Revista de Sistemas de Informação da FSMA
n. 9 (2012) pp. 45-53

SISTEMAS DE INFORMAÇÃO
http://www.fsma.edu.br/si/sistemas.html

TRILHA PRINCIPAL

Um Estudo Comparativo do Desempenho de Aplicações Paralelizadas em Cenários Distintos

Alex G. C. de Sá, *Mestrando em Ciência da Computação, UFMG*,
Marluce R. Pereira, *Professora do Departamento de Ciência da Computação, UFLA*,
Pedro M. Moura, *Mestrando em Ciência da Computação,UNICAMP* e
Luiz Henrique R. Peixoto, *Mestrando em Ciência e Tecnologia da Computação, UNIFEI*

***Resumo*—Este artigo apresenta um estudo comparativo do desempenho de aplicações paralelizadas com *threads* em cenários distintos. Essas aplicações utilizam o conceito de *threads* para solucionar tarefas. Para compor cada cenário a ser avaliado foram considerados quatro fatores: plataforma (ou biblioteca) de programação paralela com *threads*, sistema operacional, arquitetura do sistema operacional e a prioridade de execução de processos. Foi analisado o ganho de desempenho atingido com aplicações matemáticas em um processador multinúcleo. O tempo de execução e o número de operações foram os parâmetros utilizados nos comparativos de desempenho. Resultados mostraram que os fatores contemplados são realmente relevantes para o paralelismo inerente às arquiteturas multinúcleo.**

***Palavras-chave*—Desempenho, Paralelismo, Comparativo, Multinúcleo, Aplicações Paralelizadas.**

A Comparative Study of Multithreaded Applications Performance in Different Scenarios

***Abstract*—This paper presents a comparative study of the performance of multithreaded applications in different scenarios. These applications use the concept of threads to solve tasks. To make each scenario being evaluated were considered four factors: threads programming platform (or library), operating system, operating system architecture and the priority of running processes. We analyzed the performance obtained in mathematical applications in a multicore processor. The runtime operations and the number of parameters were used in comparative performance. Results showed that the factors referred to are actually relevant to the parallelism inherent to multicore architectures.**

***Index Terms*—Performance, Parallelism, Comparison, Multicore, Multithreaded Applications.**

I. Introdução

A computação paralela e distribuída surgiu com o objetivo principal de diminuir o tempo de resolução de problemas de alto custo computacional que antes possuíam um tempo inviável de processamento, devido a sua complexidade, seja ela temporal ou espacial. Áreas de pesquisa em previsão temporal, matemática, bioinformática, química e física são exemplos de desafios atuais que demandam esforços em processamento computacional. Essas áreas fazem uso de modelos algorítmicos para tratar problemas complexos, que são aqueles que exigem imensas quantidades de cálculos. Nesse caso, o tempo de retorno da solução do problema, em questão, pode ser citado como um dos pontos-chave das pesquisas.

Porém, se os problemas científicos demandam cada vez mais processamento, e consequentemente um maior tempo de resposta, em contrapartida há uma dificuldade em manter a Lei de Moore [9], que previa que os processadores iriam dobrar sua capacidade de processamento a cada 18 meses. O motivo para isso é que existem limitações físicas na redução dos transistores dentro do encapsulamento dos processadores. Essas limitações fazem com que a indústria não consiga aumentar o potencial dos processadores, pois não se consegue transpor a barreira do tamanho dos transistores e isso consequentemente resulta na estabilização da tecnologia dos processadores (quanto a sua capacidade de processamento).

A fim de contornar a limitação no processo de fabricação dos *chips*, a alternativa encontrada foi simplesmente multiplicar os núcleos de processamento do processador. Contudo, com o advento destas arquiteturas paralelas, há cada vez mais a necessidade de aplicar os conceitos de programação paralela com o intuito de utilizar-se ao máximo a capacidade de processamento dessas plataformas.

Existem basicamente duas formas de obter desempenho implementando aplicações concorrentes. A primeira abordagem é por meio de criação de processos, ponderando-se que todo o tipo de comunicação é feita por meio de mensagens, já que essas são responsáveis pela manutenção de todas as informações necessárias à execução de um programa, como conteúdo de registradores e espaço de memória [4]. Já a segunda abordagem é utilizando-se de *threads*, que também são conhecidas por processos leves [12]. Essas trocam informações apenas por meio de memória compartilhada e podem ser até 50 vezes mais ágeis na fase de criação em comparação com processos [1].

Este trabalho apresenta um estudo e análise da obtenção de desempenho de aplicações paralelas a partir da segunda abordagem, ou seja, aquela que faz uso de *threads*. A principal motivação deste estudo é conhecer quais são

Autor correspondente: Alex G. C. de Sá, alexgcsa@dcc.ufmg.br

os fatores que influenciam no ganho ou perda de desempenho de aplicações paralelas com múltiplas *threads* (*multithreading*). Foram estudados quatro fatores principais: diferenças entre sistemas operacionais (Windows *versus* Linux), diferenças entre arquiteturas para sistemas operacionais (32 bits *versus* 64 bits), utilização de bibliotecas de manipulação de *threads* distintas (PThreads *versus* WinAPI) e modificação de prioridade de escalonamento de processos (prioridade comum *versus* prioridade máxima). Desse modo, será entendido qual é a melhor das formas de modelar problemas de alto custo computacional e quais dos fatores externos ou internos à aplicação que afetam de forma mais acentuada em seu desempenho.

Esses fatores podem acentuar ou diminuir o desempenho de uma aplicação concorrente, mas isso depende do escopo de funcionamento de tal aplicação. Portanto, a justificativa deste trabalho torna-se clara ao analisar os diversos tipos de aplicações sob essa perspectiva.

Um ponto que merece ser ressaltado é que a WinAPI, em suas funcionalidades de manipulação de *threads*, fornece operações similares as operações da POSIX-Threads. Contudo, as implementações não podem ser comparadas, visto que o código fonte da WinAPI é proprietário. A falta de formas de comparação do código fonte justifica (e motiva) uma análise experimental de aplicações que utilizam o paradigma de programação concorrente sobre ambas interfaces de programação para o desenvolvimento de tais aplicações.

