deixou de ser exclusivamente relacionada à computação de alto desempenho em *mainframes* e supercomputadores, alcançando não somente as superestações de trabalhos, como também os computadores pessoais (PC). Os principais fabricantes de processadores para PCs, Intel e AMD, produzem atualmente somente processadores de múltiplos núcleos. Dessa forma, um programa escrito apropriadamente em uma máquina deste tipo pode dividir uma tarefa em partes, solucioná-las paralelamente, graças ao uso dos núcleos extras do processador, diminuindo, assim, o tempo necessário para sua execução.

A tarefa de escrever algoritmos de paralelização apresenta desafios, que são considerados mais difíceis que aqueles encontrados em algoritmos sequenciais. Em

geral, a paralelização é completamente tarefa do programador. Para auxiliar esse desenvolvimento, existem ferramentas e bibliotecas que contem recursos para o programador utilizar. Algoritmos paralelos são largamente utilizados para processamento de imagens digitais, processamento de gráficos computacionais, previsões meteorológicas em tempo real, tarefas de vídeo *encoding*, entre outras.

O presente trabalho está disposto em 9 capítulos. Esse capítulo descreve a sua introdução, tema e o conceito sobre algoritmos de paralelização. O capítulo dois explica a justificativa deste estudo. O capítulo três descreve os objetivos gerais e específicos. O capítulo quatro apresenta uma breve revisão bibliográfica referente ao tema desse trabalho. O capítulo cinco trata de trabalhos relacionados ao mesmo conteúdo proposto por este. O capítulo seis descreve os matérias e métodos utilizados. O capítulo sete apresenta os modelos implementados. O capítulo oito relata os resultados e discussão acerca dos modelos implementados, e por fim, o capítulo nove apresenta as considerações finais.

2 JUSTIFICATIVA

O presente trabalho justifica-se pelo interesse de realizar um estudo de análise e comparação entre duas bibliotecas de memória compartilhada: *pThreads* e *OpenMP*. Por meio desse estudo, será possível avaliar como um mesmo algoritmo de processamento de imagens digitais se comporta utilizando inicialmente uma biblioteca, e posteriormente a outra. Efetuado o estudo, foram feitos relatórios e conclusões acerca dos resultados obtidos.

A computação paralela tem recebido destaque em varias áreas da ciência, como publicado em (Tomov et al., 2010), que utiliza algoritmos envolvendo álgebra linear com aceleradores de unidades gráficas (GPU). Outros exemplos encontrados são o uso de scripts paralelos para aplicações na dimensão peta-escalar e além (Wilde et al, 2009), algoritmo de reconstrução de imagem 3D PET (Jones e Yao, 2004), programação paralela usando OpenMP em sistemas dual-core embarcados (Lee et al., 2011), métodos para desenvolvimento de algoritmos paralelos baseados em álgebra linear (Bosliga et al., 2011), simulador de crescimento de um tumor em um ser humano (Alias et al., 2008), entre outros.

Inserido nesse contexto, (Hillis, 1993) já previa uma transição para a computação paralela. Em seu artigo, ele declarou "*Here is how the future looks to me: By the end of the decade there will be few computers sold that do not act, at least part of the time, as parallel or distributed processors*[1]". Assim como estava certo sobre essa transição, acertou também quando descreveu que grandes estações de trabalhos distribuídas seriam responsáveis pelo armazenamento e fluxo de dados de grandes empresas.

A computação ainda passa por essa fase de transição. Atualmente já existem muitas máquinas com *hardware* compatível para a utilização paralela. O que falta na prática é a utilização e conhecimento para desenvolver *softwares* paralelos, e assim, melhorar desempenho, aproveitar mais o *hardware*, entre as outras vantagens da computação paralela.

[1] Assim é como o futuro parece para mim: até o fim da década haverá poucos computadores vendidos que não agem, pelo menos uma parte do tempo, em paralelo ou com processadores distribuídos.

3 OBJETIVOS

3.1 OBJETIVO GERAL

Constitui-se como objetivo geral desse trabalho analisar e comparar bibliotecas de memória compartilhada *pThreads* e *OpenMP*.

3.2 OBJETIVOS ESPECÍFICOS

Os objetivos específicos que norteiam esse estudo são:

- Realizar uma pesquisa sobre o estado da arte abordando a computação paralela;
- Conhecer os tipos de arquitetura paralela e modelos de programação;
- Implementar algoritmos paralelos utilizando as bibliotecas *pThreads* e *OpenMP*;
- Comparar os resultados;
- Escrita e submissão de um artigo para a área de computação paralela após a monografia.

4 REVISÃO BIBLIOGRÁFICA

Essa seção apresenta uma breve explanação sobre a computação paralela e seu estado da arte, tipos de arquitetura, tipos de programação paralela, tipos de bibliotecas que utilizam memória compartilhada e métricas usadas para medir desempenho.

4.1 COMPUTAÇÃO PARALELA

Historicamente, a computação paralela tem sido usada para modelar problemas em áreas da ciência e engenharia, como atividades na atmosfera, bioinformática, biotecnologia, genética, química molecular, sismologia, ciência da computação, matemática entre outros. Com o constante crescimento dessas áreas, as aplicações comerciais necessitam de um grande armazenamento e processamento dos dados.

Mineração de dados, exploração de petróleo, imagens e diagnósticos médicos, gerenciamento de corporações multinacionais e realidade virtual são alguns exemplos em que a informação tem que ser tratada de forma rápida e precisa. Há varias razões para o uso da computação paralela. Dentre elas, pode-se destacar: economizar tempo e/ou dinheiro, resolver problemas grandes (*Grand Challenge*), prover concorrência, utilização de recursos não locais, entre outros.

O modelo arquitetural proposto por John von Neumann, matemático húngaro que foi o pioneiro a escrever sobre os requerimentos gerais para um computador eletrônico, em suas anotações datadas com início em 1945, propõe que um computador é composto por quatro componentes principais: memória, unidade de controle, unidade lógica aritmética e dispositivos de E/S (entrada e saída). Este modelo caracteriza-se pelo processamento sequencial e, em uma execução de uma aplicação sequencial, as instruções executadas pelo processador são muito mais rápidas que as operações de E/S. Sendo assim, perde-se tempo pois no momento em que uma operação de escrita em disco está sendo executada, o processador fica ocioso esperando o término desta operação.

Para se ter uma aplicação na qual dois ou mais processos são executados de forma paralela e simultânea, são necessárias ao menos duas unidades de processamento. Com esse processamento adicional, apresentam-se algumas

complexidades como o controle de processos em execução, controle de acesso à memória e, dependendo do modelo paralelo, a comunicação entre os processos. Até os dias de hoje, praticamente todos os computadores utilizam o estilo proposto por von Neumann, a única diferença é que as máquinas mais modernas possuem múltiplas unidades de processamento, mas a arquitetura básica fundamental continua igual, mesmo após mais de sessenta anos.

4.2 ARQUITETURAS PARALELAS

Existem diferentes modos para classificar computadores paralelos. Um dos mais conhecidos e usados é a taxonomia de Flynn, proposto por Michael J. Flynn em 1996. (Flynn e Rudd, 1996) definiu quatro classificações baseadas nos números de fluxos de instruções concorrentes e de fluxos de dados disponíveis.

SISD (*Single Instruction, Single Data*) – um computador sequêncial, apenas um fluxo de instrução é ativado pela unidade de processamento em um determinado momento, assim como apenas um conjunto de dados é usado em um determinado momento. Apesar de ser o mais antigo, é ate hoje o mais comum tipo de computador.

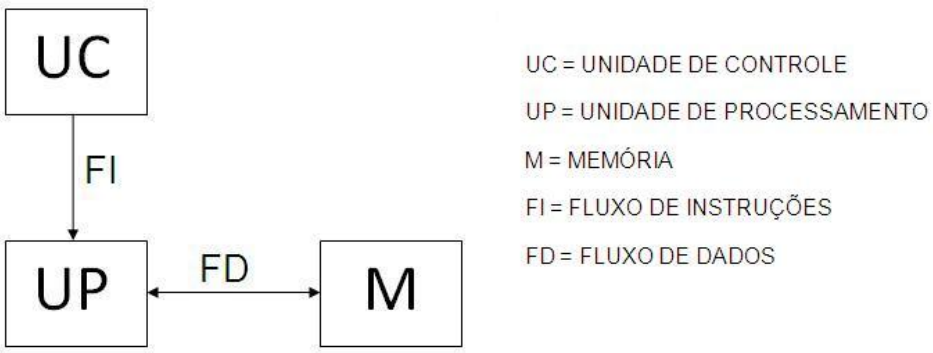

FIGURA 4.1 – MODELO SISD

SIMD (*Single Instruction, Multiple Data*) – um computador paralelo, apenas um fluxo de instruções é ativado pela unidade de processamento, porém esse fluxo pode ser operado em diferentes conjuntos de dados, sendo assim executada paralelamente. Um exemplo desse tipo de arquitetura pode ser encontrado em unidades de processamento gráfico (GPU).

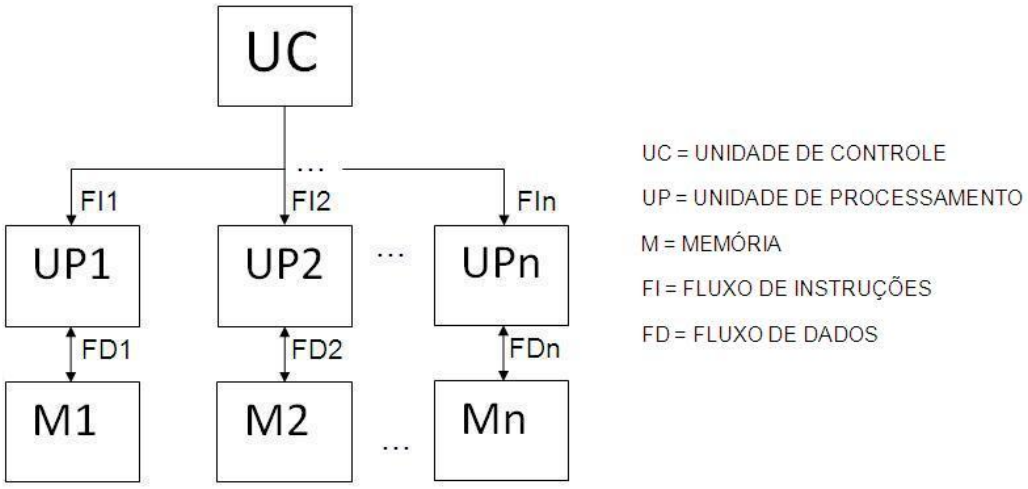

FIGURA 4.2 – MODELO SIMD

MISD (*Multiple Instruction, Single Data*) – um computador paralelo, múltiplos fluxos de instruções são enviados sobre um único conjunto de dados. É o tipo menos usado nos dias de hoje, apesar do fácil entendimento. Um exemplo do uso dessa arquitetura é uma aplicação onde múltiplos algoritmos de criptografia tentam quebrar uma única mensagem codificada.

