dx.doi.org/10.7437/NT2236-7640/2012.02.001
Notas Técnicas, v. 2, n. 2, 2012

Comparativo de desempenho das técnicas de programação paralela forks e threads e aplicação em processamento de imagens

Fernanda D. Moraes, M. Giovani, N. Alves Jr., Marcelo Portes de Albuquerque, e Márcio P. de Albuquerque

Centro Brasileiro de Pesquisas Físicas - CBPF

Rua Dr. Xavier Sigaud,

150 – Urca – Rio de Janeiro – RJ – Brasil

Resumo

Alguns algoritmos aplicados na solução de problemas específicos de física exigem um alto desempenho computacional. Este é o caso por exemplo da área de processamento digital de imagens, onde as características de desempenho em termos de velocidade, algumas vezes com respostas em tempo real, nos leva ao uso de ferramentas da programação paralela. Para atender a essa demanda é importante uma compreensão dessas ferramentas, evidenciando suas diferenças e possibilidades de aplicações. No mesmo sentido, cabe destacar, que atualmente os centros de pesquisas em todo mundo tem a sua disposição clusters de computadores, ou plataformas computacionais que disponibilizam vários núcleos para cálculos científicos, tendo assim um forte potencial para uso destas técnicas de programação paralela.

É objetivo deste trabalho caracterizar as técnicas de programação paralela por threads e forks. Ambas as técnicas permitem o desenvolvimento de códigos de execução paralela, e tem restrições próprias na comunicação das informações entre os processos e no formato de programação. Este trabalho pretende evidenciar o uso de cada uma dessas técnicas, e ao final apresentar uma aplicação na área de processamento de imagens na qual ambas foram utilizadas.

A parte dedicada a aplicação das técnicas em processamento de imagens foi desenvolvida dentro da colaboração internacional com o Laboratório JET (*Join European Torus* da Agência Europeia de Energia Atômica/EURATOM). O JET estuda as instabilidades no processo de formação do plasma, que se manifestam como bandas de radiações, conhecidas como MARFE (*Multifaceted Asymmetric Radiation From The Edge*). Apresentamos técnicas de programação em paralelo em algoritmos de processamento digital de imagens com o objetivo de detectar o MARFE a uma taxa superior a 10.000 imagens/s. Os algoritmos desenvolvidos usam as técnicas de programação por threads e de memória compartilhada entre processos independentes, equivalentes ao fork.

Abstract

Several algorithms applied to the solution of specific problems in physics require high performance computing. This is the case, for example, in the field of digital image processing, where the required performance in terms of speed, and sometimes running in a real time environment, leads to the use of parallel programming tools. To meet this demand it is important to understand these tools, highlighting differences and their possible applications. Moreover, research centers around the world has available a clusters of computer, or a multi-core platform, with a strong potential of using parallel programming techniques.

This study aims to characterize threads and forks parallel programming techniques. Both techniques allow the development of parallel codes, which with its own restrictions on the inter process communication and programming format. This Technical Note aims to highlight the use of each of these techniques, and to present an application in the area of image processing in which they were used.

The application part of this work was developed in the international collaboration with the JET Laboratory (Join European Torus of the European Atomic Energy Community / EURATOM). The JET Laboratory investigates the process of forming the plasma and its instability, which appears as a toroidal ring of increased

radiation, known as MARFE (*Multifaceted Asymmetric Radiation From The Edge*). The activities have explored the techniques of parallel programming algorithms in digital image processing. The presented algorithms allow achieving a processing rate higher than 10 000 images per second and use threads and shared memory communication between independent processes, which is equivalent to fork.

Palavras-chave: fork, thread, comparativo, desempenho, multiprocessamento, speedup.

Conteúdo

1. Introdução 1
2. Conceitos fundamentais de processamento paralelo 1
2.1 Fork 2
2.2 Threads 2
2.3 Speedup 3
3. Metodologia 3
3.1. Plataforma computacional de teste 4
4. Resultados Obtidos 4
5. Aplicação das técnicas em processamento de imagens . 6
5.1. Estratégia de paralelização 7
5.2. Módulos de Processamento de Imagens em Paralelo 7
5.3. Resultados do Paralelismo no Processamento de Imagens 8
6. Conclusão 9
7. Anexo 9
8. Referências 9

1. INTRODUÇÃO

Novos algoritmos tem exigido cada vez mais um alto poder de processamento dos dispositivos eletrônicos. Por exemplo, na área de processamento digital de imagens, diversos algoritmos demandam um poder de cálculo diferenciado, e ainda maior quando a resposta do dispositivo tem que ser em tempo real. As técnicas de aquisição de imagem, processamento, armazenamento e comunicação das informações exigem alta velocidade na sua execução e ao mesmo tempo um excelente desempenho nos sistemas de reconhecimentos de padrões utilizados. Até recentemente grande parte dos algoritmos desenvolvidos eram construídos na forma serial, i.e., executados somente por um único processador. Um interesse pelo ganho de desempenho dos algoritmos em execução paralela se tornou a saída para ter um aumento maior do desempenho. A computação paralela e distribuída pode oferecer a potência computacional adequada, por exemplo, a este tipo de aplicação.

Em programação paralela existem normalmente duas abordagens principais: *pipelining* e *paralelismo de dados*. No Pipelining a divisão da carga de trabalho é realizada no tempo, de modo que uma unidade a ser processada flui através dos processadores. Nas abordagens de paralelismo de dados, a tarefa é dividida no espaço, de modo que diferentes conjuntos de dados são manipulados por diferentes processadores.

Outra consideração importante sobre o objetivo da construção de algoritmos rápidos é a necessidade final em si. Muitas das aplicações desenvolvidas tem o objetivo de execução em tempo real. No entanto, é importante apresentarmos uma definição apropriada sobre o conceito de execução de algoritmos em tempo real e sobre os sistemas de execução rápida. O objetivo no desenvolvimento de algoritmos rápidos é construir ferramentas para minimizar o tempo médio de resposta de um determinado conjunto de tarefas. No entanto, o objetivo de computação em tempo real é cumprir uma exigência de tempo específico de cada tarefa. Ao invés de ser rápido (que é um termo relativo), a propriedade mais importante de um sistema em tempo real é a previsibilidade, ou seja, sua funcionalidade e comportamento de tempo deve ser tão determinista quanto necessário para satisfazer as especificações do sistema principal. Computação rápida é útil para atender especificações de tempos rigorosos, mas esta sozinha não garante a previsibilidade [1], [2] e [3].

Nesta Nota Técnica procuramos abordar questões práticas da programação em paralelo. Em especial utilizando duas técnicas bastante interessantes: o fork e as threads. Vamos apresentar alguns conceitos fundamentais sobre o processamento paralelo, fazendo uma comparação entre forks e threads, e faremos uma avaliação sobre cada uma delas. Ao final apresentamos uma aplicação destas técnicas em um problema de processamento de imagens na área de fusão nuclear. Na conclusão apresentamos uma discussão sobre o uso de cada uma bem como o potencial de aplicação para a área de processamento digital de imagens.