A estruturação do artigo é feita da seguinte forma. A Seção I introduziu o trabalho. Para entender como foram modeladas as aplicações concorrentes, a Seção II explora características relevantes da biblioteca PThreads e da plataforma de programação WinAPI. Já a Seção III relata trabalhos relacionados a esse artigo. A Seção IV apresenta a metologia utilizada, ou seja, os problemas que serão utilizados para a análise de desempenho das bibliotecas e as principais decisões de projeto. A Seção V apresenta resultados com uma breve discussão para cada situação. E a Seção VI encerra o artigo com as conclusões obtidas e possíveis desafios para um futuro trabalho.

II. Programação com Threads

ESTA seção apresenta a biblioteca PThreads e a plataforma de programação WinAPI. Isso é feito para que se compreenda como as aplicações utilizadas neste trabalho foram modeladas e desenvolvidas.

A. PThreads

A ideia principal da utilização de *threads* consiste em dividir um programa único em várias tarefas paralelas, para que o trabalho mais pesado seja feito de forma mais rápida [8]. Atualmente, o padrão POSIX para *threads* no Linux é comumente utilizado a partir da biblioteca PThreads (POSIX-Threads). Essa biblioteca provê uma interface que cria e manipula *threads*, que executam sobre um programa [1]. A PThreads é padronizada para a linguagem C.

Quando a biblioteca PThreads foi criada, seus desenvolvedores buscaram manter os princípios delineados pelos desenvolvedores de *kernel* (núcleo) de sistemas UNIX, incorporando-os à biblioteca. Um desses princípios é, por exemplo, a troca de contexto entre processos relacionados, que deve ser rápida o suficiente para lidar com cada segmento de *thread* em nível de usuário em uma *thread* de *kernel*. Processos de *kernel* podem ter diversos níveis de relacionamento, entretanto a especificação da PThreads exige compartilhamento entre quase todos os recursos [5].

Além de fornecer uma interface, o padrão PThreads especifica vários serviços para poder dar suporte a aplicações *multithreading*, tais como suporte a certas funções, detecção de alguns erros e funcionalidades de gerenciamento [10]. Contudo, existem muitos serviços que são opcionais à implementação da biblioteca, o que pode deixar certas implementações bem distintas umas das outras. Na Figura 1, são apresentadas as camadas de software para uma aplicação que utiliza PThreads, a partir de um diagrama em que o processo de comunicação entre aplicação, biblioteca PThreads e sistema operacional são definidos.

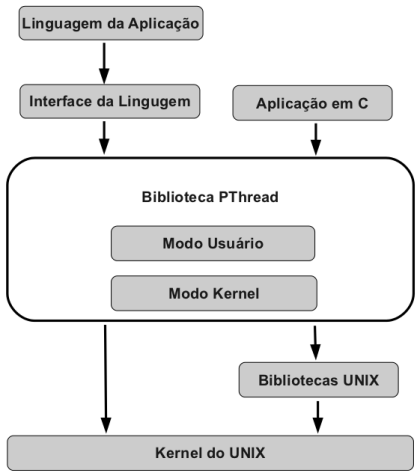

Fig. 1. Camadas de software da PThreads para uma aplicação [10].

De acordo com a Figura 1, somente a programação feita na linguagem C pode usar diretamente as rotinas da biblioteca PThreads. Qualquer outra linguagem de programação necessita de uma interface para a PThreads para passar os parâmetros corretamente, executar conversão de tipos e outros ajustes dependentes da linguagem ou do compilador. A PThreads contém um conjunto de rotinas cuja sua interface e funcionalidade são definidos pelo padrão POSIX-Threads. O código das rotinas da PThreads executa parcialmente em modo usuário e, dentro de seções críticas, opera em modo *kernel*. Isso garante exclução mútua entre as *threads*. A sua implementação utiliza uma série de bibliotecas padrões do UNIX e também de suas chamadas de *kernel*.

É importante ressaltar que a biblioteca PThreads define funções, tipos e constantes na linguagem C. Nesse trabalho, foi utilizado o cabeçalho *pthread.h*.

B. WinAPI

A *Windows Application Programming Interface* (WinAPI) é uma plataforma de programação fornecida para todas as versões do sistema operacional Windows. A sua implementação é proprietária com os direitos reservados à empresa Microsoft.

De acordo com *Bodnar* [2], a WinAPI permite por meio de sua interface de código-fonte criar diversos tipos de aplicações. Essa plataforma é criada especialmente para as linguagens de programação C e C++. Além disso, é o modo mais direto para criar aplicações do Windows.

A Figura 2 apresenta os quatro componentes básicos da WinAPI.

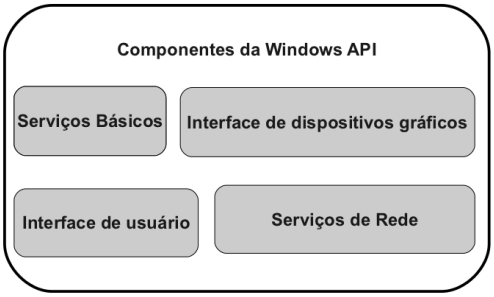

Fig. 2. Quatro componentes básicos da WinAPI [2].

Na Figura 2, os Serviços Básicos proveem acesso aos recursos fundamentais no Windows. Esses recursos incluem manipuladores de sistemas de arquivos, dispositivos, processos, *threads*, registro ou erros. A Interface de Dispositivos Gráficos (GDI, do inglês "*Graphics Device Interface*") é uma interface para trabalhar com gráficos. Ela é usada para interagir com dispositivos gráficos, como impressora, monitor ou um arquivo. A Interface de Usuário fornece funcionalidades para criar janelas e controles requisitados por uma aplicação. Os Serviços de Rede fornecem acesso aos recursos de rede do sistema operacional Windows.