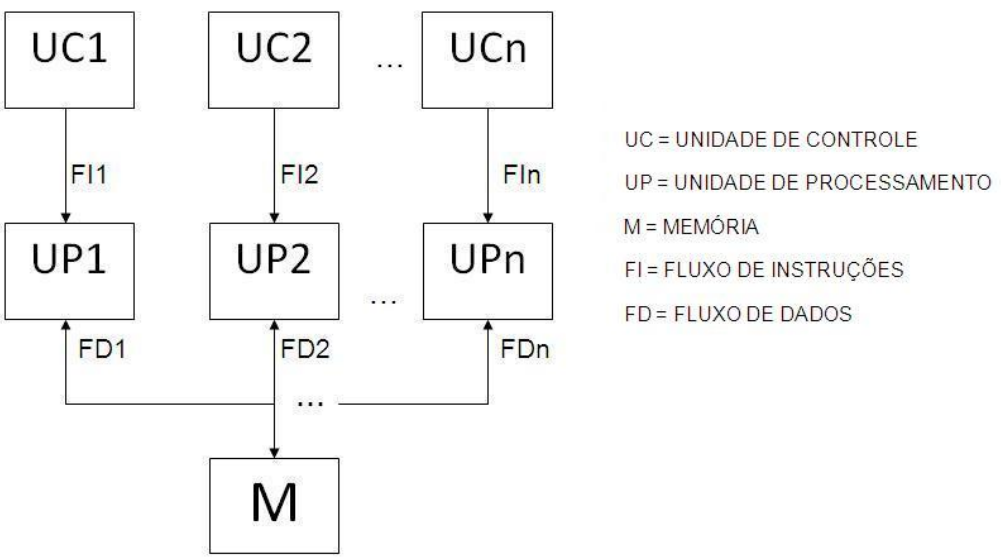

FIGURA 4.3 – MODELO MISD

MIMD (*Multiple Instruction, Multiple Data*) – um computador paralelo, múltiplos fluxos de instruções são enviados sobre múltiplos conjuntos de dados. Atualmente é

o tipo mais comum de computador paralelo. Os computadores com multiprocessadores são um exemplo que utiliza essa arquitetura.

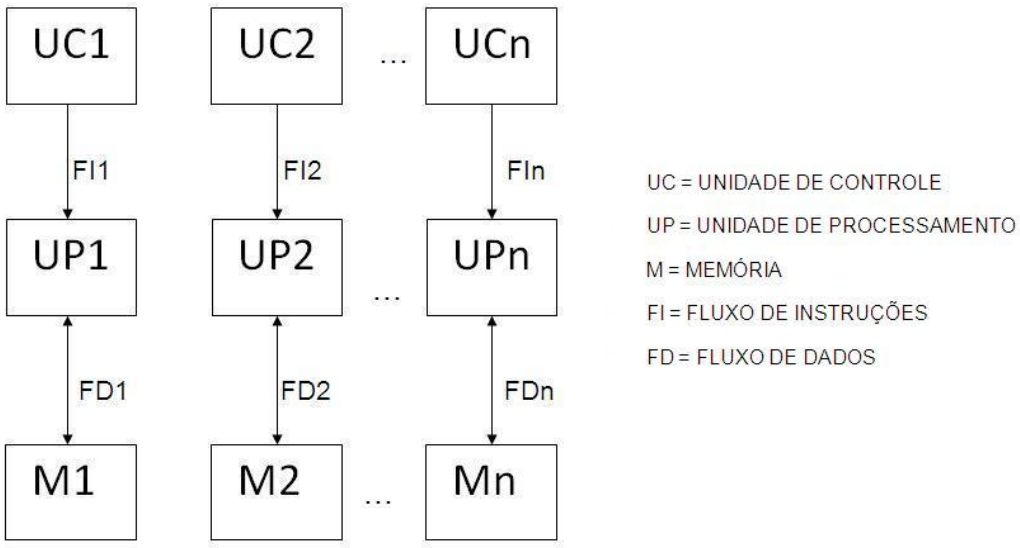

FIGURA 4.4 – MODELO MIMD

Em uma classificação mais recente do que a de (Flynn e Rudd, 1996), (Tanenbaum, 2007) divide a classe MIMD em dois tipos:

SPMD (*Single Program, Multiple Data)* – quando múltiplos processadores simultaneamente executam o mesmo programa (mas cada um em pontos diferentes) em diversos conjuntos de dados.

MPMD (*Multiple Program, Multiple Data*) – quando múltiplos processadores simultaneamente executam no mínimo dois programas independentes. Quando esse tipo de arquitetura é implementado, normalmente um dos processadores é configurado como “gerente”, que executa um programa que envia instruções para os demais processadores saberem o que devem executar em dado momento. Um exemplo utilizando essa arquitetura é o console de videogame da Sony, *Playstation 3*.

Existem também casos de máquinas específicas que não se encaixam em nenhum dos casos citados, dado a grande evolução das tecnologias ao decorrer dos anos. Baseado nas arquiteturas citadas anteriormente, são escolhidos os modelos de programação, que serão apresentados no capítulo 4.4.

4.3 ORGANIZAÇÕES DE MEMÓRIA EM ARQUITETURAS PARALELAS

Três tipos de arquiteturas de memória são mais comumente conhecidos e usados. São eles o modelo de memória compartilhada, memória distribuída, e o híbrido compartilhada-distribuída. Em cada um dos casos apresentam-se singularidades e usos específicos.

Memória compartilhada: tipicamente, um computador com esse tipo de arquitetura tem a habilidade de todos os processadores acessarem toda memória como um endereço global, o que torna a comunicação simples e implícita. Um sistema de memória compartilhada pode ser ainda dividido em dois tipos:

De acesso uniforme a memória: onde todos os processadores são idênticos e tem o mesmo acesso a memória.

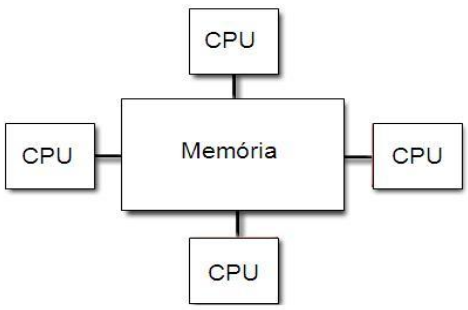

FIGURA 4.5 – MODELO DE MEMÓRIA COMPARTILHADA DE ACESSO UNIFORME A MEMÓRIA

De acesso não uniforme a memória: onde pelo menos um processador é diferente dos outros, nem todos os processadores tem o mesmo acesso a memória e o caminho de acesso à memória é mais lento comparado ao de acesso uniforme.

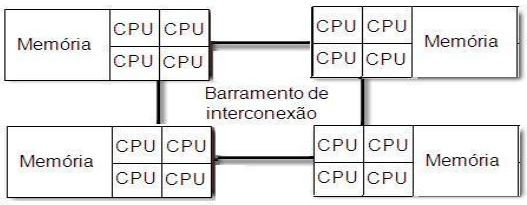

FIGURA 4.6 – MODELO DE MEMÓRIA COMPARTILHADA DE ACESSO NÃO UNIFORME A MEMÓRIA

As vantagens dessa arquitetura são a facilidade na programação, custo reduzido de desenvolvimento, simples, rápido e uniforme compartilhamento de dados graças à memória em comum e proximidade entre processador e memória. As desvantagens encontradas nessa arquitetura são que devido ao sistema estar todo presente no mesmo hardware, um problema interno ou mau funcionamento de uma das peças podem afetar o sistema todo. A parte de sincronização que garante o funcionamento e permite acesso à memória global de todos os processadores é responsabilidade do programador. Quanto mais processadores, mais difícil será o design da arquitetura.

Memória distribuída: um sistema que utiliza arquitetura de memória distribuída requer uma rede de comunicação para conectar respectivos processadores e memória. Os processadores tem sua própria memória local e opera independentemente. Quando um processador precisa acessar dados em outro processador, normalmente é tarefa do programador fazer essa ponte e definir como e onde os dados serão comunicados. Sincronização entre tarefas também é de responsabilidade do programador definir.

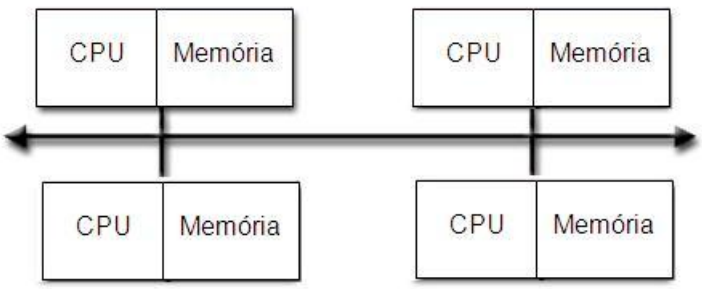

FIGURA 4.7 – MODELO DE MEMÓRIA DISTRIBUÍDA

As vantagens dessa arquitetura são que a memória é escalável com o número de processadores, pois se aumentar o número de processadores, aumenta a memória. Cada processador pode rapidamente acessar sua própria memória sem interferência e sem *overhead*, além de poder utilizar computadores extralocais e uma rede de comunicação. As desvantagens dessa arquitetura são que o programador é responsável por muitos dos detalhes associados com comunicação de dados entre processadores e pode ser difícil mapear estruturas existentes, comparado ao conceito de memória global.

Memória compartilhada-distribuída: os maiores e mais rápidos computadores do mundo hoje utilizam essa arquitetura, que é uma fusão do modelo de memória

compartilhada com o de memória distribuída. O componente da memória compartilhada pode ser uma máquina com múltiplos processadores (SMP) onde compartilham o mesmo endereço e/ou uma GPU. O componente de memória distribuída é a rede que conectam múltiplas máquinas SMP/GPU, que compartilham somente sua memória.

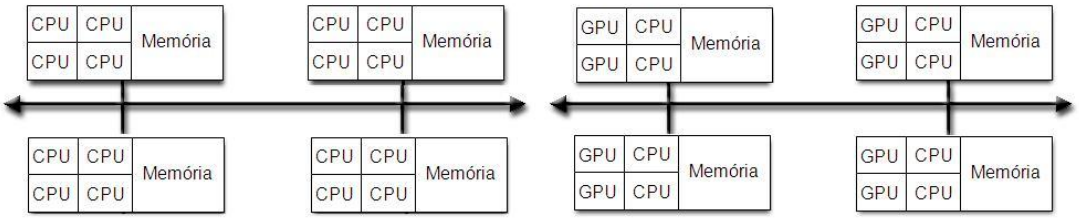

FIGURA 4.8 – MODELO DE MEMÓRIA COMPARTILHADA-DISTRIBUÍDA

As vantagens e desvantagens dessa arquitetura são tudo o que o modelo de memória compartilhada e o modelo de memória distribuída têm em comum.

4.4 MODELOS DE PROGRAMAÇÃO

Existem vários modelos de programação para sistemas paralelos. Serão explanados aqui os modelos: memória compartilhada (sem *threads*), threads, memória distribuída com *message passing interface* (MPI), modelo de dados paralelos (*data parallel*) e modelo híbrido. Apesar dos modelos serem semelhantes, eles não são específicos para um tipo de máquina ou de arquitetura de memória. Como é comum na computação, não existe um modelo melhor que o outro, existem modelos que podem ser mais apropriados em certas circunstâncias.

4.4.1 MODELO DE MEMÓRIA COMPARTILHADA SEM THREADS

Nesse modelo, as tarefas dividem um espaço de endereço comum, onde são lidas e escritas assíncronamente. Para controlar o acesso a memória, mecanismos como *locks* e semáforos podem ser usados. Talvez a principal vantagem desse

modelo consiste no ponto de vista do programador, que para ele não existe noção de “dono” da memória. Com isso, não precisa especificar explicitamente a comunicação de dados entre as tarefas.