2. CONCEITOS FUNDAMENTAIS DE PROCESSAMENTO PARALELO

O processamento paralelo é uma forma de computação eficiente do processamento da informação com ênfase na exploração simultânea de eventos na execução de um software, i.e., funciona com o princípio de que grandes problemas muitas vezes podem ser divididos em outros menores, que são então resolvidos simultaneamente (”em paralelo”). O principal motivo para a criação do processamento paralelo é a possibilidade de aumentar a capacidade de processamento de uma única máquina. Uma vez que existe limitação tecnológica da velocidade das máquinas sequenciais, a solução empregada para aumentar o poder de processamento é a utilização de multiprocessadores.

No entanto, não basta apenas ter uma arquitetura paralela, ou seja, com vários processadores, pois o software também tem que estar no formato de execução paralelo. A seguir, vamos apresentar duas formas de desenvolver programas paralelizados.

2.1. Fork

Em geral, os computadores que possuem sistemas operacionais multitarefa disponibilizam um conjunto de funções para divisão e compartilhamento do(s) processador(es) e da memória. Estes sistemas costumam disponibilizar chamadas ao kernel que possibilitam a criação de múltiplos processos[1]. Se a máquina tem mais de um processador, o sistema operacional distribui os processos pelos processadores. No GNU/Linux e nas variantes do Unix, um processo pode ser clonado com a função fork. A comunicação entre os processos é feita de forma simplicada com o uso de *pipes*[2][4]. Outras tecnologias também podem ser citadas no suporte a troca de informações entre processos, tais como: arquivos (files), memória compartilhada (shared memory - shm) [5], ipc (*inter process communication*) [6], semáforos (sem) [7], passagem de sinais (*signal*) [8], sockets [9] dentre outros. Nesta Nota Técnica, abordaremos o uso de forks com a comunicação através de memória compartilhada.

Basicamente, quando é usada a instrução fork dentro do programa, este cria uma cópia exata a partir do ponto em que o comando é executado e dois processos idênticos são executados ao mesmo tempo. O processo principal, o qual chama o comando fork, é denominado processo pai e os processos que são gerados pelo pai são chamados de processos filhos. Os processos filhos ao serem criados ganham um novo espaço de memória onde terão variáveis globais diferentes das variáveis do processo pai, ainda que possuam os mesmos nomes na programação. Além disso, todos os processos podem ter acesso a uma região memória compartilhada especialmente criada onde poderão compartilhar (ler, escrever e editar) as mesmas informações.

Em computação paralela a memória compartilhada pode ser definida como um método de comunicação entre processos. Um dos processos irá criar uma área em memória na qual outro processo pode acessá-la, figura 1. Uma vez que ambos os processos podem acessar a região de memória compartilhada como uma memória de uso regular, é criada um ponte de comunicação extremamente rápida (em comparação com métodos como pipes, sockets, etc). No entanto, a necessidade destes processos em funcionar no mesmo computador e não usar a rede de computadores, torna-o um formato mais restrito de comunicação entre aplicações.

As bibliotecas dinâmicas são normalmente utilizadas e mapeadas para os múltiplos processos. As normas POSIX do UNIX padronizou uma API para uso da memória compartilhada (POSIX Shared Memory). As comunicações entre processos na arquitetura POSIX (*POSIX:XSI Extension*) inclui funções para implementação de memória compartilhada como: shmat, shmctl, shmdt e shmget.

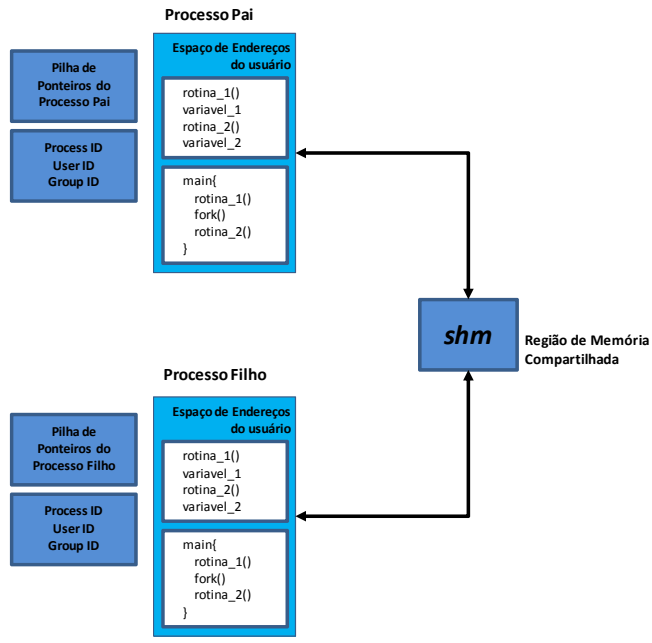

Figura 1: Diagrama representando o programa desenvolvido com a técnica fork. A comunicação entre os processos é feita pela região de memória compartilhada (shm). As rotinas e as variáveis estão em segmentos de memória distintos e os processos não podem acessar as regiões de memória um do outro.

2.2. Threads

Threads (ou processos leves) é uma das maneiras utilizada por um processo para dividir a si mesmo em duas ou mais tarefas que podem ser executadas simultaneamente, em geral, em arquiteturas multiprocessadas. O suporte a threads deve ser oferecido pelo sistema operacional e as aplicações devem ser implementadas fazendo uso de alguma biblioteca para esse fim. Neste trabalho foi usada a biblioteca “pthread” (*POSIX Threads*)[10].

Threads usufruem do compartilhamento de memória, por consequência cada thread pode acessar qualquer posição de memória dentro do espaço de endereçamento do processo, figura 2. Por isso, é possível a uma thread ler, escrever ou até apagar informações usadas por outra thread, exigindo um maior cuidado por parte do programador.

O compartilhamento do endereçamento em memória torna as threads mais ágeis no processo de troca de contexto pelo escalonador em comparação com processos independentes (forks). Essa característica pode impactar diretamente no desempenho da aplicação, uma vez que o sistema pode executar a troca de contexto dezenas de vezes por segundo ou até mesmo centenas de vezes por segundo.

[1] Processo é um programa em execução o qual contém fluxo ordenado de execução em um segmento de memória, possuindo suas próprias variáveis em memória e um identificador único (ID) que, associado a esse identificador, o Sistema Operacional gerencia permissões, prioridades, acesso a i/o, registros e pilhas. Se duas instâncias de um programa são executadas pode-se dizer que dois processos foram iniciados pois estes receberão identificadores diferentes. O conceito de processo pode ser simplificado como uma unidade de trabalho do Sistema Operacional ou mais basicamente como ”um programa em execução”.

[2] *Pipe* pode ser considerado um canal de comunicação que liga dois processos e permite um fluxo de informação unidirecional.