Complementando a última literatura, *Spinellis* [14] relata que as funcionalidades da WinAPI podem ser divididas em oito categorias:

- Administração e gerenciamento
- Diagnósticos
- Gráficos e multimídia
- Redes
- Segurança
- Serviços do sistema
- Interface com o usuário

A funcionalidade utilizada neste trabalho é a de *threads* e processos que se encaixam dentro da categoria de serviços do sistema.

III. Trabalhos Relacionados

O trabalho de *Silva e Yokoyama* [13] tem o objetivo de comparar o desempenho de bibliotecas de *threads*. As bibliotecas comparadas foram a GNU Pth [7], a Protothreads [6] e a PM2. Marcel [16]. Durante o comparativo realizado no trabalho relatado, funções básicas de gerenciamento, sincronização, inicialização e união do trabalho das *threads* foram avaliadas. Operações que demandam muita/pouca computação de CPU e pouca/muita de Entrada e Saída (CPU Bound/IO Bound) também foram analisadas. Contudo, o desempenho das bibliotecas de manipulação de *threads* foi somente medido a partir de médias de tempo de execução, variando-se o número de operações e a quantidade de *threads* do problema. Desse modo, não é analisado nessa literatura o paralelismo real obtido por arquiteturas multinúcleo. Isso poderia ser feito a partir do uso de métricas de desempenho de programas, como o *speedup*. Essa é diferença primordial do trabalho de *Silva e Yokoyama* para o presente estudo, feito por este artigo.

Já *Torelli e Bruno* [17] possuem uma proposta semelhante ao deste trabalho. É descrito no texto uma comparação entre duas ferramentas para programação paralela, o OpenMP e a PThreads. Os testes foram realizados em um multiprocessador simétrico ou SMP (do inglês "*Simetric Multiprocessor*") que possui quatro processadores. Essa arquitetura é do tipo paralela e possui memória compartilhada. Um algoritmo de transformada de distância euclidiana, que é um algoritmo utilizado em processamento de imagens, foi utilizado para comparar os resultados. A ideia principal desse trabalho relacionado foi avaliar e comparar o tipo de programação com *threading* explícito (PThreads) e o tipo de programação baseada em diretivas (OpenMP). Os resultados mostraram um menor tempo de execução para a ferramenta PThreads. Além disso, foi visto que com o aumento do tamanho da imagem a ser analisada, maior o desempenho alcançado pela ferramenta.

A diferença do trabalho de *Torelli e Bruno* para o presente artigo é que no primeiro não foram analisados e comparados diferentes aplicações e nem mesmo outros cenários (sistemas operacionais e arquiteturas distintas). Somente foram realizadas variações no tamanho da imagem na qual o algoritmo de transformada de distância euclidiana é aplicado. Existem diversos tipos de aplicações e outros cenários em que resultados poderiam ser distintos. No presente artigo, esses pontos são amplamente discutidos na Seção V.

Em *Penha et al.* [11] é apresentada uma avaliação do desempenho de diferentes paradigma e linguagens de programação utilizando o tipo de programação *multithreading*. Também foi feito um estudo utilizando algoritmos de processamento digital de imagens. Nessa literatura comparou-se tanto os paradigmas procedural e orientado a objetos, quanto as linguagens de programação C++ e Java, relevando-se sempre o desempenho e a facilidade de desenvolvimento. Observou-se no trabalho em questão que o paradigma procedural obteve melhores resultados que o orientado objetos, considerando somente a linguagem C++. Outro ponto importante a ser ressaltado foi que o desempenho obtido pela linguagem Java foi superior ao C++ orientado a objetos.

A distinção para este trabalho em relação a última literatura relatada é que o presente artigo não se preocupou em avaliar diferentes paradigmas e linguagens de

programação. Utilizou-se somente a linguagem C dentro do paradigma procedural. Muito embora esse estudo seja bastante interessante, pois atualmente o conceito de arquiteturas multinúcleo não foi totalmente explorado pelos paradigmas de programação. Mesmo que o trabalho de *Penha et al.* tenha estudado diferentes linguagens e paradigmas, outros fatores relevantes não foram contemplados. Exemplos desses fatores são: o uso e estudo de diferentes interfaces de programação *multithreading* para linguagem C (ou outra linguagem, no caso); diferenças de desempenho de aplicações paralelas em sistemas operacionais distintos; estudo do impacto da mudança de arquitetura de sistema operacional; e a distinção de tempos de execução e quantidade de operações realizadas quando se muda a prioridade de escalonamento de um processo, o qual é orquestrado por uma aplicação do tipo *multithreading*.

Portanto, releva-se a importância desse trabalho, que além de observar atentamente o paralelismo real inerente de aplicações que executam sobre arquiteturas de processador multinúcleo, também analisa-se os diversos fatores que influenciam no desempenho de tais aplicações.

IV. Metodologia

BUSCANDO realizar a análise de desempenho de um conjunto de aplicações concorrentes que utilizam *threads*, foram avaliados quatro fatores. Esses fatores determinam cenários que influenciam na aquisição de desempenho por uma aplicação.

O primeiro fator constitui a diferença de sistema operacional, considerando que implementações das aplicações foram feitas tanto no ambiente Linux, quanto no Windows.

Para os dois sistemas operacionais foi feita a distinção de arquiteturas. Testes sobre as aplicações foram realizados na arquitetura de 32 bits e na arquitetura de 64 bits. A diferença de arquitetura do sistema operacional é considerada um segundo fator.

Considera-se que as aplicações possam ser compiladas e executadas nos dois sistemas operacionais, um terceiro fator teve sua relevância. Duas plataformas de programação para aplicações concorrentes foram utilizadas para isso. A plataforma WinAPI foi utilizada para Windows e a PThreads, utilizada em sistemas UNIX-*like*, como aqui feito.

E por fim, o quarto fator se distingue dos demais pois usa prioridades diferentes na execução de processos. Um processo que contém *threads* em sua estrutura será escalonado com prioridade comum (configurada pelo sistema operacional) e máxima (configurada antes da execução).

A Tabela I apresenta as configurações descritas, especificando o que constituirá cada cenário, ou seja, quais os fatores avaliados em cada um desses cenários.