Quando se fala em computação paralela, consequentemente se pensa em alta performance. Uma desvantagem desse modelo em relação à performance é que se torna mais difícil entender e gerenciar a localidade dos dados. Quando um fluxo de dados se torna exclusivo de um processador, conserva-se acesso a memória, atualização de *cache*, e tráfego entre os comunicadores *bus.* Quando esse fluxo não se torna exclusivo, pode-se perder performance.

4.4.2 MODELO DE THREADS

Esse é um modelo mais utilizado em arquiteturas de memória compartilhada. Quando trabalha com *threads*, um único processo pode ter múltiplos caminhos de execução concorrentes. Para descrever o modelo de *threads*, de forma análoga, podemos imaginar um programa A, que é agendado para ser executado pelo sistema operacional. Esse programa A carrega e obtêm dados necessários do sistema para iniciar. Depois disso, o programa A executa uma série de trabalhos e cria um número de tarefas (*threads*) que podem ser agendadas para serem executadas pelo sistema operacional concorrentemente.

Cada uma das *threads* possuem dados locais, mas também dividem uma série de recursos de A. Com isso, livra-se do *overhead* causado com a replicação de recursos de programação para cada *thread*. Cada *thread* também se beneficia de uma vista global da memória, pois compartilha o espaço de memória de A. O trabalho de uma *thread* pode ser descrito como uma sub-rotina dentro do programa A.

Qualquer *thread* pode executar qualquer sub-rotina a qualquer momento. Elas se comunicam através da memória global. Isso requer estruturas de sincronização para garantir que mais de uma *thread* não esteja atualizando o mesmo endereço na memória global ao mesmo tempo. As *threads* podem iniciar e finalizar, mas o programa A continua presente para fornecer os recursos compartilhados até o fim da aplicação.

Para implementar o modelo de *threads*, geralmente se usam elementos como bibliotecas de sub-rotinas que são chamadas de dentro do código fonte, uma série

de diretivas de compilação fixas em códigos fontes paralelos ou sequenciais. Em qualquer um dos casos, o programador é responsável para determinar o paralelismo.

Esse tipo de implementação de *threads* não é algo novo na computação. Antigamente os desenvolvedores de *hardware* implementavam sua própria versão de *threads*, o que tornava *thread* um recurso sem um padrão deferido. Esforços não relacionados para criar padrões resultaram em dois modelos de implementação bem diferentes: POSIX *Threads* (*pThreads*) e *OpenMP* que serão discutidos posteriormente.

4.4.3 MODELO DISTRIBUÍDO / MPI

Esse modelo consiste em uma série de tarefas que utilizam a própria memória local. Diversas tarefas podem residir na mesma máquina física e/ou entre um número arbitrário de máquinas. As tarefas trocam dados entre comunicações enviando e recebendo mensagens. A troca de dados normalmente requer operações cooperativas para serem executadas por cada processador. Por exemplo, uma operação de envio deve combinar com uma operação de recebimento.

Para implementar esse modelo, MPI normalmente utiliza uma biblioteca de sub-rotinas. Chamadas dessas sub-rotinas são colocadas no código fonte e o programador é responsável por todo o paralelismo. Desde a década de 1980, existem várias bibliotecas baseadas em passagens de mensagens. Essas bibliotecas diferem-se substancialmente, tornando difícil o desenvolvimento de aplicações portáveis. O MPI1 foi lançado em 1994, e em 1996 o MPI2. O MPI é atualmente a biblioteca padrão baseada em passagem de mensagens, substituindo virtualmente todas as outras implementações de passagem de mensagem no ramo comercial.

4.4.4 MODELO DE DADOS PARALELOS

A maioria do trabalho paralelo nesse modelo é focado em fazer operações sobre um conjunto de dados. O conjunto de dados geralmente é organizado em um tipo de estrutura de dados comum, como um vetor ou um cubo. Um conjunto de

tarefas trabalha coletivamente na mesma estrutura de dados, porém em partes diferentes da mesma. Em arquiteturas de memória compartilhada, todas as tarefas devem ter acesso à estrutura de dados através da memória global. Em arquiteturas de memória distribuída, a estrutura de dados é dividida e armazenada na memória local de cada tarefa.

Programar usando esse modelo é normalmente feito escrevendo um programa com construtores de dados paralelos. Os construtores podem ser chamados através de uma sub-rotina paralela ou por diretivas de compilação reconhecidas por um compilador de dados paralelos. Três formas conhecidas para programar usando esse modelo são: utilizando Fortran 90 e 95, *High Performance Fortran* (HPF) e por diretivas de compilação. Arquiteturas de memória distribuída que utilizam esse modelo geralmente requerem que o compilador produza um *object code* (resultado da compilação) com chamadas para uma biblioteca de passagem de mensagens (tradicionalmente a MPI) para distribuição de dados. Essa passagem é feita de forma invisível para o programador.

4.4.5 MODELO HÍBRIDO

Como o próprio nome já diz, o modelo híbrido combina mais de um dos modelos descritos anteriormente. Um exemplo de um modelo híbrido é a combinação do MPI com o modelo utilizando *threads* (OpenMPI). As *threads* fazem *kernels* computacionais intensivos utilizando dados locais do seu nó, e a comunicação entre processos em diferentes nós ocorre pela rede responsável utilizando MPI. Esse modelo se destaca graças ao crescimento no ambiente de hardwares com os *clusters* utilizando máquinas multicores.

Outro exemplo popular desse modelo é utilizando MPI com GPU. Onde as GPUs fazem o mesmo papel das *threads* no exemplo citado anteriormente, e o modelo MPI é responsável pela comunicação entre os diferentes nós através da rede.

4.5 POSIX THREADS

Como citado anteriormente, antigamente quando os desenvolvedores escreviam suas próprias versões para threads, a falta de padrão tornou aplicações com thread um mercado complexo, pois dificultava o desenvolvimento de aplicações portáveis que utilizavam threads. Para obter o máximo das vantagens de utilizar threads, um padrão precisava ser criado. Para sistemas UNIX, o padrão POSIX Threads foi especificado pela IEEE POSIX 1003.1c em 1995. Implementações de threads que aderem a esse padrão são chamadas de POSIX threads ou *pThread*.

A *pThread* é definida como uma série de conceitos da linguagem C como tipos de dados, funções e chamadas de *procedures.* É implementada usando pthread.h no cabeçalho do código e uma biblioteca que utiliza thread. Através dessa biblioteca que utiliza thread, é possível ter partes de outras bibliotecas como libc, dependendo da aplicação.

Um dos principais motivos para utilizar *pThread* é o ganho computacional. Comparado com o custo de criar e gerenciar um processo, uma *thread* pode ser criada com menos operações do sistema. Ou seja, quando gerencia uma *thread*, são gastos menos recursos operacionais do que gerenciando um processo. Toda *thread* dentro de um processo divide o mesmo espaço em memória. Essa intercomunicação entre as *threads* é mais eficiente e na maioria das vezes mais fácil de se fazer do que comunicação entre processos.

Aplicações utilizando *threads* oferecem otimização em várias outras maneiras além de redução de tempo comparado com as que não utilizam. Com *threads,* é possível definir prioridades para certas tarefas sobre outras, podendo interromper ou substituir tarefas com menos prioridade, além de outros recursos como tratamento de eventos assíncronos e *overlapping* de trabalho computacional. Aplicações com as seguintes características são bem indicadas para o uso da *pThread*: usar muitos ciclos do CPU em certos pontos e não em outros, bloqueios em tarefas que esperam longas instruções de E/S, tarefas mais importantes que outras e aplicações que devem responder eventos assíncronos.

4.6 OPENMP

No início dos anos 1990, os vendedores de máquinas com memória compartilhada utilizavam em sua maioria programas em Fortran. Dessa maneira, o usuário tinha que inserir programas sequenciais em Fortran com instruções especificando em quais lugares (*loops*) deviam ser paralelizados. Assim o compilador seria responsável por paralelizar tais loops automaticamente entre os multiprocessadores da máquina. A especificação para o padrão *OpenMP* foi iniciado em 1997, substituindo o padrão ANSI X3H5, quando máquinas com arquitetura de memória compartilhada começaram a se tornar mais popular.

OpenMP (*Open Multi-Processing*) é uma interface entre aplicativo e programação (API) que pode ser usada para paralelismo de aplicações de memória compartilhada que utilizam *multi-thread*. É composta por três componentes: diretivas de compilação, biblioteca de rotinas e variáveis de ambiente. Sua portabilidade é uma de suas maiores vantagens, podendo utilizar programas em linguagem C, C++ e Fortran (77, 90 e 95), na maioria dos sistemas operacionais, como Linux, Unix, AIX, Solaris, OS X e Windows.

Os objetivos da *OpenMP*, além da padronização de um modelo para arquiteturas de memória compartilhada, era reduzir o número de diretivas para desenvolver o paralelismo, podendo fazer grandes mudanças com apenas 3 ou 4 diretivas. Um programa com *OpenMP* é baseado na existência de múltiplas threads, pois um processo em memória compartilhada consiste em várias threads. O modelo de programação é de maneira explícita, porem não automática, oferecendo ao programador total controle perante a paralelização.

Toda aplicação em *OpenMP* utiliza o modelo *fork-join*, e todos os programas começam com um único processo, a thread principal. Ela executa sequencialmente até a primeira estrutura de paralelismo ser encontrada. A thread principal cria um conjunto de threads paralelas, assim tudo que estiver dentro de uma região demarcada por uma estrutura paralela será executada em paralelo juntamente com os outros conjuntos de threads. Quando o conjunto de threads termina de executar a região da estrutura paralela, elas sincronizam e termina-se a execução, deixando apenas a thread principal.

Grande parte do paralelismo da *OpenMP* é especificado através do uso de diretivas de compilação que são incluídas no código fonte em C, C++ ou Fortran. A API oferece a possibilidade de usar trechos de paralelização dentro de outro trecho

de paralelização, dependendo apenas da aplicação para ver se é capaz de suportar esse recurso. A API fornece também para o ambiente em tempo real a escolha de alterar o número de *threads* usadas para executar certo trecho de códigos paralelos.

4.7 MÉTRICAS DE DESEMPENHO E EFICIÊNCIA

Computação paralela é frequentemente associada à melhora de desempenho (performance). Performance é um termo utilizado na computação para se referir ao tempo total de execução de um programa. Quanto menor o tempo, melhor é a performance. Utilizando esse tempo, é possível aplicar algumas métricas conhecidas para avaliar essa performance. Entre essas métricas podemos citar o *speedup* e eficiência.

O *speedup* é uma equação que possibilita calcular a razão entre o tempo de execução de um programa sequencial pelo tempo de execução do mesmo programa em paralelo. A fórmula para esse cálculo é:

speedup = desempenhoSequencial / desempenhoParalelo

EQUAÇÃO 1 – CÁLCULO DE SPEEDUP.

Eficiência proporciona uma indicação da utilização efetiva de processadores. Tendo o valor do *speedup*, é possível calcular a eficiência dado o número de processadores. O cálculo fica representado pela seguinte equação:

eficiência = *speedup* / nº processadores

EQUAÇÃO 2 – CÁLCULO DE EFICIÊNCIA.