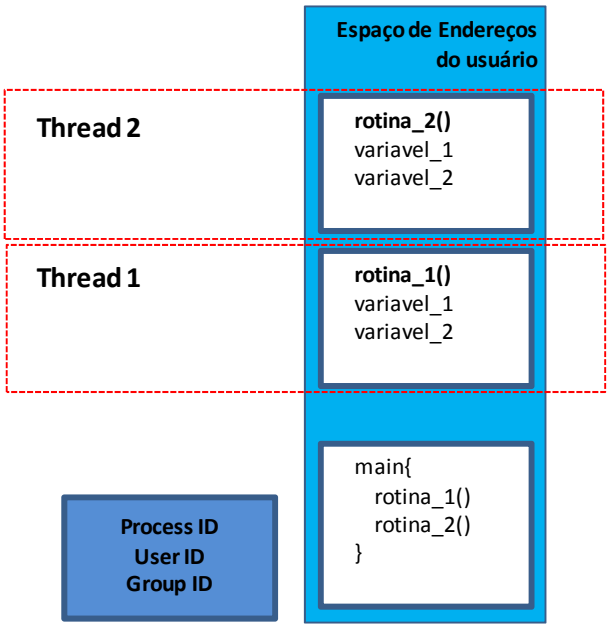

Figura 2: Diagrama representando o programa desenvolvido com a técnica de threads. As rotinas e as variáveis ocupam o mesmo segmento de endereçamento de memória.

2.3. Speedup

A computação paralela tem como objetivo aumentar a performance no processamento de informações, para isso é preciso realizar medições de desempenho. A forma mais utilizada para medir desempenho de um algoritmo paralelo é comparando-o com o da sua versão serial. Essa comparação recebe o nome de "speedup".

O Speedup mede o fator de redução do tempo de execução de um programa paralelizado em um número n de processadores e, ele é obtido pelo calculo da razão do intervalo de tempo de processamento do programa em um único processador denominado Ts (s = serial) pelo intervalo de tempo de execução em um computador com P processadores, denominado Tp.

Existem dois tipos de Speedup, o dito relativo quando Ts é igual ao tempo de execução do programa paralelo executado em um processador e é dito absoluto quando Ts é igual ao tempo de execução do "melhor" programa sequencial. Neste trabalho só temos o interesse em utilizar o Speedup relativo.

Um programa pode ser dividido em uma parte que pode ser executada em paralela e outra que só poderá ser executada na forma serial. Segundo a lei de Amdhal [11] (Eq 1), o speedup é obtido pela relação entre o tempo total de execução (Ts + Tp) sob a soma do tempo da parte serial (Ts) com o tempo total das partes paralelas (Tp) dividido pelo número de CPUs usadas para sua execução (n), onde 1 é a forma parametrizada da soma total de tempo de execução de forma serial (Ts + Tp).

$$Speedup = \frac{1}{T_s + \frac{t_p}{n}} \tag{1}$$

A tarefa proposta para a avaliação das duas tecnologias de paralelismo, criação e multiplicação de matrizes, possui um nível de paralelismo extremamente alto. Considerando os tempos de criação de processo somado aos tempos de criação de variáveis e ajustes de ambiente, com o tempo computacional necessário ao cálculo do produto das matrizes, pode-se dizer que a tarefa pode ser completamente paralelizável. Neste caso específico a lei de Amdhal ficaria:

$$Speedup_{T_s \to 0} = \frac{T_s + T_p}{T_s + \frac{T_p}{n}} = \frac{T_p}{\frac{T_p}{n}} = \frac{n\,T_p}{T_p} \rightarrow Speedup = n \tag{2}$$

A Equação 2 descreve em teoria o comportamento esperado nos gráficos de Speedup avaliados neste trabalho. Caso a tecnologia necessária para promover o paralelismo não possuísse custos computacionais adicionais os gráficos de Speedup deveriam ser uma função linear crescente diretamente proporcional ao número de CPUs usadas nos cálculos.

3. METODOLOGIA

Este trabalho tem como objetivo observar o comportamento da programação em paralelo sob duas diferentes técnicas de paralelismo, fork e thread. Para isso, serão executados sucessivamente dois programas similares, cada um utilizando uma das técnicas em várias CPUs, e a partir disto extrair médias dos intervalos de tempo para a geração de gráficos comparativos.

Os algoritmos apresentados consistem em gerar duas matrizes quadradas, "A" e "B", e em seguida realizar o produto entre elas colocando o resultado da operação na matriz "C". Este cálculo se encaixa bem em uma análise para avaliação de algoritmos de execução em paralelo. A medida que os programas forem executados, não mais de maneira sequencial, então cada tarefa será dividida entre as CPUs. Esta divisão de tarefas é simples, considerando que n seja o número de núcleos de processador, cada matriz tem sua $1/n$ parte paralelamente gerada e calculada por uma CPU. Isto acontece para que o tempo gasto, na geração das matrizes e no cálculo do produto seja divido pelo número de núcleos, que é a principal vantagem da programação em paralelo.

Os algoritmos serão executados para sete tamanhos diferentes de matrizes "A" e "B" (720, 1160, 1440, 1800, 2160, 2520 e 3000 linhas) e para os seguintes conjunto de processadores: 2, 3, 4, 6 e 8. Também será realizado a versão do algoritmo em um único processador para fins de comparação. Cada programa será executado sete vezes sob estes diferentes casos para obtermos a média dos tempos de execução.

Devemos ressaltar que durante o desenvolvimento dos programas tivemos a preocupação em mantê-lo o mais fiel possível nas duas implementações, i.e., mesmo que os códigos tenham sido escritos utilizando diferentes técnicas

de paralelismo, ambos usufruem dos mesmos métodos ao gerar as matrizes e calcular o produto entre elas.

Para se evitar o favorecimento de alguma técnica, tornando os executáveis mais rápidos por meio de otimizações de compilação, todos os programas usados neste trabalho fizeram uso das configurações default do compilador gcc (GNU C Compiler). Cabe destacar que nenhuma flags de compilação de otimização foi utilizada, como por exemplo a -O3".

Para realizar a medição do tempo de execução dos programas, será usada a função clock_gettime() definida no arquivo cabeçalho da linguagem C (header) time.h. A resposta do comando tem como conteúdo o tempo real, com a resolução na escala de nanosegundos. O intervalo de tempo será medido com a colocação deste comando entre o início e a finalização do programa [12].

3.1. Plataforma computacional de teste

A plataforma computacional de teste foi um nó Linux de um dos Clusters computacionais do CBPF, com as seguintes características: Supermicro, Super-Server, montagem de rack de 1U, 2 Placas mãe Supermicro X8DTT-H (Motherboard - Mobo), 16GB de memória RAM DDR3 1333MHz por MoBo, 2 processadores Intel Xeon E5550 HT Quad-Core 2.67GHz (16 pseudo núcleos/Mobo), com 16MB de memória cache L1/MoBo, 1TB de HD SATAII, Sistema operacional Unix-Like x86_64 / CentOS-5.5, Compilador C gcc versão 4.1.2, kernel 2.6.18-194.el5xe-SMP, x84_64, figura 3.