Devido às incompatibilidades de padrões de desenvolvimento da versão do Gnome C Compiler (GCC) para arquitetura de 64 bits do Windows, foram realizados testes apenas para sua versão de 32 bits, mesmo a arquitetura desse sistema operacional sendo de 64 bits.

TABELA I
Cenários avaliados e seus fatores

Cenário	Sistema Oper.	Arquit.	Plataforma	Prioridade
1	Ubuntu 11.04	32 bits	PThreads	Comum
2	Ubuntu 11.04	32 bits	PThreads	Máxima
3	Ubuntu 11.04	64 bits	PThreads	Comum
4	Windows 7	32 bits	WinAPI	Comum
5	Windows 7	64 bits	WinAPI	Comun
6	Windows XP	32 bits	WinAPI	Comun

Outra questão que merece enfoque é que para avaliar a execução das aplicações nos sistemas operacionais Windows e Linux foi necessária uma adaptação das implementações das aplicações, especialmente na escolha da plataforma de manipulação de *threads* utilizada e na troca de arquitetura do sistema operacional.

O conjunto de aplicações relatado será descrito a seguir. Quatro problemas (aplicações) foram utilizados para estimar os desempenhos dos cenários apresentados: cálculo de milhões de operações inteiras por segundo (MIPS), cálculo de milhões de operações em ponto flutuante (MFLOPS), cálculo do π utilizando o método de Leibniz e o cálculo da integral definida seguindo a regra dos trapézios. Para igualar as situações dos testes realizados, todas as variáveis quantitativas para cálculos e iterações foram as mesmas, tanto para um cenário quanto para um outro qualquer. Os quatros problemas utilizados para desenvolver o comparativo de desempenho são abordados a seguir.

A. MIPS

O algoritmo MIPS (Milhões de Operações de Inteiro por Segundo) procura estimar número de operações de inteiro que uma máquina consegue calcular em razão do tempo. O problema usa todos os tipos de operações alternadamente (soma, subtração, divisão, multiplicação), realizando então a mesma carga de todos os tipos de operação. O método é paralelizado de uma maneira simples, são criadas N *threads*, cada uma dessas faz o cálculo em paralelo das operações e ao final o resultado é integrado. O número de operações nos testes realizados foi de dez bilhões (1E+10).

B. MFLOPS

O MFLOPS (Milhões de Operações de Ponto Flutuante por Segundo) é semelhante ao MIPS, com a diferença que as operações são feitas com números fracionários em notação de ponto flutuante, que possuem maior custo computacional. O número de operações utilizado foi igual a cem milhões (1E+8).

C. Método de Leibniz

O método de Leibniz utiliza uma série de Taylor, cujo somatório tende ao valor de $\frac{\pi}{4}$. Esta série é mostrada na Equação 1. O método de Leibniz foi paralelizado dividindo faixas de termos da série entre as *threads*, considerando que cada *thread* calcula um intervalo e depois estes resultados

são somados e, quando sincronizados plenamente, multiplicado finalmente por quatro. A carga de cálculo utilizada nos testes foi de um bilhão (1E+9) de termos do somatório.

$$\sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots = \frac{\pi}{4} \qquad (1)$$

D. Integral por regra dos trapézios

A regra dos trapézios é um método utilizado para ter uma aproximação do resultado de uma integral definida. A área de integração é a região sob a curva, que é definida por uma função $f(x)$ e delimitada por pontos, como a e b. O método consiste em dividir a região de integração em N trapézios com altura h, tendo a área dividida, o valor da integral é a soma da área de todos os trapézios sob a curva (Equação 2).

$$\int_b^a f(x)dx = h * \sum_{i=0}^{N-1} \left(\frac{f(x_i) + f(x_{i+1})}{2} \right) \qquad (2)$$

Para os testes, foram utilizados cem milhões (1E+8) de trapézios em cada execução. A função utilizada para o cálculo da integral foi $f(x) = x^3 + x^2 + 2x + 1$ e os pontos delimitadores da integral foram $a = 1$ e $b = 2$.

E. Outras decisões de projeto

Os códigos fonte dos problemas descritos anteriormente foram implementados utilizando linguagem de programação C e compilados com compilador GCC 4.5 em ambos sistemas operacionais. Os sistemas operacionais utilizados foram o sistema Ubuntu 11.04, o Windows XP Service Pack 3 e o Windows 7 Service Pack 1. No Linux foi utilizado o cabeçalho *pthread.h* para biblioteca PThreads e no Windows o *windows.h* foi utilizado para invocar os métodos de *threads* da plataforma WinAPI. O computador utilizado nas execuções foi um AMD Phenom™ II X4 B95 com 2,99 GHz de frequência e 3,00 GB de memória RAM.

Para que o código desenvolvido controle de forma correta a quantidades de operações/iterações especificadas nas subseções anteriores, a quantidade de *threads* e a prioridade do processo como máxima (ou normal), foram criados *scripts* manipulando os executáveis criados a partir da compilação das aplicações, ajustando cada um de seus parâmetros. Laços de repetição também foram codificados a fim de se realizar a execução dos testes diversas vezes.

Para escalonar um processo com prioridade máxima de execução no Linux utilizou-se o seguinte comando:

nice -n -20 EXECUTÁVEL <PARÂMETROS>.

O comando *nice* configura a prioridade, determinando que o executável da aplicação gere um processo com prioridade mais alta ou mais baixa desde o início da execução do processo. Nesse caso, utilizou-se prioridade igual a -20, a prioridade mais alta do processo em sistemas UNIX-*Like*. A prioridade nesse tipo de sistema está dentro do intervalo de -20 a 19.

Já no Windows, foi feito o uso do seguinte comando para determinar a prioridade máxima do processo:

start /high EXECUTÁVEL <PARÂMETROS>.