5 TRABALHOS RELACIONADOS

O avanço nas pesquisas científicas se dá, muitas vezes, graças ao avanço da tecnologia. Um exemplo concreto é o que aconteceu no Centro de Oncologia Molecular do Hospital Sírio-Libanês, onde em 2012 foi investido recursos para adquirir novas máquinas de sequenciamento para acelerar a customização do diagnóstico e do tratamento de câncer, usando características genéticas dos tumores para selecionar drogas e técnicas mais eficientes. Como essa área está em constante expansão, diversas aplicações relacionadas ao tema do presente trabalho tem sido desenvolvidas. Abaixo contém alguns dos trabalhos que se enquadram nesse aspecto.

(Zhong et al., 2007) propôs a implementação de um estudo onde utiliza-se *OpenMP* e *pThread* para algoritmos paralelos de predição de estruturas de proteínas secundárias. Foram implementados classificadores terciários para predição de proteínas secundárias sobre a arquitetura *Denoeux Belief Neural Network* (DBNN), incluindo matrizes do tipo hidrofobicidade, ortogonais, BLOSUM62 e PSMM. Os resultados experimentais contribuíram para a criação de novos esquemas de codificação. A precisão dos classificadores terciários com os esquemas de codificação usando PSMM chegaram a 72%, que é 10% mais preciso do que os resultados previamente obtidos.

Ambas *pThread* e *OpenMP* foram utilizadas em (Zhong et al., 2007), e as duas foram encarregadas de paralelizar sobre a DBNN usando *Hyper-Threading.* O valor do *speedup* para 16 *pthreads* foi de 4.9 e para 16 *OpenMP threads* foi de 4 nos 4 processadores. A *pThread* foi um pouco superior comparado a *OpenMP* nesse trabalho. Graças a este algoritmo paralelizado, milhares de aminoácidos podem ser processados em um tempo eficiente para a área. Além disso, o trabalho também mostrou que a arquitetura *Hyper-Threading* da Intel é efetiva para algoritmos paralelos na área da biologia.

Em (Tarmyshov e Müller-Plathe, 2005) foi feito uma paralelização de um algoritmo molecular dinâmico usando *OpenMP*. O algoritmo conhecido pelo nome de YASP foi paralelizado restritamente nas áreas em que mais se consumia tempo de CPU. Boa parte do código sequencial em FORTRAN foi mantido e os construtores paralelos criados foram feitos através dos padrões de diretivas de compiladores da

OpenMP. Em uma máquina com 16 processadores Xeon *dual*, o valor do *speedup* foi de 1.7.

(Jang, Park e Jung, 2008) propôs uma implementação de uma rede neural utilizando CUDA (plataforma de computação paralela de propósito geral que tira proveito das unidades de processamento gráfico (GPU) da NVIDIA) e OpenMP. Em um algoritmo de processamento de imagens e reconhecimento de padrões, o pesquisador conseguiu um resultado quinze vezes mais rápido utilizando CUDA e OpenMP em relação à utilizando somente CPU e quatro vezes mais rápido usando CUDA e OpenMP em relação à utilizando somente GPU.

Podemos ver também o uso de *pThreads* com MPI em (Pfeiffer e Stamatakis, 2010), onde foi implementado uma paralelização híbrida sobre o código filogenético RAxML, onde trecho de código do MPI foi adicionado ao já existente *pThread*. Essa versão híbrida do RAxML é especialmente útil para uma análise filogenética e execução de inicializações rápidas seguidas por uma pesquisa completa de probabilidade.

O código híbrido também permite uma utilização mais eficiente dos processadores dado o número de núcleos. Em um dos problemas utilizados, o valor de *speedup* para a versão híbrida foi de 35 comparado a versão sequencial e de 6.5 comparado a versão que anteriormente só usava *pThread* (sem MPI). Em outro problema com outros dados de entrada, o *speedup* foi de 38 com dois nós (64 núcleos) comparado à versão sequencial.

Outro estudo que foi feito (Curtis-Maury et al., 2008) testa a eficiência da biblioteca OpenMP em processadores baseados em simultâneas *multi-threads* e (SMT) processadores *multicores* (CMP). Foi notado que devido ao alto nível de recursos compartilhados nos SMT's resultaram em problemas na performance. Já nos CMP, foi notado que a exploração dos múltiplos núcleos de cada processador resultaram em ganhos significativos. Foi concluído então que, para esta pesquisa, código usando *OpenMP* é mais escalável em CMP's do que em SMT's. Foi destacado também que para maximizar a eficiência da *OpenMP* em processadores SMT's, novas técnicas em tempo de execução precisam se tornar acessíveis.

Em um trabalho mais recente (Babu, Krishna e Khalid, 2011), foi feito um estudo divulgando as técnicas de otimização existentes para evolução de

performance usando *OpenMP* em uma arquitetura *multicore* usando *multithreads*. Foi ressaltado que existe uma variedade de técnicas de otimização de programação, porém é difícil de se achar um histórico detalhando a sequência em que as técnicas devem ser aplicadas para obter máxima performance. Para concluir, os autores fizeram um passo a passo de o que fazer primeiro até chegar no ponto em que o código está otimizado.

(Weng, Perng e Chapman, 2007) propôs um estudo onde implementaram a biblioteca *OpenMP* para um programa de simulação de circuito chamado SPICE3. Foram desenvolvidas dois tipos de códigos, um com pouca mudança no código e uma sem modificação da original usando o construtor *taskq* da Intel. No experimento, utilizando um modelo de simulação SRAM, foi compilado usando o compilador SUN rodando em um SunFire V880 UltraSPARC-III 750 MHz e pelo compilador Intel icc rodando em um IBM Itanium com quatro CPU's e um Xeon de dois processadores.

(Murao, 2006) desenvolveu um experimento envolvendo *multithreads* e álgebra computacional usando *OpenMP*. Os esforços foram para aplicar sobre o OpenXM um menor custo de desenvolvimento possível, mas com uma margem razoável de ganho de desempenho. Foi incluído também gerenciamento de memória no ambiente de *multithreads*. Os resultados empíricos sugerem que o ganho de performance pode ser obtido em diversos casos estudados.

Em (Rufai et al., 2004), os autores destacaram a facilidade em que se pode obter um bom desempenho de paralelismo com pouco custo de programação. Em seu estudo, usaram *OpenMP* junto com linguagem C em um simples programa multiplicador de matrizes em que o tempo de execução paralelo foi consideravelmente melhor do que sequencial. Além da facilidade de programação, ressaltaram também a facilidade de debugar e otimizar usando as funcionalidades da *OpenMP*.

Outro estudo feito em 2011 (Deepak Shekhar et al., 2011) propõe a comparação de três dos mais comuns modelos de paralelização, *OpenMP*, *pThreads* e GCD. Foi ressaltado que uma máquina ter múltiplos processadores com múltiplos núcleos não significa que o desempenho será na mesma proporção. Boa parte da responsabilidade está nos desenvolvedores em saber usar as funcionalidades disponíveis, dado que aplicações *multicore* ou *multithread* requerem

mais trabalho e são mais complexas de se trabalhar comparando com aplicações *monocore* ou *singlethread.*

Temos também um trabalho em que foi implementado o algoritmo KMP baseado em *OpenMP* e MPI (Duan et al., 2012). O algoritmo KMP é um algoritmo utilizado para verificar *strings* compatíveis. Utilizando este modelo hibrido, o algoritmo foi melhorado para a verificação dessas palavras chaves, conseguindo executar mais rapidamente os resultados encontrados. Além de ser mais rápido, conseguiu-se um aproveitamento maior da memória da estação de trabalho e de menos tempo de inter-comunicação de taferas comparado à MPI puro.

Um ótimo exemplo em que ambas as bibliotecas utilizadas no presente trabalho foram usadas, são os estudos (Lee e Downar, 2001) e (Stamatakis e Ott, 2008), e em cada uma delas uma biblioteca levou vantagem sobre a outra. No primeiro, foram usadas *OpenMP* e *pThread* para implementar um código de analise transiente de um reator nuclear chamado PARCS, rodando em duas estações de trabalho SUN e SGI. Considerando o tempo de execução e o tempo requerido para comparar, a versão em *OpenMP* foi a melhor escolha para programação paralela em arquiteturas de memória compartilhada.

Já no segundo trabalho, em que foi desenvolvido um algoritmo de cálculos de probabilidade filogenética na aplicação RAxML, largamente usado na bioinformática. Foi feito o trabalho utilizando *OpenMP*, *pThreads* e MPI sobre uma variedade de plataformas arquiteturais diferentes. Os resultados indicaram que não existe um modelo universal melhor do que o outro, porém, os autores, ao final da pesquisa, recomendaram que para a aplicação RAxML é mais vantajoso utilizar MPI e *pThreads* baseado em um critério de engenharia de software.

6 MATERIAIS E MÉTODOS

Esta seção contém o material utilizado para o desenvolvimento deste estudo, entre eles especificações do sistema (*hardware*), ferramentas e tecnologias utilizadas e o objeto de estudo deste trabalho, o algoritmo.

6.1 ALGORITMO

O algoritmo cujo trabalho foi desenvolvido foi abordado em (Shishido, 2010), utilizado originalmente em sua dissertação de mestrado. Tal algoritmo foi escrito para processamento de imagem digital, especificamente em imagens geográficas. Ele utiliza uma convolução de *pixels*, que realça bordas e padrões de textura, permitindo analisar o índice de fragmentação da imagem.

Esta convolução de *pixels* consiste em atuar sobre um ponto (*x,y*) da imagem por meio de uma matriz aplicada sobre todos os *pixels* de uma imagem inicial para produzir uma imagem final. Técnicas de convolução podem fazer uso de diferentes algoritmos como, soma ponderada dos *pixels* vizinhos, mediana dos *pixels*, entre outros.

Na versão implementada neste trabalho, cada *pixel* da imagem é percorrido e considera uma máscara (matriz) quadrada de *pixels* vizinhos ao redor do *pixel* atual, para determinar o índice de fragmentação. O valor da máscara é informado pelo usuário através de parâmetros na hora de execução. Quanto maior o valor dessa máscara, mais tempo de processamento irá gastar para cada *pixel* analisado na imagem.

Quando a imagem é processada, dado valor de um *pixel* em uma posição (*x,y*), a matriz é formada de acordo com a máscara formando um quadrado com os pixels vizinhos. Dentro desta máscara, cada *pixel* é analisado e é obtido o valor do mesmo na escala de cinza 0-255. Dependendo do valor na escala, um outro valor chamado de índice é atribuído para cada *pixel,* obedecendo a seguinte tabela:

VALOR	ÍNDICE
0-28	0
29-57	1
58-86	2
87-115	3
116-144	4
145-173	5
174-202	6
203-231	7
232-255	8

Tabela 6.1 Tabela de valores do índice na escala de cinza.

Se o valor na escala de cinza estiver entre 0 e 28, o índice do *pixel* tem valor igual a 0. Se o valor estiver entre 29 e 57, o valor do índice é igual a 1, assim sucessivamente. Depois de determinar o índice para cada *pixel,* é determinado o número de índices distintos na máscara e o percentual do *pixel* central, para gerar o valor a ser multiplicado por 255 para geração da nova imagem.