Figura 3: Visão geral da plataforma computacional utilizada para desenvolvimento e caracterização dos algoritmos em C. Um nó do Cluster do CBPF.

4. RESULTADOS OBTIDOS

A seguir, serão apresentadas as análises das simulações realizadas com os programas desenvolvidos a partir das duas técnicas de paralelismo. Cabe lembrar que durante todo o trabalho houve a preocupação em não deixar que programas de terceiros rodassem no mesmo momento dos testes, ou seja, a máquina, neste momento, estava dedicada exclusivamente à execução destas simulações.

A Figura 4 apresenta dois gráficos com as médias dos intervalos de tempo das sete vezes que cada programa foi executado, em função do tamanho das matrizes geradas. Nota-se que, naturalmente, o tempo de execução diminui à medida que se aumenta a quantidade de CPUs. Cada cor nesta figura representa a execução do programa em um número diferente de CPUs.

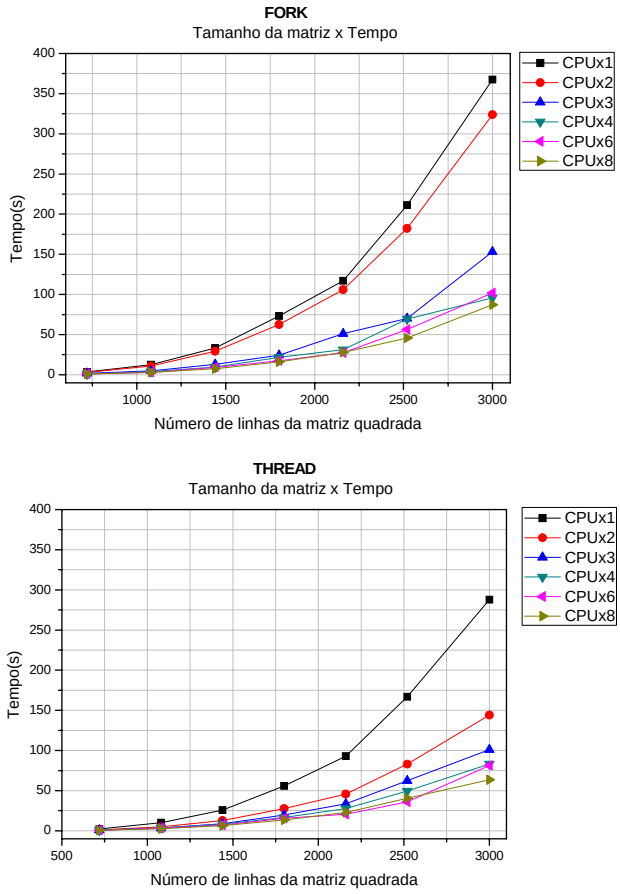

Figura 4: Intervalo de tempo médio de execução do programa em função do tamanho da matriz para diferentes quantidades de CPUs.

Para uma melhor visualização, apresentamos na figura 5 o mesmo resultado da figura anterior, porém agora no eixo das abscissas está a quantidade de núcleos de processador usada na paralelização dos programas.

A figura 6 apresenta o Speedup calculado a partir das simulações com matrizes quadradas de aresta igual a 3000, por número de CPUs. Estes valores foram obtidos pela razão entre a médias dos intervalos de tempos de execução para uma CPU e a média dos tempos para 2, 3, 4, 6 e 8 CPUs.

Para a construção deste gráfico os programas foram executados sete vezes para cada tamanho de matriz e quantidade de CPUs. Os pontos marcados no gráfico são referentes aos tempos médios de execução normalizados pelo tempo médio

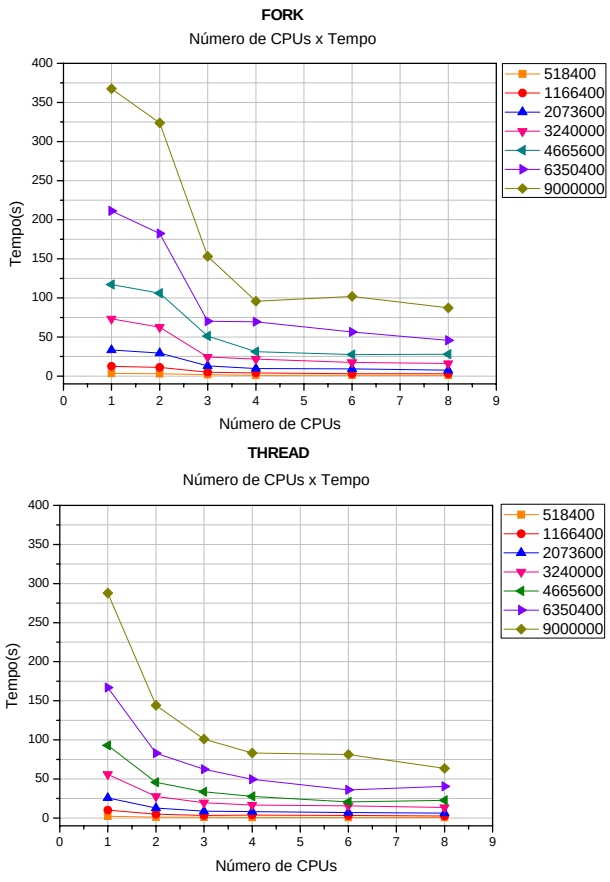

Figura 5: Intervalo de tempo médio de execução por número de CPUs. Cada curva corresponde ao número total de elementos das matrizes (produto linha x coluna).

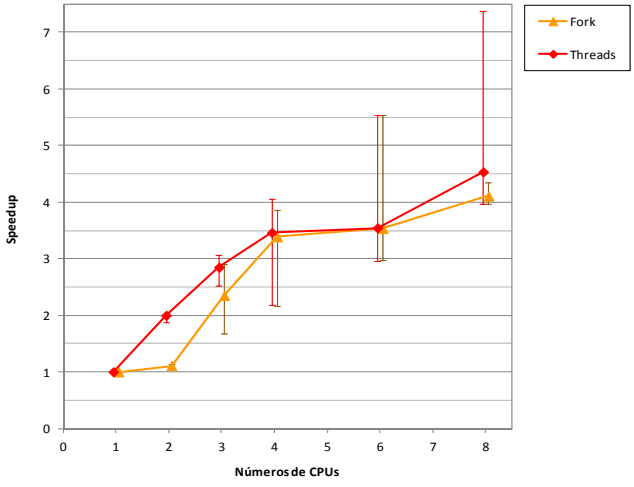

Figura 6: Speedup para matriz de 3000 linhas usando forks e threads.

da execução serial. O gráfico apresenta para cada ponto os limites superiores e inferiores correspondentes respectivamente aos valores máximos e mínimos dos speedup obtidos. O limite superior se refere ao máximo speedup encontrado nas execuções, obtido através da razão do tempo serial médio pelo tempo paralelo mínimo (Eq. 3). É possível notar neste gráfico que para o caso de 8 CPUs ao menos uma execução obteve um speedup próximo a 7,5.