O comando *start* pode iniciar um processo com uma prioridade específica. Essa prioridade permanecerá até o fim da execução do processo. No caso de todas as aplicações desenvolvidas, o parâmetro *high* foi utilizado e esse determina que o processo da aplicação executará com prioridade alta. O Windows é limitado quanto às suas classes de prioridades, permitindo os seguintes tipos: *low* (baixa), *bellownormal* (abaixo do normal), *normal* (normal), *abovenormal* (acima do normal), *high* (alta) e *realtime* (tempo real).

V. Resultados e Discussão

CADA problema foi executado 33 vezes com uma, duas, três e quatro *threads* simultâneas e, em seguida, calculada a média das execuções, como é mostrado nas figuras de 3 a 6. Esse número de execuções foi escolhido para obtenção de resultados estatísticos confiáveis. Já o limite do número de *threads* foi igual a quatro, pois esse valor é igual ao número de núcleos do processador utilizado para os testes. Desse modo, o paralelismo real inerente a esse tipo de arquitetura de processador poderá ser melhor estudado.

Para os gráficos das Figuras 3, 4, 5 e 6, as legendas possuem as seguintes definições da Tabela I. Cada cenário dessa tabela conterá um fator diferente a ser avaliado por esse trabalho.

As Tabelas II, III, IV e V também possuem as mesmas legendas para suas linhas. Suas colunas significam o *speedup* alcançado para cada número de *threads*, ou seja, o ganho de desempenho utilizando duas, três e quatro *threads* em relação à forma sequencial com apenas uma *thread*.

Para todos os gráficos, os resultados são baseados em médias. Por essa razão, são considerados nos gráficos os intervalos de confiança de 95% para os mesmos. Isso foi feito para medir a precisão das mensurações realizadas, ou seja, 95% das vezes que um dos problemas forem executados, esses apresentarão o mesmo resultado, considerando as margens dos intervalos de confiança. Isso será ilustrado em cada barra do gráfico por uma outra barra de cor preta, delimitando o máximo e mínimo desse intervalo.

Os resultados mostraram que as execuções tiveram desvio padrão baixo, considerando as baixas margens dos intervalos de confiança. Houve uma acentuação do desvio padrão e, consequentemente, nas margens dos intervalos de confiança nas execuções do MFLOPS em todos os cenários de execução. Esse acentuação do desvio padrão é vista com mais ênfase em todos os cenários com três e quatro *threads*, o que indica um problema de escalonamento das *threads* pelo processo (*thread* de nível de usuário) a medida que a quantidade de linhas de execução vão aumentando. Isso pode ser explicado pela quantidade e tipo de operação (ampla precisão) em ponto flutuante que ocorre no MFLOPS. No gráfico da Figura 4 essa acentuação é percebida com mais detalhes.

Na Figura 3, em todos os cenários de execução os resultados mostraram que ao aumentar o número de *threads*

o número de MIPS executadas também aumentaram. O **Cenário 6** apresentou cerca de 5,32% de MIPS a mais em relação aos demais cenários.

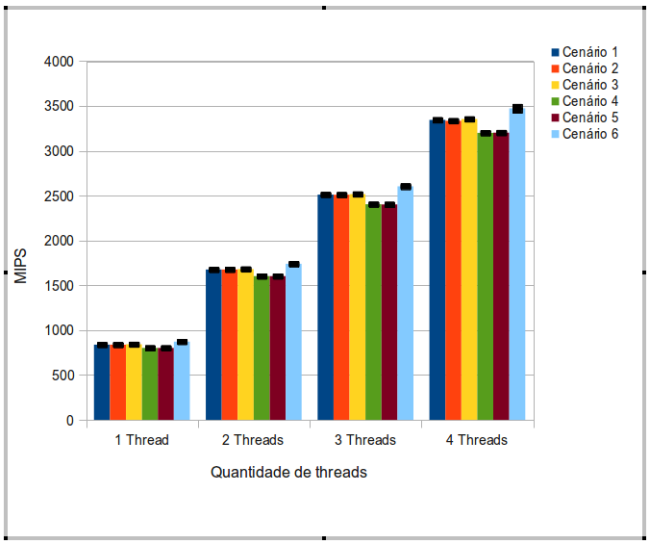

Fig. 3. Resultados para MIPS.

Em termos de *speedup*, ao aumentar o número de *threads* também houve ganho de desempenho para todos os cenários avaliados e na mesma proporção. Isso pode ser visto na Tabela II. Neste caso, nenhum cenário é beneficiado.

TABELA II
SPEEDUP PARA MIPS

MIPS	P/ 2 threads	P/ 3 threads	P/ 4 threads
Cenário 1	1,99836	2,99522	3,98812
Cenário 2	1,99999	2,99439	3,97763
Cenário 3	1,99400	2,98604	3,97917
Cenário 4	1,99767	2,99662	3,98881
Cenário 5	1,99826	2,99513	3,99230
Cenário 6	1,99440	2,98423	3,98073

Para os resultados presentes na Figura 4, pode ser analisado que ao aumentar o número de *threads*, o número de MFLOPS executados também aumentou em todos os contextos analisados. Porém, o **Cenário 3** realizou a execução de mais de 7,25% de MFLOPS do que os demais cenários.

Ainda de acordo com a Figura 4, apesar de **Cenário 5** ter apresentado o segundo maior número de MFLOPS executados, o *speedup* (Tabela III) que o primeiro alcançou foi maior do que **Cenário 3**. Mesmo assim, como ambos os sistemas se utilizam da arquitetura de 64 bits, esses se beneficiam para a execução desta aplicação, pois essa possui diversas variáveis que durante a execução do problema realizam operações com números em notação de ponto flutuante, que nessa arquitetura tem um melhor desempenho.