6.2 ESPECIFICAÇÕES TÉCNICAS, FERRAMENTAS E TECNOLOGIAS

Todo o trabalho foi desenvolvido utilizando a mesma estação de trabalho, ou seja, os resultados obtidos foram feitos na mesma máquina. As configurações são: Sistema operacional Mac OS X dual boot com Linux Ubuntu 10.10, processador Core i7 2.4ghz, memória RAM 10GB DDR3, 750GB de HDD. Os resultados foram feitos rodando no Linux Ubuntu 10.10, para simular o mesmo ambiente de trabalho que o autor da versão pThread (Shishido, 2010) na época do estudo.

As ferramentas e tecnologias utilizadas foram: biblioteca de processamento de imagens OpenCV v1.0, gerenciador de perfil de execução Gprof, linguagem C para o desenvolvimento e compilador GNU via terminal.

7 MODELOS / IMPLEMENTAÇÕES PARALELAS

Este capítulo contém os modelos paralelos criados utilizando a biblioteca *OpenMP*, as versões em *pThread* e a sequencial. Todas as versões utilizam um procedimento de compilação similar, executando alguns comandos via terminal no Linux. Para executar, por exemplo, a versão *OpenMP* – V.1, usa-se os seguintes comandos :

```
1 - cd /home/victor/TCC/OpenMP/Versao1
2 - gcc -pg -o cv cv_versao1.c -I/usr/include/opencv -lcv -lhighgui -
fopenmp
3 - time ./cv 0 15 cbersgde.tif
4 - gprof ./cv gmon.out  > output.txt
```

A primeira linha descreve o comando para acessar o diretório em que se encontra o arquivo C. A segunda linha descreve o comando de compilação deste arquivo C, passando o nome do arquivo, a biblioteca *OpenCV* de processamento de imagem e o parâmetro de uso da biblioteca *OpenMP*. Na terceira linha usa-se o comando “time” do Linux para computação do tempo que foi gasto para executar o código. O parâmetro “0” informa que o *debug* será desativado.

O parâmetro “15” é o tamanho da máscara matricial que será usada para percorrer os *pixels* e, em seguida, passa-se o nome da imagem. Por fim na quarta linha é executado o comando para gerar o *profile* da execução anterior. É gerado um arquivo texto pela aplicação *Gprof* no mesmo diretório em que se encontra o arquivo C, que contém detalhadamente quanto tempo levou para percorrer cada função dentro do código.

7.1 Versão Sequencial

A versão sequencial não possui nenhum trecho de paralelização de código. Para executá-la, segue os passos via terminal no Linux:

```
1 - cd /home/victor/TCC/Sequencial
2 - gcc -pg -o cv cv_sequencial.c -I/usr/include/opencv -lcv -lhighgui -fopenmp
3 - time ./cv 0 15 cbersgde.tif
4 - gprof ./cv gmon.out  > output.txt
```

7.2 Versão 1 – OpenMP

Para todas as versões paralelas com *OpenMP,* foi adicionado uma linha de código na função *main* para dizer quantas *threads* serão utilizadas, sendo o valor N o número inteiro respectivo ao número de *threads*, ficando da seguinte maneira (linha número 2):

```
1.   int main (int args, char *argv[]) {
2.    omp_set_num_threads(N);
3.    IF ( (args >=4) && ((atoi(argv[1]) == 0) || (atoi(argv[1]) == 1)) &&
((atoi(argv[2]) > 2) && (atoi(argv[2]) < 50))) {
4.          char *imagemArquivo = argv[3];
5.          DADOS_PROCESSADOS = atoi(argv[1)];
6.          TAM_MASCARA = atoi(argv[2]);
7.          abrirArquivo();
8.          carregarImagem (imagemArquivo);
9.          dividirIntervalo();
10.         convolucao();
11.         fecharArquivo();
13.   } else {
14.         printf(“\nSintaxe Incorreta”);
15.   }
16. }
```

Listagem 7.1 Trecho de código para setar número de threads a serem usadas.

A versão 1 utiliza a biblioteca *OpenMP*. Foi adicionado trecho de paralelização na função “substituirNDintervalo”, nas linhas de número 9 e 16. A linha 9 serve para indicar que o laço de repetição executado posteriormente será paralelizado. A linha 16 explicita que ocorrerá uma sincronização entre as *threads* e aguardarão até que todas se encontrem naquele ponto.

```
1.  float substituirNDporIntervalo( int lin, int col ) {
2.    int mm[MAXIMUM][MAXIMUM];
3.    int dividendo = -1;
4.    float novoPixel = -1;
5.    int intervalo = -1;
6.    int distintos = -1;
7.    CvScalar pixelIndex;
8.    int i, j, k;
9.    #pragma omp parallel for
10.   for (i=0; i< TAM_MASCARA; i++) {
11.         for (j=0; j<TAM_MASCARA; j++){
12.               intervalo = getIndiceIntervalo(matrizImagem[lin+i][col+j]);
13.               mm[i][j] = intervalo;
14.         }
15.   }
16.   #pragma omp barrier
17.   distintos = contarIntervalosDistintos(mm);
```

```
18.    dividendo = distintos - 1;
19.    if( dividendo < 0 )
20.          printf("\nDividendo negativo: lin,col -> %d,%d", lin, col);
21.    novoPixel = (float) ( (dividendo*1.0) / ((TAM_MASCARA * TAM_MASCARA) -1 ) );
22.    if( novoPixel < 0 )
23.          printf("\nNovo Pixel é negativo!");
24.    return novoPixel;
25. }
```

Listagem 7.2 Trecho de paralelização da versão 1 - OpenMP.

Para executar esta versão, segue os comandos via terminal no Linux:

```
1 - cd /home/victor/TCC/OpenMP/Versao1
2 - gcc -pg -o cv cv_versao1.c -I/usr/include/opencv -lcv -lhighgui -
fopenmp
3 - time ./cv 0 15 cbersgde.tif
4 - gprof ./cv gmon.out  > output.txt
```

7.3 Versão 2 – OpenMP

A versão 2 utiliza a biblioteca *OpenMP*. Foi adicionado trecho de paralelização na função “convolucao”, na linha número 28 para executar um laço de repetição padrão (*for*) porém usando múltiplas threads de forma automática.

```
1.  void convolucao() {
2.     int l = -1;
3.     int c = -1;
4.     int i;
5.     CvScalar cor;
6.     int valorMapeado = -1;
7.     float pixelCentral = -1;
8.     LARGURA = img->width;
9.     ALTURA = img->height;
10.          NUM_BANDAS = img->nChannels;
11.          showImageProperties();
12.          LARG_SAIDA = LARGURA-(TAM_MASCARA);
13.    ALT_SAIDA = ALTURA-(TAM_MASCARA);
14.          printf("\nTamanho da mascara: %d", TAM_MASCARA);
15.    printf("\nLargura: %d - Altura: %d", LARGURA, ALTURA);
16.          printf("\nLargSaida: %d - AltSaida: %d", LARG_SAIDA, ALT_SAIDA);
17.          printf("\nALTURA-MASCARA: %d - LARGURA-MASCARA: %d", (ALTURA-
TAM_MASCARA),       (LARGURA-TAM_MASCARA));
18.    matrizImagem = (int**)calloc(ALTURA, sizeof(int));
19.          for ( i = 0; i < ALTURA; i++)
20.                matrizImagem[i] = (int*)calloc(LARGURA, sizeof(int));
21.          for ( l = 0; l < ALTURA; l++ ) {
22.                for ( c = 0; c < LARGURA; c++ ) {
23.                matrizImagem[l][c] = (int) cvGet2D(img, l, c).val[0];
24.                }
25.    }
26.          nova = cvCreateImage(cvSize(LARG_SAIDA, ALT_SAIDA), IPL_DEPTH_8U, 1);
27.    for (l=0; l< (ALTURA-TAM_MASCARA-15); l++) {
```

```
28.          #pragma omp parallel for
29.          for (c=0; c<(LARGURA-TAM_MASCARA-15); c++) {
30.                pixelCentral = substituirNDporIntervalo(l,c);
31.                valorMapeado = (int) (pixelCentral * 255);
32.                cor.val[0] = valorMapeado;
33.                cvSet2D(nova, l, c, cor);
34.          }
35.    }
36.    cvSaveImage("newCbers.tif", nova, 0);
37. }
.
```

Listagem 7.3 Trecho de paralelização da versão 2 - OpenMP.

Para executar esta versão, segue os comandos via terminal no Linux:

```
1 - cd /home/victor/TCC/OpenMP/Versao2
2 - gcc -pg -o cv cv_versao2.c -I/usr/include/opencv -lcv -lhighgui -
fopenmp
3 - time ./cv 0 15 cbersgde.tif
4 - gprof ./cv gmon.out  > output.txt
```

7.4 Versão 3 – OpenMP

A versão 3, utilizando a biblioteca *OpenMP*, assim como na versão anterior, foi colocado trecho de paralelização na função "convolução", vendo que era onde demandava mais tempo computacional. Foram adicionadas as linhas de número 27, 29 e 37. A linha 27 descreve que a seção a seguir será paralelizada além de criar as *threads* que serão utilizadas. A linha 29 denota que o laço de repetição posterior será paralelizado.

Foi usado um *for* com *schedule dynamic* para dizer que nem todas as *threads* terão a mesma quantidade de trabalho. Utilizando o *dynamic,* as iterações são distribuídas entre as *threads* à medida que as mesmas solicitam mais iterações. Cada *thread* executa um bloco de iterações (do tamanho definido pelo parâmetro *chunk_size*), em seguida solicita outro bloco até que já não existam mais blocos de iterações. O número 1 significa qual é o *chunk_size* deste loop, ou seja, explicitamente informando quantas iterações cada *thread* fará. A linha 37 explicita que ocorrerá uma sincronização entre as *threads* e aguardarão até que todas se encontrem naquele ponto.

```
1.  void convolucao() {

2.     int l = -1;
3.     int c = -1;
```

```
4.     int i;
5.     CvScalar cor;
6.     int valorMapeado = -1;
7.     float pixelCentral = -1;
8.     LARGURA = img->width;
9.     ALTURA = img->height;
10.          NUM_BANDAS = img->nChannels;
11.          showImageProperties();
12.          LARG_SAIDA = LARGURA-(TAM_MASCARA);
13.    ALT_SAIDA = ALTURA-(TAM_MASCARA);
14.          printf("\nTamanho da mascara: %d", TAM_MASCARA);
15.    printf("\nLargura: %d - Altura: %d", LARGURA, ALTURA);
16.          printf("\nLargSaida: %d - AltSaida: %d", LARG_SAIDA, ALT_SAIDA);
17.          printf("\nALTURA-MASCARA: %d - LARGURA-MASCARA: %d", (ALTURA-
TAM_MASCARA),      (LARGURA-TAM_MASCARA));
18.    matrizImagem = (int**)calloc(ALTURA, sizeof(int));
19.          for ( i = 0; i < ALTURA; i++)
20.                matrizImagem[i] = (int*)calloc(LARGURA, sizeof(int));
21.          for ( l = 0; l < ALTURA; l++ ) {
22.                for ( c = 0; c < LARGURA; c++ ) {
23.                matrizImagem[l][c] = (int) cvGet2D(img, l, c).val[0];
24.                }
25.    }
26.          nova = cvCreateImage(cvSize(LARG_SAIDA, ALT_SAIDA), IPL_DEPTH_8U, 1);
27.    #pragma omp parallel
28.    for (l=0; l< (ALTURA-TAM_MASCARA-15); l++) {
29.          #pragma omp parallel for schedule (dynamic,1)
30.          for (c=0; c<(LARGURA-TAM_MASCARA-15); c++) {
31.                pixelCentral = substituirNDporIntervalo(l,c);
32.                valorMapeado = (int) (pixelCentral * 255);
33.                cor.val[0] = valorMapeado;
34.                cvSet2D(nova, l, c, cor);
35.          }
36.    }
37.    #pragma omp barrier
38.    cvSaveImage(“newCbers.tif”, nova, 0);
39. }
```

Listagem 7.4 Trecho de paralelização da versão 3 - OpenMP.