$$Speedup_{max} = \frac{T_s \text{ (médio)}}{T_p \text{ (minimo)}} \tag{3}$$

O limite inferior se refere ao mínimo speedup encontrado nas execuções, obtido através da razão do tempo serial médio pelo tempo paralelo máximo (Eq. 4).

$$Speedup_{min} = \frac{T_s \text{ (médio)}}{T_p \text{ (máximo)}} \tag{4}$$

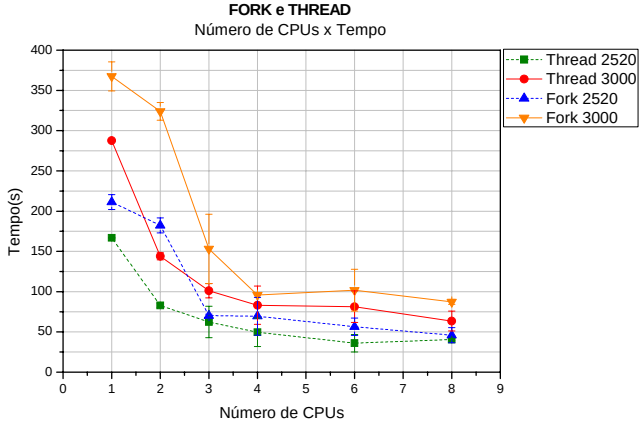

Figura 7: Intervalo de tempo médio de execução em função do número de CPUs para forks e threads, para matrizes com 2160 e 2520 linhas.

Na figura 7 foram selecionadas as curvas para os casos onde as matrizes tem 2520 e 3000 linhas. Neste gráfico é possível fazer uma melhor comparação do desempenho entre as técnicas fork e thread. As barras de erro correspondem ao desvio padrão de cada uma das medidas.

Do ponto de vista da execução dos programas aqueles implementados em threads foram mais rápidos para todos os 6 casos para as respectivas dimensões das matrizes . Os programas em threads são mais vantajosos principalmente porque estão utilizando um único segmento de memória, reduzindo assim a carga de trabalho do sistema operacional para gerenciar recursos necessários ao paralelismo. No caso de forks, este gerenciamento é necessário para o acesso das variáveis utilizadas em regiões de memórias compartilhadas.

Quanto maior o número de forks ou threads, maior o custo computacional do sistema operacional para gerenciar as inúmeras tarefas dedicadas ao paralelismo, denominado

como "*parallel overhead*", [13]. O acesso direto à um único range de endereço de memória de programa (threads) permite uma comunicação mais simples do que o gerenciamento de segurança e das permissões de endereços de memória compartilhada utilizada no fork (shm). A criação e destruição de novos processos também consomem recursos computacionais, tomando mais tempo quando comparada com processos equivalentes em suas versões seriais.

Essas características, dentre outras[3], justificam a não linearidade observada no gráfico do Speedup, figura 6. Como é possível observar nesta figura, no início a assíntota aparenta ter um comportamento linear, mas ao continuarmos crescendo o número de CPUs é possível chegar a um limite para o paralelismo. Independentemente do tipo de comunicação entre processos utilizado esse comportamento irá acontecer. Se estes programas estivessem utilizando a comunicação via rede de computadores o comportamento limitante do speedup seria evidenciado muito antes. Nesta análise comparativa a comunicação é realizada pela memória interna (para os dois casos), logo a mudança de comportamento acontecerá com o acréscimo de mais CPUs e será certamente posterior a uma situação em comunicação via rede.

5. APLICAÇÃO DAS TÉCNICAS EM PROCESSAMENTO DE IMAGENS

As técnicas de programação paralela foram aplicadas para o processamento de imagens em tempo real na área de fusão nuclear. Neste caso, o objetivo é processar mais de 10 mil imagens por segundo em um processo de reconhecimento de padrões. O algoritmo descrito nesta seção foi desenvolvido dentro da cooperação do Brasil com a Agência Europeia de Fusão Nuclear (EURATOM/EFDA) no Laboratório Joint European Torus (JET)[4], e coordenada pela Rede Nacional de Fusão (RNF/CNEN). Neste laboratório são desenvolvidas pesquisas com o objetivo de utilização da fusão nuclear como fonte de energia onde existem atualmente diversos desafios tecnológicos para a operacionalização deste tipo de equipamento. Um destes desafios é o controle e o diagnóstico em tempo real do plasma no interior da câmara toroidal magnética (tokamak). Diversas técnicas de controle têm sido propostas e uma das mais recentes e promissoras é a analise de assinaturas em vídeos obtidos por câmeras de alta velocidade situadas no interior do tokamak. A utilização destas câmeras de alta taxa de aquisição geram da ordem de GBytes de dados e partir destas imagens, uma série de técnicas de processamento têm sido propostas com o objetivo final de análise em tempo real para obter controle sobre a fusão nuclear. Na prática este se traduz pela detecção em tempo real de uma instabilidade que aparece como uma faixa luminosa de radiação denominada MARFE (*Multifaceted Asymmetric Radiation From The Edge*) [14], figura 8.

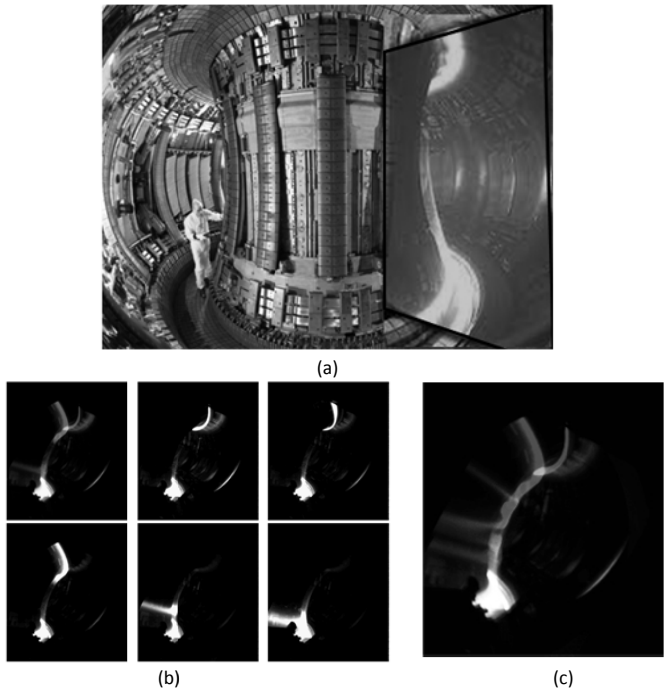

Figura 8: (a) Visão geral do Tokamak no Laboratório JET. A área em destaque corresponde a região onde são obtidas as imagens pela câmera de espectro visível (http://www.jet.efda.org). (b) Da esquerda para a direita e de cima para baixo é apresentado uma sequencia típica do fenômeno MARFE; (c) Imagens superpostas caracterizando uma sequencia completa de MARFEs.