Em relação à aplicação MFLOPS também pode ser analisado que ocorreram exceções que de alguma forma

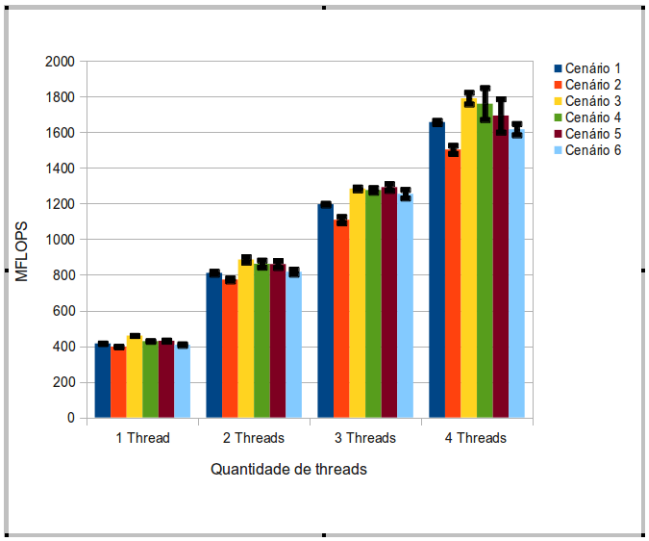

Fig. 4. Resultados para MFLOPS.

influenciaram seu desempenho, visto que o *speedup* alcançado por essa aplicação não foi estável. O *speedup* mais estável alcançado foi no **Cenário 6**. No **Cenário 4** para quatro *threads* e no **Cenário 6** para três *threads* foram percebidos *speedups* super-lineares. A Tabela III apresenta as características de *speedup* para a aplicação MFLOPS.

TABELA III
SPEEDUP PARA MFLOPS

MFLOPS	P/ 2 threads	P/ 3 threads	P/ 4 threads
Cenário 1	1,95273	2,87901	3,98785
Cenário 2	1,94719	2,79152	3,78374
Cenário 3	1,92784	2,79359	3,89541
Cenário 4	2,01294	2,97793	4,10829
Cenário 5	2,00000	3,00323	3,93656
Cenário 6	2,00283	3,07327	3,96040

No problema do cálculo do π utilizando o método de Leibniz, houve ganho de desempenho para duas, três e quatro *threads*. O intervalo de confiança também foi baixo, demonstrando uma certa estabilidade durante as execuções. Isso pode ser visto na Tabela IV, onde o *speedup* alcançado foi sempre próximo ao limite da capacidade do processador de quatro núcleos.

TABELA IV
SPEEDUP PARA LEIBNIZ

Leibniz	P/ 2 threads	P/ 3 threads	P/ 4 threads
Cenário 1	1,98469	2,97873	3,95099
Cenário 2	1,98974	2,98022	3,97332
Cenário 3	1,98717	2,97128	3,95378
Cenário 4	1,99813	2,99383	3,98998
Cenário 5	1,99771	2,99461	3,98846
Cenário 6	2,00055	3,00082	3,99713

Já em relação ao gráfico da Figura 5 pondera-se que pelo tipo de operações do método de Leibniz serem similares aos

do MFLOPS (notação de ponto flutuante), é justificado o menor tempo de execução no **Cenário 3**.

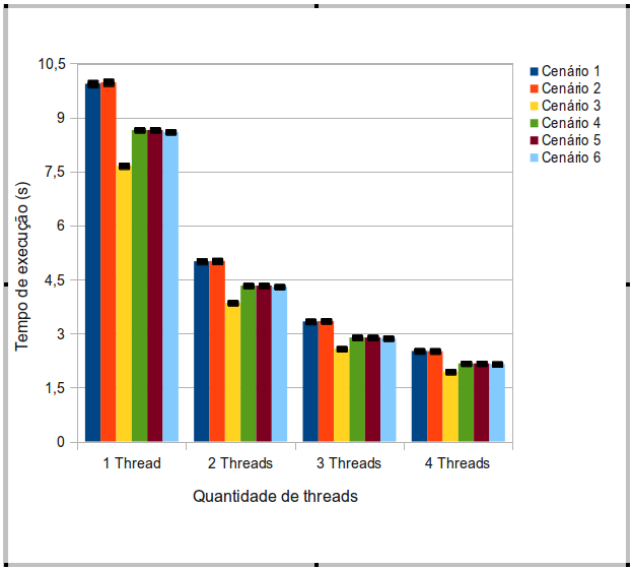

Fig. 5. Resultados para Leibniz.

Para a aplicação da solução da integral por regra dos trapézios o comportamento de todas execuções foram praticamente iguais em todos os cenários avaliados. Como pode ser visto no gráfico da Figura 6, ocorreram poucas alterações de comportamento quando as execuções foram feitas.

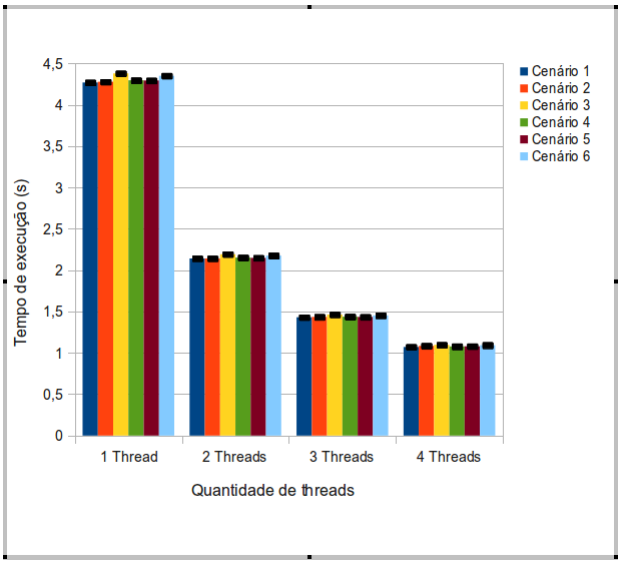

Fig. 6. Resultados para Integral.

Considerando os resultados de *speedup* da Tabela V, houve ganhos de desempenhos com duas, três e quatro *threads*. Percebe-se por essa tabela que os resultados também ficaram próximos ao limite da capacidade do processador de quatro núcleos. Caso ocorra exceções quanto ao desempenho atingido nessa aplicação, as mesmas podem estar relacionadas ao escalonamento do sistema operacional.