Para executar esta versão, segue os comandos via terminal no Linux:

```
1 - cd /home/victor/TCC/OpenMP/Versao3
2 - gcc -pg -o cv cv_versao3.c -I/usr/include/opencv -lcv -lhighgui -
fopenmp
3 - time ./cv 0 15 cbersgde.tif
4 - gprof ./cv gmon.out  > output.txt
```

7.5 Versão 4 – OpenMP

Esta versão foi uma tentativa de simplificar a versão anterior, abordando uma tentativa mais simples no ponto de vista da programação. Foram adicionadas as

linhas de número 27, 29 e 37. A linha 27 indica que a seção a seguir será paralelizada além de criar as *threads* que serão utilizadas. A linha 29 indica que o laço de repetição posterior será paralelizado. A linha 37 explicita que ocorrerá uma sincronização entre as *threads* e aguardarão até que todas se encontrem naquele ponto.

```
1.  void convolucao() {
2.     int l = -1;
3.     int c = -1;
4.     int i;
5.     CvScalar cor;
6.     int valorMapeado = -1;
7.     float pixelCentral = -1;
8.     LARGURA = img->width;
9.     ALTURA = img->height;
10.          NUM_BANDAS = img->nChannels;
11.          showImageProperties();
12.          LARG_SAIDA = LARGURA-(TAM_MASCARA);
13.    ALT_SAIDA = ALTURA-(TAM_MASCARA);
14.          printf("\nTamanho da mascara: %d", TAM_MASCARA);
15.    printf("\nLargura: %d - Altura: %d", LARGURA, ALTURA);
16.          printf("\nLargSaida: %d - AltSaida: %d", LARG_SAIDA, ALT_SAIDA);
17.          printf("\nALTURA-MASCARA: %d - LARGURA-MASCARA: %d", (ALTURA-
TAM_MASCARA),       (LARGURA-TAM_MASCARA));
18.    matrizImagem = (int**)calloc(ALTURA, sizeof(int));
19.          for ( i = 0; i < ALTURA; i++)
20.                matrizImagem[i] = (int*)calloc(LARGURA, sizeof(int));
21.          for ( l = 0; l < ALTURA; l++ ) {
22.                for ( c = 0; c < LARGURA; c++ ) {
23.                matrizImagem[l][c] = (int) cvGet2D(img, l, c).val[0];
24.                }
25.    }
26.          nova = cvCreateImage(cvSize(LARG_SAIDA, ALT_SAIDA), IPL_DEPTH_8U, 1);
27.    #pragma omp parallel
28.    for (l=0; l< (ALTURA-TAM_MASCARA-15); l++) {
29.          #pragma omp for
30.          for (c=0; c<(LARGURA-TAM_MASCARA-15); c++) {
31.                pixelCentral = substituirNDporIntervalo(l,c);
32.                valorMapeado = (int) (pixelCentral * 255);
33.                cor.val[0] = valorMapeado;
34.                cvSet2D(nova, l, c, cor);
35.          }
36.    }
37.    #pragma omp barrier
38.    cvSaveImage(“newCbers.tif”, nova, 0);
39. }
```

Listagem 7.5 Trecho de paralelização da versão 4 - OpenMP.

Para executar esta versão, segue os comandos via terminal no Linux:

```
1 - cd /home/victor/TCC/OpenMP/Versao4
2 - gcc -pg -o cv cv_versao4.c -I/usr/include/opencv -lcv -lhighgui -
fopenmp
3 - time ./cv 0 15 cbersgde.tif
4 - gprof ./cv gmon.out  > output.txt
```

7.6 Versão 5 – OpenMP

Na versão 5 utilizando a biblioteca *OpenMP*, foi uma versão similar à versão 3, mudando o laço de repetição *for schedule dynamic* para *for schedule static.* Nesse caso, as iterações são divididas em pedaços de tamanho definido pelo parâmetro *chunk_size*. A distribuição dos blocos de iteração entre as *threads* é feita de forma estática e cíclica, respeitando a ordem de numeração das mesmas. Se a divisão do número de iterações pelo *chunk_size* não for exata, o último bloco terá um número menor de iterações. Quando o *chunk_size* não é especificado, cada *thread* fica com um bloco de tamanho aproximadamente igual ao número de iterações dividido pelo número de *threads*. Quando a cláusula *schedule* não é utilizada no construtor *for*, o padrão do *OpenMP* é o mesmo definido pelo static.

Foram adicionadas as linhas de número 28 e 36. A linha 28 indica que o laço de repetição posterior será paralelizado. A linha 36 explicita que ocorrerá uma sincronização entre as *threads* e aguardarão até que todas se encontrem naquele ponto.

```
1.  void convolucao() {
2.     int l = -1;
3.     int c = -1;
4.     int i;
5.     CvScalar cor;
6.     int valorMapeado = -1;
7.     float pixelCentral = -1;
8.     LARGURA = img->width;
9.     ALTURA = img->height;
10.          NUM_BANDAS = img->nChannels;
11.          showImageProperties();
12.          LARG_SAIDA = LARGURA-(TAM_MASCARA);
13.    ALT_SAIDA = ALTURA-(TAM_MASCARA);
14.          printf("\nTamanho da mascara: %d", TAM_MASCARA);
15.    printf("\nLargura: %d - Altura: %d", LARGURA, ALTURA);
16.          printf("\nLargSaida: %d - AltSaida: %d", LARG_SAIDA, ALT_SAIDA);
17.          printf("\nALTURA-MASCARA: %d - LARGURA-MASCARA: %d", (ALTURA-
TAM_MASCARA),       (LARGURA-TAM_MASCARA));
18.    matrizImagem = (int**)calloc(ALTURA, sizeof(int));
19.          for ( i = 0; i < ALTURA; i++)
20.                matrizImagem[i] = (int*)calloc(LARGURA, sizeof(int));
21.          for ( l = 0; l < ALTURA; l++ ) {
22.                for ( c = 0; c < LARGURA; c++ ) {
23.                matrizImagem[l][c] = (int) cvGet2D(img, l, c).val[0];
24.                }
25.    }
26.          nova = cvCreateImage(cvSize(LARG_SAIDA, ALT_SAIDA), IPL_DEPTH_8U, 1);
27.    for (l=0; l< (ALTURA-TAM_MASCARA-15); l++) {
28.          #pragma omp parallel for schedule (static)
29.          for (c=0; c<(LARGURA-TAM_MASCARA-15); c++) {
```

```
30.                 pixelCentral = substituirNDporIntervalo(l,c);
31.                 valorMapeado = (int) (pixelCentral * 255);
32.                 cor.val[0] = valorMapeado;
33.                 cvSet2D(nova, l, c, cor);
34.           }
35.    }
36.    #pragma omp barrier
37.    cvSaveImage("newCbers.tif", nova, 0);
38. }
```

Listagem 7.6 Trecho de paralelização da versão 5 - OpenMP.

Para executar a versão 5, segue os comandos via terminal no Linux:

```
1 - cd /home/victor/TCC/OpenMP/Versao5
2 - gcc -pg -o cv cv_versao5.c -I/usr/include/opencv -lcv -lhighgui -
fopenmp -O3
3 - time ./cv 0 15 cbersgde.tif
4 - gprof ./cv gmon.out  > output.txt
```

7.7 Versão 1 - pThread

A versão pThread V1 foi executada para comparar com os resultados obtidos nas versões com *OpenMP.* Esta é a versão desenvolvida em (Shishido, 2010), sem nenhuma alteração. Para executá-la, segue os comandos via terminal no Linux:

```
1 - cd /home/victor/TCC/pThread
2 - gcc -o cv_thread cv_thread.c -I/usr/include/opencv -L/usr/lib -lcv -
lhighgui -lpthread
3 - time ./cv_thread 15 cbersgde.tif X
4 - gprof ./cv_thread gmon.out  > output.txt
```

7.8 Versão 2 - pThread

Nesta versão, não houve mudanças referente ao código da versão 1 *pThread.* Foi adicionado o marcador "-O3" ao final da linha de compilação para tentar obter um resultado melhor do que a primeira versão. Para executá-la, segue os comandos via terminal no Linux:

```
1 - cd /home/victor/TCC/pThread2
2 - gcc -o cv_thread cv_thread2.c -I/usr/include/opencv -L/usr/lib -lcv -
lhighgui -lpthread -O3
3 - time ./cv_thread 15 cbersgde.tif X
4 - gprof ./cv_thread gmon.out  > output.txt
```

8 RESULTADOS E DISCUSSÃO

Este capítulo tem por objetivo apresentar os resultados encontrados, além dos pontos positivos e negativos de cada versão desenvolvida. Foram executadas as cinco versões paralelas com *OpenMP*, duas versões paralelas com *pThread* e a versão sequencial.

8.1 Tempos

A Tabela 8.1 contem os tempos de execução de todas as versões, referente ao número de *threads* utilizadas:

	Versões do algoritmo de convolução							
Nº de threads	**Sequencial**	**OpenMP V.1**	**OpenMP V.2**	**OpenMP V.3**	**OpenMP V.4**	**OpenMP V.5**	**pThread V.1**	**pThread V.2**
1	**43m 30s**	46m 11s	56m 12s	43m 32s	44m 35s	44m 10s	01h 48m 24s	47m 28s
2	43m 30s	48m 11s	01h 01m 40s	30m 33s	29m 24s	**23m 26s**	1h 14m 16s	24m 28s
4	43m 30s	01h 08m 26s	51m 16s	22m 26s	16m 02s	**12m 07s**	01h 03m 01s	15m 49s
8	43m 30s	01h 08m 40s	46m 30s	13m 30s	08m 30s	**06m 22s**	38m 16s	14m 45s

Tabela 8.1 Tabela geral dos tempos de execução todas as versões.

Dentre as versões utilizando *OpenMP*, a versão *OpenMP* V1 foi a pior versão em relação ao desempenho. Ela foi incorporada a este trabalho justamente para demonstrar que o mau uso da programação paralela pode reduzir o desempenho de um programa. Nesta versão, quanto mais *threads* foram alocadas, mais demorado era, causando uma curva de desempenho inversamente proporcional ao do conceito da programação paralela, que é descendente em relação ao tempo.

Quando se trabalha com máquinas *multicore*, às vezes, podem acontecer problemas quando os múltiplos núcleos tentam escrever em lugares da memória *cache* próximos um dos outros. A memória *cache* é dividida em “linhas”, com extensão de 32 ou 64 bytes dependendo do sistema operacional e da arquitetura do

processador. Quando um núcleo lê um valor da memória, na realidade carrega toda a linha de dados *cache,* não somente o valor a ser utilizado. Porém, quando outro núcleo precisa ler um valor da mesma linha da *cache* que o primeiro modificou, é posto em espera até que o primeiro núcleo libere o acesso a *cache*. Desse modo, para garantir que o segundo núcleo obtenha o valor atualizado, a unidade de controle bloqueia toda a linha da *cache* usada pelo primeiro núcleo. Por isso que nesta versão, quanto maior o número de *threads* alocadas, maior o tempo de execução.