Na tentativa de realizar este processamento em tempo real foi desenvolvido um conjunto de algoritmos em Linguagem C/C++ utilizando a biblioteca OpenCV [3] para processamento das imagens e LIBSVM para classificação de padrões [15]. O algoritmo desenvolvido conta com os seguintes módulos: aquisição da imagem (Op), estimação de imagem de fundo pela média das últimas *N* imagens (para detecção de movimento - módulo SAv), binarização da imagem (Bin), extração de características (FHu)[5] e classificação (Cls),

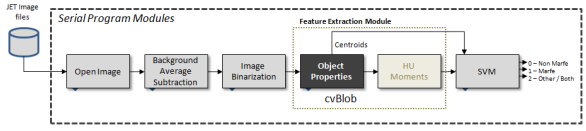

Figura 9: Módulos de Processamento de Imagens em sua versão seria e como base para implementação em processamento paralelo. O módulo de abertura da Imagem (Open Image - Op) foi considerado separadamente. Todo o processamento está dividido nos seguintes módulos: Subtração da imagem pela média das imagens de fundo (SAv); Binarização da imagem (Bin); Extração de características (FHu) e Classificação pela técnica de máquinas de vetores de suporte (Cls).

[3] Deve ser destaca outras características que também geram parallel overhead: sincronização, bibliotecas, ferramentas do sistema operacional e o compilador utilizado.

[4] $JET/EFDA - http://www.jet.efda.org$

[5] O módulo de Extração de Características (FHu) implementa a detecção da região de pixels pela técnica de centro gravidade e momentos Hu. Um

figura 9. No processo de paralelização descrito nesta secção todos os módulos foram levado em consideração, exceto o de abertura da imagem. Para mais detalhes sobre este módulo e a tarefa de processamento de imagens realizadas pelos outros veja a referência [15].

5.1. Estratégia de paralelização

O desenvolvimento de programas em paralelo é normalmente uma tarefa complexa. Mesmo os algoritmos que têm implementações eficientes e escaláveis, em sua versão paralela, são muitas vezes maiores, mais complexos, e muito mais difíceis para validar que seus homólogos na versão serial.

Um exemplo da imagem utilizada neste estudo pode ser visto na figura 10, com o destaque para a Região de Interesse onde o processamento estará concentrado. O restante da imagem é descartado para não influenciar no tempo total de processamento.

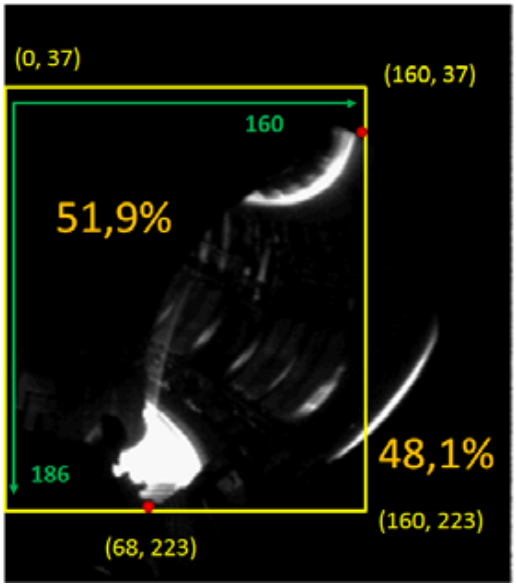

Figura 10: A imagem original com a representação da região de interesse, onde o processamento estará concentrado. O tamanho final da imagem é de 160 x 186 pixels e frequência de aquisição pela câmera KL8 do JET é de 30.000 quadros por segundo.

5.2. Módulos de Processamento de Imagens em Paralelo

Os módulos de processamento das imagens foram divididos em dois grupos. Ambos foram re-divididos em três tarefas executadas em paralelo. O módulo de Subtração da Imagem de Fundo (SAv) e Binarização da Imagem (Bin) constituem o Grupo 1 e os módulos de Extração de Características (FHu) e Classificação (Cls) o Grupo 2. Todos os processos foram executados com a máxima prioridade do Sistema Operacional Linux e no modo tempo real, em uma fila do tipo FIFO e direcionados para um processador específico[6]. Um processador adicional foi utilizado para controlar todo o processo. O diagrama de divisão dos grupos é apresentado na figura 11 abaixo.

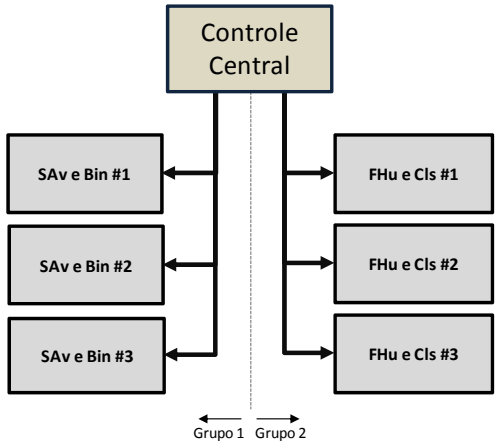

Figura 11: Estratégia de paralelização: o código foi dividido em dois grupos (1: Subtração da Imagem de Fundo - SAv) e Binarização da Imagem (Bin) e 2: com os módulos de Extração de Características e Classificação). Cada grupo foi redividido e executado em tres processadores exclusivos. Uma thread principal ocupou uma outra CPU, sendo dedicada ao controle de todos as outras tarefas em todos os processadores.

- Grupo 1: Os módulos de Subtração da Imagem de Fundo e de Binarização da Imagem foram paralelizados utilizando a técnica de paralelismo de dados. A imagem foi dividida em três regiões iguais e processada separadamente por três CPUs. Para esta implementação utilizamos a técnica de threads e as variáveis foram protegidas usando a técnica de programação em seção crítica. Uma seção crítica corresponde à parte do código que acessa um recurso compartilhado que não deve ser acessado simultaneamente por mais de uma thread. A técnica de programação por secções críticas também foram utilizados para controlar a distribuição de imagens para as tarefas do Grupo 2.

  A imagem foi dividida em três regiões e cada thread executa a sua parte de processamento no calculo da imagem de fundo. Elas devem finalizar com o cálculo da imagem binária e de novo e recalcular a imagem

trabalho complementar onde é discutida está técnica pode ser encontrado na referência [16].

[6] O comando "taskset"do Linux pode ser utilizado para alocar o processo a um processador específico. Com isso é possível evitar que os processos venham ser executados pelo mesmo processador evitando uma degradação do desempenho final.

de fundo para o processo seguinte como a média calculada a partir das N imagens anteriores, figura 12. Ao final de cada cálculo uma variável compartilhada e protegida é ajustada com o objetivo de sincronizar todos as threads com aquela dedicada ao controle central.