TABELA V
SPEEDUP PARA INTEGRAL

Integral	P/ 2 threads	P/ 3 threads	P/ 4 threads
Cenário 1	1,99378	2,98423	3,98439
Cenário 2	1,99809	2,98152	3,94500
Cenário 3	1,99736	2,99384	3,99304
Cenário 4	1,99674	2,98969	3,98805
Cenário 5	1,99824	2,99309	3,97724
Cenário 6	1,99913	2,99543	3,98352

Analisando também os resultados do **Cenário 1** e do **Cenário 2** percebe-se que em quase todas as execuções dos quatro problemas o desempenho do **Cenário 2** foi inferior. A diferença entre ambos cenários é que o segundo se utiliza de prioridade máxima de processo, enquanto o primeiro utiliza prioridade configurada pelo sistema operacional. Uma possível explicação está presente no trabalho de *Carissimi et al.* [3]. Nessa literatura é relatado que em muitos casos forçar uma aplicação ter sempre prioridade máxima de execução não é o ideal. Como essa prioridade é configurada pelo usuário do sistema, o mesmo pode comprometer o desempenho de uma aplicação, visto que não considera a fila atual de escalonamento e outros fatores importantes do sistema operacional que exploram as características de sua própria arquitetura de *hardware*.

Considera-se também que o tipo de *thread* utilizada nesse trabalho é *thread* de usuário, que é controlada pelo processo [15], o que reforça essa explicação.

Portanto, se uma aplicação que cria quatro *threads* (situação deste trabalho) no escopo de um processo, sendo que o usuário configura o mesmo a ter sempre prioridade máxima de execução, pode ocorrer que quando um dos núcleos do processador é liberado, uma *thread* que está ociosa é imediatamente escalonada para executar suas tarefas dentro desse núcleo, não obedecendo um padrão da fila de escalonamento dos processos do sistema e também a fila de escalonamento das *threads*, controlada pelo processo. Desse modo, se a *thread* citada estava executando anteriormente em outro núcleo do processador ou era necessário um outro processo entrar em execução automaticamente no núcleo atual, essa troca de contexto com relação ao núcleo de processamento do processo ou da *thread* pode aumentar a latência de execução de um processo como um todo, porque essa nova condição pode afetar a configuração padrão do sistema operacional (descritores da memória virtual, filas do escalonador e outras estruturas de dados da mesmo), comprometendo assim seu desempenho.

Se o usuário desenvolvedor deixa a responsabilidade do escalonamento para o sistema operacional, este com uma maior supervisão e previsão dos eventos do sistema pode esperar o término da tarefa daquele processador que o processo ou *thread* está esperando para executar e esse tempo de espera pode ser menor que o tempo de troca de contexto com relação ao núcleo de processamento. Isso pode diminuir o tempo de execução, aumentando assim o desempenho.

Para validar esse resultado e sua explicação, novos testes

foram realizados em todos os cenários, com exceção do **Cenário 2**, que já apresenta um resultado com prioridade máxima de execução. A Tabela VI apresenta o quão melhor ou pior são os resultados para os cenários considerando suas execuções com prioridade máxima e com prioridade comum.

TABELA VI
RESULTADOS COM PRIORIDADE MÁXIMA EM RELAÇÃO À PRIORIDADE COMUM DE EXECUÇÃO DO PROCESSO

Cenário	MIPS	MFLOPS	Leibniz	Integral
1	<0,08971%	<6,87317%	<0,15905%	<0,42007%
3	<0,27061%	<3,87829%	<0,16772%	<0,81466%
4	<0,19856%	<2,73659%	>2,25321%	>0,10840%
5	<0,23265%	<2,22516%	>2,22593%	>0,04917%
6	<0,37546%	<0,09574%	>0,18584%	>0,06284%

Os resultados presentes na Tabela VI denotam que o uso de prioridade pode aumentar o desempenho de aplicações paralelas, mas isso depende do sistema operacional utilizado e da própria aplicação, como percebido nos resultados da tabela. Apenas as aplicações do método de Leibniz e da Integral conseguiram obter desempenho maior configurando a prioridade máxima que no caso do próprio sistema operacional gerenciando esse fator. Para essas duas aplicações, desempenhos superiores foram atingidos no Windows. Já em todos os outros demais casos, o desempenho da aplicação utilizando prioridade máxima de execução do processo não auxiliou a aplicação a obter um maior desempenho, como explicado nos parágrafos anteriores.

VI. Conclusões

NESTE trabalho foi realizado um estudo de análise de desempenho de aplicações com características paralelas em diferentes cenários. Cada cenário que foi avaliado considerou uma gama de fatores que podem determinar o desempenho de tais aplicações. Os fatores estão relacionados com a mudança de sistema operacional, a diferença de arquitetura de sistema operacional, a distinção entre plataformas ou bibliotecas de programação paralela e a configuração da prioridade de execução dos processos, que são orquestrados pelas aplicações. Essas aplicações utilizadas estão no contexto matemático e manipulam amplamente do conceito de *threads* para solucionar seus cálculos.

Os resultados mostraram que os fatores estudados são relevantes e condicionam as aplicações a alcançarem um desempenho adequado para uma arquitetura multinúcleo. Isso pode ser percebido pelos gráficos de tempo de execução e número de operações, e também pelas tabelas *speedup* adequado para aplicações paralelas, que na maioria dos casos obedeceu a um desempenho esperado.

Além disso, foi visto que o tipo da aplicação também favorece o desempenho de alguns cenários, ou seja, as classes de estruturas utilizadas por umas aplicação pode possivelmente favorecer ou desfavorecer seu desempenho. Por exemplo, as aplicações que manipulam grandes quantidades de dados em ponto flutuante com precisão dupla são favorecidos pela arquitetura de 64 bits.

Em relação à prioridade de execução do processo, a qual foi estudada, pode-se afirmar que na maioria dos casos influenciou negativamente o desempenho da aplicação, com algumas exceções que depende unicamente do tipo da aplicação e das características do sistema operacional.