O melhor tempo da versão V2 usando *OpenMP* foi de 46 minutos e 30 segundos, utilizando 8 *threads*. O resultado foi pior do que o da versão sequencial que atingiu 43 minutos e 30 segundos. Isso indica que não basta paralelizar o código para aumentar seu desempenho. É preciso conhecer a arquitetura da máquina para encontrar os gargalos para otimizar o tempo de execução. Na versão *OpenMP* V2, ocorreram alguns problemas encontrados na versão *OpenMP* V1, quando utilizada 2 *threads.* O tempo de ociosidade do processador levou a uma perda de performance comparada com apenas 1 *thread.* Já com 4 e 8 *threads*, apesar dessa ociosidade, o benefício ainda conseguiu ser maior do que o gasto, graças ao ganho computacional atingido com o número de *threads*.

Na versão *OpenMP* V3, o ganho em desempenho foi considerável. Utilizando o total de *threads* da máquina (8 *threads*), o resultado foi três vezes superior ao da versão sequencial. Para descobrir quais lugares do algoritmo levavam mais tempo para serem executados, foi usado o *gprofile,* e assim tornou-se possível descobrir que existia um alto tempo de execução no método “convolução”, onde seria o mais adequado para trabalhar com programação paralela. Com isso foi focado o ponto onde demandava maior tempo e adicionado o código de paralelização. Uma variação desta versão foi a *OpenMP* V4, cujo tempo de processamento foi o melhor dentre as anteriores.

Melhorar o tempo de 43m e 30s para 08m e 30s é um resultado consideravelmente bom. Esta versão prova que quanto maior a capacidade computacional da máquina, tanto em número de processadores quanto em número de núcleos, os resultados tendem a formar uma linha descendente em relação ao tempo de desempenho.

A versão *OpenMP* V5 foi uma tentativa de melhorar a versão V4. Com a adição do parâmetro –O3 na compilação, o tempo que era de 08m e 30s reduziu-se para 06m e 22s. Este foi o melhor resultado das versões paralelas com *OpenMP*, e

se comparado com a melhor versão em *pThread* que foi de 14m, também foi superior.

A versão *pThread* V1 foi a versão paralelizada com *pThread* sem usar o marcador de otimização –O3. Esta versão foi pouco melhor do que a versão sequencial, mas não conseguiu concorrer contra as outras versões usando *OpenMP*. Usando uma máscara de 3 *pixels*, esta versão foi mais eficiente do que as outras versões, incluindo as versões em *OpenMP* que a superaram usando a máscara de 15 *pixels*. Para tentar melhorar ainda mais o resultado, foi realizado testes em uma versão com otimização na compilação.

A *pThread* V2, com a adição do código –O3 na compilação, fez o tempo de execução que era de 38 minutos e 18 segundos cair para 14 minutos e 45 segundos, se tornando um ótimo resultado e competindo com as versões em *OpenMP*. Dependendo da arquitetura utilizada, em uma máquina com mais processadores, esta versão poderia superar as melhores versões com *OpenMP*, devido ao ganho que a arquitetura com mais processadores forneceria. A figura 8.1 ilustra os tempos de todas as versões pelo número de *threads.*

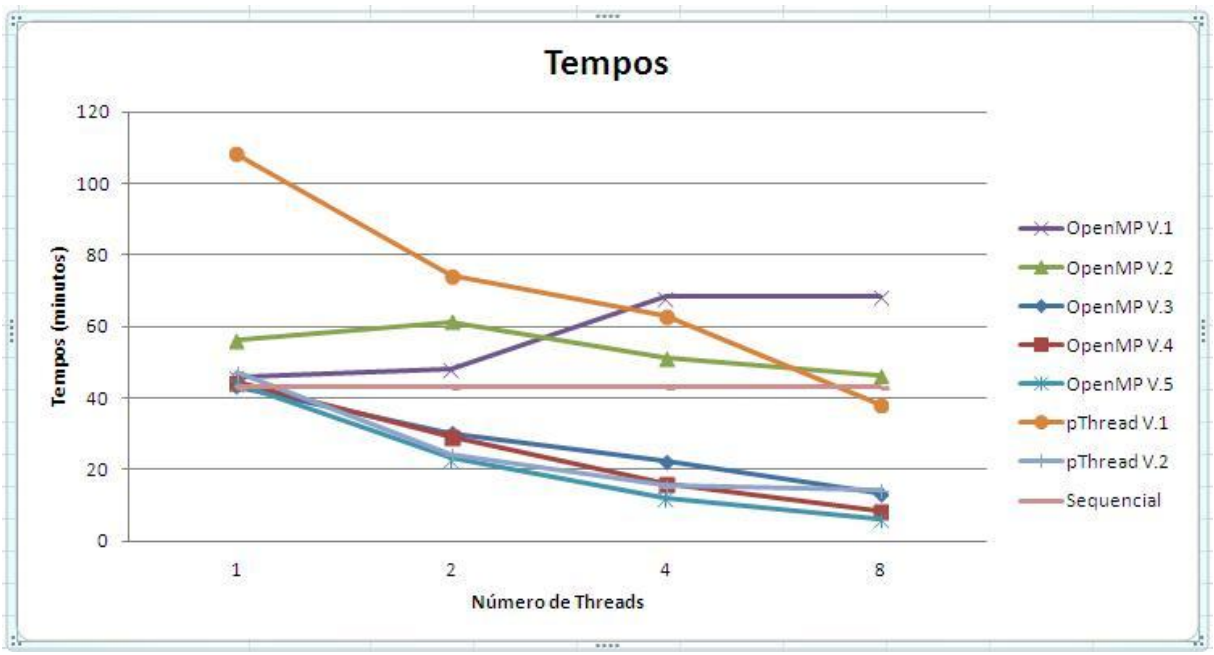

Figura 8.1 Gráfico geral dos tempos de execução todas as versões.

8.2 Speedup

A Tabela 8.2 contém o valor do *speedup* de todas as versões paralelas, referente ao número de *threads*:

	Versões do algoritmo de convolução						
Nº de threads	**OpenMP V.1**	**OpenMP V.2**	**OpenMP V.3**	**OpenMP V.4**	**OpenMP V.5**	**pThread V.1**	**pThread V.2**
1	0,94	0,77	**0,9992**	0,9757	0,9849	0,4012	0,9164
2	0,90	0,70	1,4238	1,4795	**1,8563**	0,5857	1,1779
4	0,63	0,84	1,939	2,713	**3,59**	0,6902	2,7502
8	0,63	0,93	3,2222	5,1176	**6,8324**	1,1357	2,9491

Tabela 8.2 Tabela geral dos valores de speedup de todas as versões.

Speedup está diretamente relacionado a quantas vezes o código paralelo foi melhor que o sequencial. Nesse sentido, os melhores valores para 2, 4 e 8 *threads* foram da melhor versão desenvolvida, *OpenMP* V5. Quanto ao valor referente a apenas uma *thread*, não é necessário dar muita importância pois tivemos 4 versões que ficaram na margem de 0,94 e 0,99. Um detalhe a ser ressaltado é o valor de *speedup* para quatro e oito *threads* das versões *pThread* V2 e *OpenMP* V4. O da *pThread* foi discretamente superior utilizando quatro *threads*, porém analisando o valor para oito, o resultado da versão *OpenMP* é bem superior.

Esta análise nos leva a concluir que a *OpenMP* consegue trabalhar melhor quando se depara com processadores *multicore* e *multithreads. O hyper threading* nesse caso fez uma diferença significativa a favor da versão *OpenMP*, considerando que a máquina que rodou os tempos possui quatro processadores agindo com 8 *threads*.

A figura 8.2 apresenta os valores de *speedup* das versões paralelas.

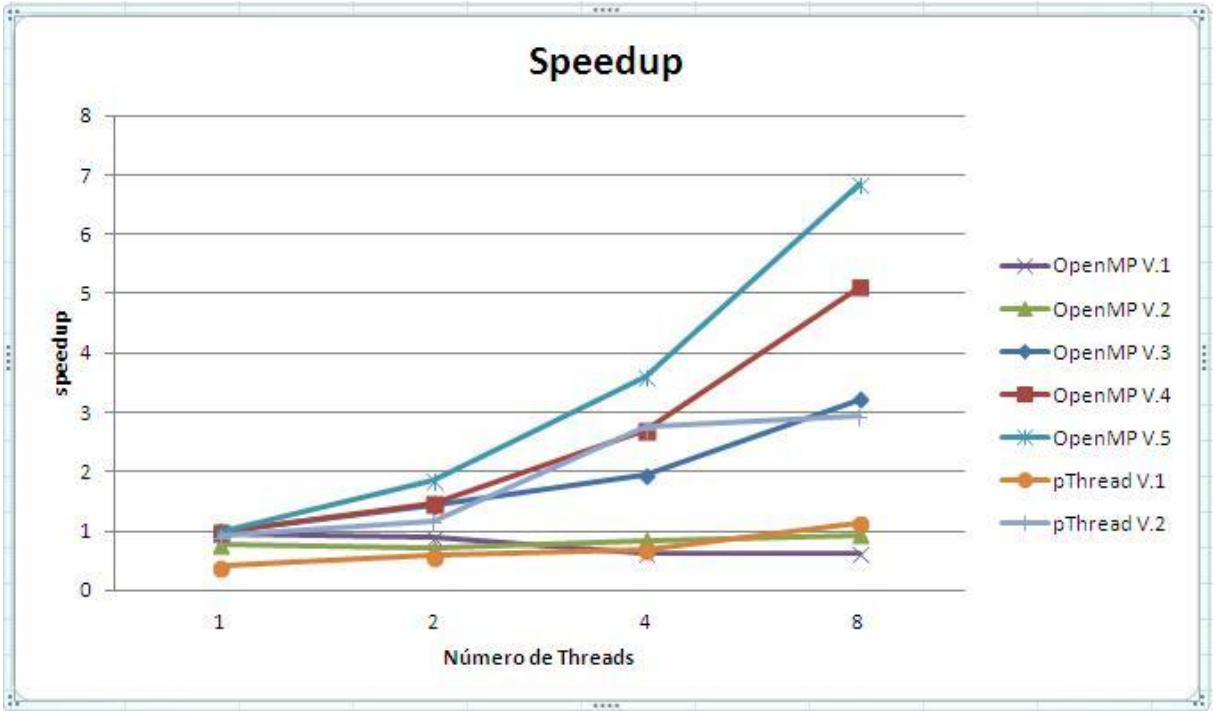

Figura 8.2 Gráfico geral dos valores de speedup de todas as versões.

8.3 Eficiência

A Tabela 8.3 apresenta a eficiência de todas as versões paralelas, referente ao número de *threads*:

	Versões do algoritmo de convolução						
Nº de threads	**OpenMP V.1**	**OpenMP V.2**	**OpenMP V.3**	**OpenMP V.4**	**OpenMP V.5**	**pThread V.1**	**pThread V.2**
1	94,18%	77,4%	**99,92**%	97,57%	98,49%	40,12%	91,64%
2	45,14%	35,27%	71,19%	73,97%	**92,81**%	29,28%	88,89%
4	15,89%	21,21%	48,47%	67,82%	**89,75**%	17,25%	68,75%
8	7,91%	11,69%	40,27%	63,97%	**85,4**%	14,19%	36,86%

Tabela 8.3 Tabela geral dos valores da eficiência de todas as versões.