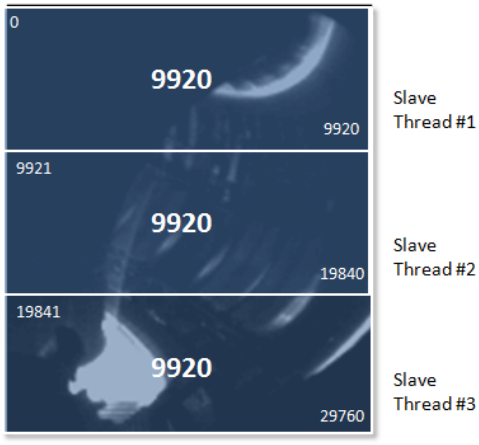

Figura 12: Estratégia para o paralelismo do Grupo 1 para o cálculo do imagem de fundo e binária. A imagem representa as três regiões de interesse de 9920 pixels processadas por três threads.

- Grupo 2: os módulos de Extração de Características e Classificação foram paralelizados utilizando a técnica de pipelining, implementadas em processos independentes (técnica análoga ao fork). Ao final do cálculo da imagem binária pelas threads do Grupo 1, esta está pronta para ser transferida para um processo do grupo 2 disponível. O algoritmo executa um procedimento de pesquisa para determinar qual das três CPUs está livre, e transfere a imagem binária para ela. Vale ressaltar que em um dado instante de tempo, o Grupo 2 está trabalhando em três imagens diferentes. Esta é a principal vantagem do método de processamento de pipelining.

5.3. Resultados do Paralelismo no Processamento de Imagens

O objetivo final é o aumento da taxa de processamento de imagens, medido pela quantidade de imagens processadas por segundo. Nesta seção iremos apresentar os resultados e discutir como as técnicas facilitaram a implementação dos códigos.

Normalmente a análise de desempenho de algoritmos em paralelo exigem o acompanhamento da execução dos processos em um digrama de tempo. A figura 13, apresenta este diagrama de tempo tendo como exemplo o processamento das imagens 552, 553 e 554 de um conjunto de 9950 imagens do Laboratório JET. No eixo das abscissas está representado o tempo e nas ordenadas cada um dos 7 processadores ocupados (ou não) pela tarefas em execução paralela. O processador número 0 é responsável pelo controle central e pela abertura da imagem. Os processadores 1 até 3 são responsáveis pelas tarefas do Grupo 1. Neste caso foi empregado a técnica de thread para cálculo da imagem final a ser transferida e processada posteriormente pelos processadores do Grupo 2. Os processadores 4 até 6 são responsáveis pelas tarefas do Grupo 2. Estes processos foram lançados de forma independente e são executados em um segmento próprio de memória. A comunicação entre esses processos é feita por meio de memória compartilhada.

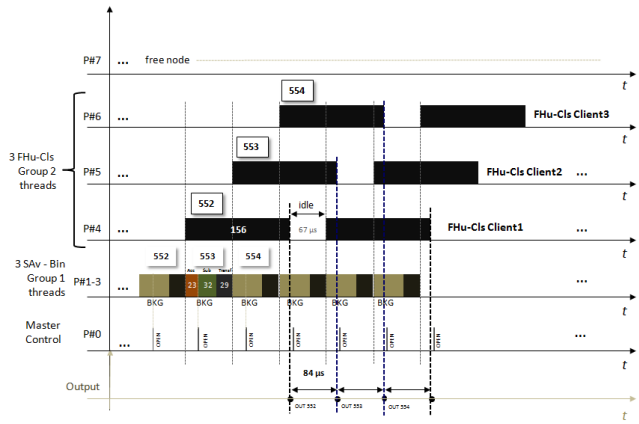

Figura 13: Diagrama de tempo da execução paralela do algoritmo dedicado a detecção de MARFEs, para a sequencia de imagens (552, 553 e 554). As threads do Grupo 1 (SAv e Bin) foram alocadas aos processadores 1 até 3, utilizando a técnica de paralelismo de dados. As tarefas do Group 2 (FHu e Cls) foram processadas pelos nós 4, 5 e 6 utilizando a técnica de pipeline. O processador número 0 é responsável pelo controle e distribuição das atividades. Os três números no interior da linha de tempo das tarefas do Grupo 1 (23, 32 e 29 μs) apresentam o tempo para cálculo de acumulação, subtração da imagem de fundo e transferência para o Grupo 2 respectivamente.

Com este diagrama podemos estimar o tempo de processamento de cada imagem, observada no processador 0, que foi de 84μs (aproximadamente 11.900 imagens / segundo)[7]. A versão serial deste mesmo algoritmo, na mesma plataforma computacional, teve um desempenho médio de 650 imagens/segundo.

Em quase todos os casos de desenvolvimento de algoritmos paralelo, o inicio se dá pela validação da versão serial. Este foi também o caso para o processamento das imagens de fusão nuclear apresentadas aqui. Os códigos desenvolvidos para o processamento das imagens são longos e extensos e a sua adaptação para a versão paralela é uma importante etapa de validação em si. Neste caso ficou claro, que as técnicas de programação paralela (seja por threads ou por processos independentes, como forks, em conjunto com as de passagem de mensagens por memória compartilhada) permitiram a realização de testes de desempenho em uma versão paralela inicial do código de processamento de imagens. O desenvolvimento por threads foi muito útil

[7] A medida de tempo em ambientes multi-core deve ser realizada em um único processador [17].

quando a modificação no código serial foi pequena, como foi o caso para os módulos do Grupo 1. No caso do Grupo 2, o uso de processos independentes em memória permitiu ter várias instâncias do mesmo programa funcionando sem ter que adaptar todas as variáveis internas de um código relativamente extenso. Em uma versão futura entendemos que um novo código deverá ser desenvolvido utilizando principalmente as threads como técnica de programação.

6. CONCLUSÃO

O presente trabalho teve como objetivo principal fazer uma análise comparativa das técnicas de programação paralela por threads e forks. A comparação partiu de uma simulação realizada a partir de um algoritmo de multiplicação de matrizes, desenvolvido na forma serial e paralela. Em seguida, foram realizadas médias do intervalo de tempo de execução dos programas sob diferentes possibilidades de implementação e, então, foram feitas avaliações de desempenho de ambas as técnicas.

A comparação não visou destacar a melhor técnica, e sim apresentar as vantagens que cada uma pode oferecer no desempenho final respeitando as suas restrições próprias na comunicação das informações entre os processos e no formato de programação. A partir disto, foi possível estabelecer o melhor uso das técnicas de forks e threads em processamento das imagens de fusão nuclear, com o objetivo de alcançarmos o melhor desempenho em termos de velocidade (processamento de imagens por segundo).