Os valores da eficiência para as melhores versões da *OpenMP* deixaram evidente que o uso de cada processador foi mais utilizado comparado com os da *pThread*, principalmente no caso de *hyper-threading* onde a *OpenMP* V5 alcançou

incríveis 85,4% contra 36,86% da *pThread* V2. A *OpenMP* V5 manteve um alto índice de eficiência, independente do número de threads.

A figura 8.3 foi gerada a partir dos valores da eficiência, dispostos da seguinte forma:

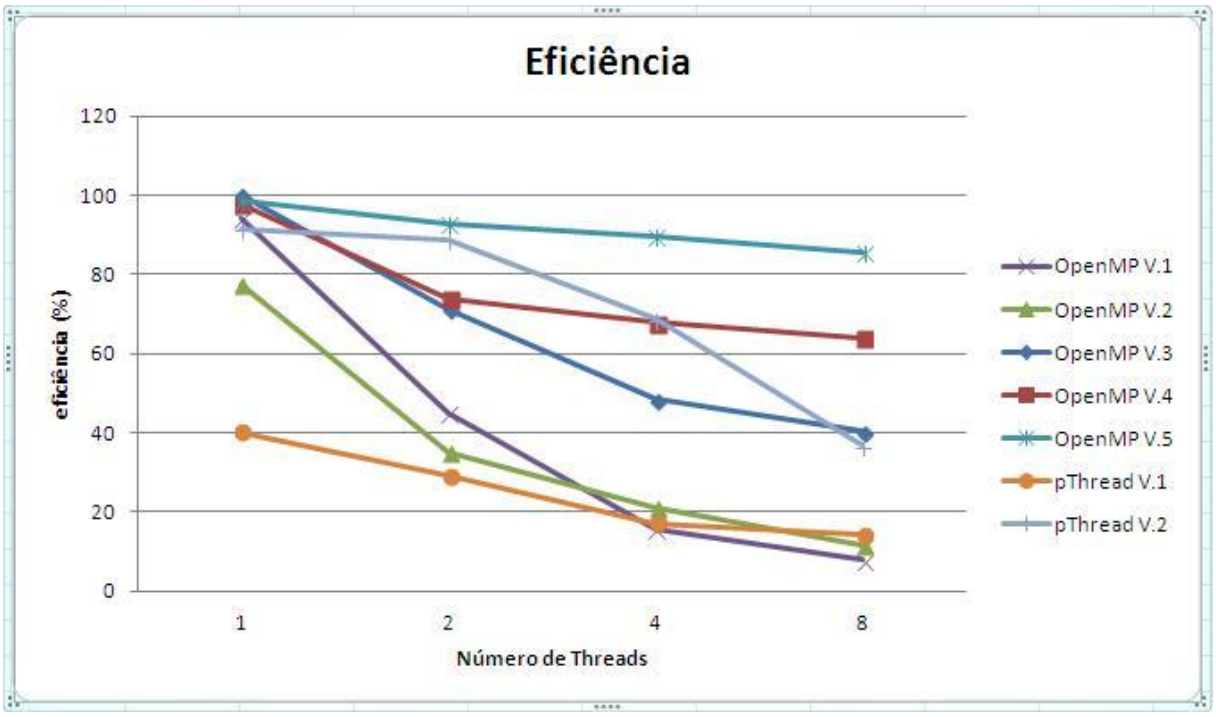

Figura 8.3 Gráfico geral dos valores de eficiência de todas as versões.

9 Considerações Finais

Não é possível afirmar que uma biblioteca é melhor do que a outra. Existem fatores a serem analisados, como complexidade de código, controle de paralelização e conhecimento sobre a arquitetura alvo. A *OpenMP* é mais recomendada quando você deseja apenas adicionar alguns trechos de `#pragma` e já ter uma versão paralela sem muitos esforços. Mas se é necessário maior controle sobre a paralelização, a mais recomendada é a *pThread*.

Outro fator que caracteriza diferenças entre as bibliotecas é que a *pThread* é baseada em *thread* enquanto a *OpenMP* é baseada em tarefas. Ou seja, a *OpenMP* pode alocar o número de threads igual ao número de processadores, tendo assim uma solução escalar. Para atingir o mesmo objetivo usando *threads* puras, demanda mais trabalho por parte do programador.

Em relação a controle, a *pThread* oferece um controle maior do que a *OpenMP* (criação, *joins*, exclusão múltiplas, sincronização explícita, entre outros). É importante ressaltar que as bibliotecas não possuem o mesmo objetivo. Para um desenvolvimento de uma aplicação laços de repetição aninhados e complexos, a *OpenMP* é mais indicada. Apesar da *pThread* também oferecer ganhos nessas áreas, é recomendada quando se deseja ter controle sobre toda a aplicação, desde a criação das *threads* até o fim da execução

O desenvolvimento deste trabalho é baseado em uma aplicação científica, utilizando muitos laços de repetição aninhados. Para um iniciante na área, levando em conta o algoritmo e a arquitetura computacional utilizada para fazer este estudo, seria interessante utilizar a biblioteca *OpenMP*, devido a facilidade de programação e ganho computacional nos diversos laços de repetição existentes. Pelos resultados encontrados, a *OpenMP* foi superior à *pThread*, principalmente porque a *OpenMP* trabalha melhor nas situações de laços aninhados.

REFERÊNCIAS BIBLIOGRÁFICAS

ALIAS, N.; SAID, N. M.; HIDAYAH, K.; SITI, N.; TIEN, C.; DOLLY, S.; TAU, P. The parallel algorithm for human tumor growth using distributed multiprocessor computer system. *Pervasive Computing Technologies for Healthcare.* PervasiveHealth, Second International Conference*.* p.331, 2008.

BABU, M. R.; KRISHNA, P. V.; KHALID, M. Optimization Techniques and Performance Evaluation of a Multithread Multi-Core Architecture Using OpenMP. *Advances in Computing and Communications.* v. 190, p. 182-191, 2011.

BARNEY, B. Introduction to parallel computing. Lawrence Livermore National Laboratory. Disponível em: https://computing.llnl.gov/tutorials/parallel_comp/. Acesso em: 12 maio 2012.

BOSILCA, G.; BOUTEILLER, A.; DANALIS, A.; FAVERGE, M.; HAIDAR, A.; HERAULT, T.; KURZAK, J.; LANGOU, J.; LEMARINIER, P.; LTAIEF, H.; LUSZCZEK, P.; YARKHAN, A.; DONGARRA, J. Flexible Development of Dense Linear Algebra Algorithms on Massively Parallel Architectures with DPLASMA. *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium.* p.1432-1441, 2011.

CURTIS-MAURY, M.; DING, X.; ANTONOPOULOS, C.D.; NIKOLOPOULOS, D.S. An Evalutation of OpenMP on Current and Emerging Multithreaded/Multicore Processors. *OpenMP Shared Memory Parallel Programing.* v. 4315, p. 133-144, 2008.

DEEPAK SHEKHAR, T.C.; VARAGANTI, K.; SURESH, R.; GARG, R.; RAMAMOORTHY, R. Comparison of Parallel Models for Multicore Architectures. *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium.* p. 1675-1682, 2011.

DUAN, G., WEICHANG, S., JIAO, C., LIN, Q. The implementation of KMP algorithm based on MPI + OpenMP. *Fuzzy systems and Knowledge Discovery (FSKD), 2012 9th International Conference.* p. 2511-2514, 2012.

FLYNN, M.; RUDD, K. Parallel architectures. *ACM Comput. Surv.* v. 28, n. 1, p.67-70, 1996.

HILLIS, W.D. Wrestling the future from the past: the transition to parallel computing. *Parallel & Distributed Technology: Systems & Applications, IEEE*. v.1, n.1, p. 6-7, 1993.

JANG, H.; PARK, A.; JUNG, K. Neural Network Implementation Using CUDA and OpenMP. *Digital Image Computing: Techniques and Applications (DICTA), IEEE.* p. 155-166, 2008.

JONES, M.D.; YAO, R. Parallel programming for OSEM reconstruction with MPI, OpenMP, and hybrid MPI-OpenMP. *Nuclear Science Symposium Conference Record, 2004 IEEE*. v.5, p.3036-3042, 2004.

LEE, D. J.; DOWNAR, T. J. The application of POSIX Threads and OpenMP to the U.S. NRC Neutron Kinetics Code PARCS. *OpeMP Shared Memory Parallel Programming.* v. 2104, p. 90-100, 2001.

LEE, M. K.; SONG, H. T.; YOON, H. S.; KWON, H. K.; JEON, W. J. OpenMP parallel programming using dual-core embedded system. *Control, Automation and Systems (ICCAS), 2011 11th International Conference.* p.762-766, 2011.

MURAO, H. Experiment of Multithreading Symbolic and Algebraic Computations with OpenMP. *Mathematical Software – ICMS 2006.* p. 426-437, 2006.

PFEIFFER, W.; STAMATAKIS, A. Hybrid MPI/Pthreads parallelization of the RAxML phylogenetic code. *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium.* p. 1-8, 2010.

RUFAI, R.A.; BOZYGIT, M.; ALGHAMALDI, J.S.; AHMED, M. Multithreaded Parallelism with OpenMP. *High Performance Scientific and Engineering Computing.* v. 750, p. 3-14, 2004.

SHISHIDO, H. Y. Paralelização de algoritmo de processamento de imagens digitais. Maringá, 2010. Dissertação de mestrado – Programa de Pós-Graduação em Ciência da Computação (PCC), Universidade Estadual de Maringá, 2010.

STAMATAKIS, A.; OTT, M. Exploiting Fine-Grained Parallelism in the Phylogenetic Likelihood Function with MPI, Pthreads, and OpenMP: A performance Study. *Pattern Recognition in Bioinformatics.* v. 5265, p. 424-435, 2008.

TARMYSHOV, K.; MÜLLER-PLATHE, F. Paralellizing a Molecular Dynamics Algorithm on a Multiprocessor Workstation Using OpenMP. *J. Chem. Inf. Model.* v.45, p.1943-1952, 2005.

TOMOV, S.; NATH, R.; LTAIEF, H.; DONGARRA, J. Dense linear algebra solvers for multicore with GPU accelerators. *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium*. p.1-8, 2010.

WENG, T-H.; PERNG, R-K.; CHAPMAN, B. OpenMP Implementation of SPICE3 Circuit Simulator. *International Journey of Parallel Programming.* v. 35, p.493-505, 2007.

WILDE, M.; FOSTER, I.; ISKRA, K.; BECKMAN, P.; ZHANG, Z.; ESPINOSA, A.; HATEGAN, M.; CLIFFORD, B.; RAICU, I. Parallel Scripting for Applications at the Petascale and Beyond. *IEEE Computer – computer.* v.42, n.11, p.50-60, 2009.

ZHONG, W.; ALTUN, G.; TIAN, X.; HARRISON, R.; TAI, P.; PAN, Y. Parallel protein secondary structure prediction schemes using Pthread and OpenMP over hyper-threading technology. *The jornal of Super Computing.* v.41, n.1, p.1-16, 2007.