Uma das maiores vantagens do uso de múltiplos threads em relação ao fork é a facilidade de comunicação e compartilhamento de informações entre threads, dado que ambas trabalham no mesmo espaço de endereço de um único processo e compartilham variáveis globais [18]. A comunicação entre processos por uso de sinais é limitada e mecanismos de IPC (Inter Process Comunication) mesmo que variados (pipes, memória compartilhada, etc) impõem complexidades adicionais por vezes desnecessárias na escrita de um código. Consequentemente, criar uma nova thread torna-se mais eficiente que um novo processo, isto pode ser observado nos gráficos apresentados, já que o programa desenvolvido por thread apresenta uma menor média de tempo de execução em relação ao fork. Por outro lado, existem casos em que o programador necessita de restrições quanto ao acesso às variáveis globais, o que faz do fork a melhor opção, visto que a duplicação do programa gera variáveis globais distintas para os diferentes processos. E caso esses mesmos processos necessitem ler, escrever ou editar as mesmas informações, todos poderão fazê-lo a partir do uso da memória compartilhada.

A evolução no desempenho com o aumento do número de CPUs se manteve semelhante para as duas tecnologias, com uma pequena vantagem para o uso de threads que, além de possuir um crescimento ligeiramente mais acentuado (figura 6), apresentou variações positivas superiores àquelas do forks. Esse fato pode ser observado mais claramente na marcação de 8 CPUs da figura 6, onde se pode observar que algumas execuções de threads forneceram speedups superiores à 7. Nota-se, no mesmo ponto da figura 6, que as execuções mais lentas de ambas as tecnologias apresentaram speedups semelhantes, consequentemente os tempos de execução das threads continuaram inferiores aos dos forks, visto que o tempo inicial de cálculo do speedup das threads é inferior ao tempo inicial das forks.

Além de sustentar uma taxa de crescimento de desempenho ligeiramente superior, observado na figura 6, ao apresentado pelos forks, os cálculos em threads se iniciaram e mantiveram com tempos de execução inferiores aos forks, figura 7, evidenciando seu melhor desempenho final.

Com relação às técnicas de paralelização, pode-se concluir que o custo computacional para promover o paralelismo em threads é inferior ao das forks, permitindo um melhor desempenho dos aplicativos que fazem uso dessa tecnologia.

No entanto, ambas as técnicas estão limitadas a quantidade de processadores presentes em um único computador. É muito comum atualmente termos a disposição clusters computacionais com diversos processadores distribuídos por vários computadores. O primeiro passo para a continuidade do presente trabalho poderá focar na implementação do processamento paralelo em ambientes de cluster de computadores, i.e., partir da atual simulação de paralelismo em apenas um computador com processador multicore para um conjunto de computadores multicores conectados em rede. Este processamento distribuído poderia ser desenvolvido utilizando por exemplo a linguagem Charm++ [19], que é uma linguagem paralela de troca de mensagem eficiente que pode ser dimensionada para uma grande variedade de máquinas.

7. ANEXO

Os códigos desenvolvidos neste trabalho estão disponíveis na linguagem C na página eletrônica do CBPF na seguinte URL: http://www.cbpf.br/~fdutra/nt201201/

[1] P. E. McKenney; "'Real Time' vs. 'Real Fast': How to Choose?"; Proceedings of the 11th Linux Symposium - Dresden, Germany, Sept. 2009

[2] K. Koolwal; "Myths and Realities of Real-Time Linux Software Systems"; Proceedings of the 11th Linux Symposium - Dresden, Germany, Sept. 2009

[3] Gambier, A.; "Real-time control systems: a tutorial"; 5th Asian Control Conference, vol.2, pp. 1024- 1031, July; 2004

[4] C. HUGHS e T. HUGHES.; "Object Oriented Multithreading using C++: architectures components"; vol 1, John Wiley Sons, 2 ed., 1997.

[5] M. Tim Jones; "Anatomy of Linux Kernel Shared Memory:

Memory de-duplication in the Linux kernel"; Disponível em: http://www.ibm.com/developerworks/linux/library/l-kernel-shared-memory/index.html. Acesso em: 22/03/2012.

[6] IBM Corporation; "UNIX-Type – Interprocess Communication (IPC) APIs"; 2006; Disponível em: http://publib.boulder.ibm.com/infocenter/ iseries/v5r4/topic/ apis/unix3.pdf. Acesso em : 21/03/2012.

[7] E. W. Dijkstra; "The structure of the \"THE\"-multiprogramming system"; Communications of the ACM, vol.11, Issue: 5; May/1968, pag. 341-346. DOI=10.1145/363095.363143

[8] IBM Corporation, "UNIX and Linux signal handling", 1999, Disponível em: http://publib.boulder.ibm.com/infocenter/wmqv7/v7r1/topic/ com.ibm.mq.doc/fg11910_.htmAcesso em : 2/03/2012.

[9] M. Tim Jones, "Boost socket performance on Linux: Four ways to speed up your network applications", 2006, Disponível em : http://www.ibm.com/developerworks/linux/library/l-hisock/index.html. Acesso em : 22/03/2012.

[10] B. Barney (Lawrence Livermore National Laboratory), "POSIX Threads Programming". Disponível em : https://computing.llnl.gov/tutorials/pthreads/#PthreadsAPI. Acesso em : 2/03/2012.

[11] G.M. Amdahl, "Validity of the single-processor approach to achieving large scale computing capabilities", In Proceedings of the April 18-20, 1967, Spring joint Computer Conference (AFIPS 1967 (Spring)). ACM, New York, NY, EUA, pag. 483-485. DOI=10.1145/1465482.1465560.

[12] Linux man page web site: http://linux.die.net/man/3/clock_gettime. Acesso em : 13/03/2012.

[13] B. Barney (Lawrence Livermore National Laboratory), "Introduction to Parallel Computing". Disponível em : https://computing.llnl.gov/tutorials/parallel_comp/. Acesso em : 05/04/2012.

[14] Murari, A.; Camplani, M.; Cannas, B.; Mazon, D.; Delaunay, F.; Usai, P.; Delmond, J.F.; , "Algorithms for the Automatic Identification of MARFEs and UFOs in JET Database of Visible Camera Videos,"Plasma Science, IEEE Transactions on , vol.38, no.12, pp.3409-3418, Dec. 2010

[15] M. Portes de Albuquerque, M. P. de Albuquerque, G. Chacon, E.L. de Faria, A. Murari and JET EFDA contributors. "*High Speed Image Processing Algorithms for Real TimeDetection of MARFEs on JET.* Disponível em: http://www.iop.org/Jet/article?EFDP11031&EFDP11031. Acesso em: 23/03/2012.

[16] M. Portes de Albuquerque, Marcelo P. de Albuquerque; G. Chacon; E. Gastardelli; F. D. Moraes e G. Oliveira, "Aplicação da técnica de momentos invariantes no reconhecimento de padrões em imagens digitais", NT/CBPF - CBPFIndex -006/2011

[17] Ref. McKenney; "When Do Real Time Systems Need Multiple CPUs?" Proceedings of the 11th Linux Symposium - Dresden, Germany, Sept. 2009.

[18] Stevens, W. R.; Rago, S. A. : Advanced Programming in the Unix Environment. 2005.

[19] "Parallel Languages/Paradigms: Charm ++ - Parallel Objects", Parallel Programming Laboratory - DCS/Univ. of Illinois. Disponível em: http://charm.cs.uiuc.edu/research/charm. Acesso em: 26/03/2012.