Como trabalho futuro pretende-se estudar outros fatores que possam influenciar o desempenho de aplicações do tipo *multithreading*. Como visto nos trabalhos relacionados, a quantidade de iterações ou de operações pode ser considerado um fator, pois afeta o *speedup* atingido pela aplicação. Outro ponto que poderia esclarecer algumas questões é a diferença de arquitetura do processador. Assim, o uso de um tipo de arquitetura de sistema operacional poderia consequentemente trazer benefícios quanto ao desempenho das aplicações, tratando diferentes arquiteturas de processadores multinúcleo.

Além disso, também estão previstos estudos usando aplicações de *benchmarks* conhecidos para que se possa determinar com maior exatidão a influência de fatores em aplicações paralelizadas.

Agradecimentos

Os autores desse artigo agradecem ao Departamento de Ciência da Computação da Universidade Federal de Lavras (DCC/UFLA) pelo apoio técnico a esse trabalho.

Referências

[1] BARNEY, B. "POSIX Threads Programing". Lawrence Livermore National Laboratory. 2011. Disponível em: <https://computing.llnl.gov/tutorials/pthreads>. Acesso em: 08/01/2012.

[2] BODNAR, J. "The WinAPI (C Win32 API, No MFC) tutorial". 2007. Disponível em <http://zetcode.com/tutorials/winapi/>. Acesso em: 07/02/2012.

[3] CARISSIMI, A., DUPROS, F., MÉHAUT, J. and POLANCZYK, R. V. "Aspectos de Programação Paralela em Arquiteturas NUMA". Minicursos do VIII Workshop em Sistemas Computacionais de Alto Desempenho (WSCAD). 2007.

[4] CORSINI, J. F., FREITAS, L. G. and ROSSI, F. D. "Comparando Processos entre Unix e Windows". Revista INFOCAMP. Março de 2006.

[5] DREPPER, U. and MOLNAR, I. "The Native POSIX Thread Library for Linux". Technical Report. Red Hat, Inc. 2005.

[6] DUNKELS, A. "Protothreads - Lightweight, Stackless Threads in C". 2005. Disponível em:<http://www.sics.se/adam/pt/>. Acesso em janeiro/2012.

[7] ENGELSCHALL, R. S. "The GNU Portable Threads". 2006. Disponível em: <http://www.gnu.org/software/pth/>. Acesso em janeiro/2012.

[8] JOHNSON, P. "Pthread Performance in an MPI Model for Prime Number Generation". Technical report, University of Colorado. 2006.

[9] MOORE, G. E. "Cramming more Components onto Integrated Circuits". Readings in Computing Architecture. Editors: HILL, M. D., NORMAN, P. J. and GURINDAR S. S. . Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. Pages: 56-59. 2000.

[10] MUELLER, F. "A Library Implementation of POSIX Threads under UNIX". Proceedings of the USENIX Conference. Pages: 29-41. 1993.

[11] PENHA, D. O., CORRÊA, J. B. T., POUSA, C. R., RAMOS, L. E. S. and MARTINS, C. A. P. S. "Performance Evaluation of Programming Paradigms and Languages Using Multithreading on Digital Image Processing". Proceedings of 4th WSEAS International Conference on Applied Mathematics and Computer Science. 2005.
[12] SCHEFFER, R. "Uma visão Geral sobre Threads". Revista Campo Digit@l. Volume 2. Número 1. Páginas: 7-12. 2007.
[13] SILVA, R. R. and YOKOYAMA, R. S. "Avaliação do Desempenho da Utilização de Threads em User Level em Linux". Revista de Informática Teórica e Aplicada - RITA. Volume 18. Número 1. 2011.
[14] SPINELLIS, D. "A Critique of the Windows Application Programming Interface". Computer Standards and Interfaces. Volume 20, Issue 1, Pages: 1-8. 1998.
[15] TANENBAUM, A. S. "Modern Operating Systems (3rd edition)". Prentice Hall Press, Upper Saddle River, NJ, USA. 2008.
[16] THIBAULT, S. "PM2. Marcel: A Posix-Compliant Thread Library for Hierarchical Multiprocessor Machines". 2011. Disponível em: <http://runtime.bordeaux.inria.fr/marcel/>. Acesso em janeiro/2012.
[17] TORELLI, J. C. and BRUNO, O. M. "Programação Paralela em SMPS com OPENMP E POSIX Threads: Um estudo comparativo". Anais do IV Congresso Brasileiro de Computação (CBComp). Volume 1. Páginas: 486-491. 2004.

Alex G. C. de Sá graduou-se no final de 2011 em Ciência da Computação pela Universidade Federal de Lavras (UFLA). É mestrando em Ciência da Computação pela Universidade Federal de Minas Gerais (UFMG). Já trabalhou com Redes Neurais Artificiais aplicada a biologia (Bioinformática) e atualmente tem atuado em diversas linhas de pesquisa, como por exemplo: Modelagem e Simulação de Sistemas Heterogêneos (ênfase na parte de comunicação), Inteligência Computacional aplicada às Redes de Sensores Sem Fios e Programação Paralela e Distribuída.

Marluce R. Pereira é Professora Adjunta III do Departamento de Ciência da Computação da Universidade Federal de Lavras. Possui graduação em Informática pela Universidade Federal de Juiz de Fora (UFJF - 1999), mestrado em Engenharia de Sistemas e Computação pela Universidade Federal do Rio de Janeiro (UFRJ - 2001) e doutorado em Engenharia de Sistemas e Computação pela Universidade Federal do Rio de Janeiro (UFRJ - 2006). Tem experiência na área de Ciência da Computação, atuando principalmente nos seguintes temas: Programação Paralela e Distribuída, Processamento Paralelo, Sistemas Inteligentes, Programação Lógica com Restrições, Processo de Desenvolvimento de Software.

Pedro M. Moura é bacharel em Ciência da Computação pela Universidade Federal de Lavras (UFLA) no segundo semestre de 2011. Atua principalmente nas áreas de Redes Sem Fio, Redes de Nova Geração e Programação Paralela e Distribuída. É aluno de mestrado em Ciência da Computação pela Universidade Estadual de Campinas (UNICAMP) .

Luiz Henrique R. Peixoto possui curso técnico/profissionalizante em Eletroeletrônica pelo Serviço Nacional da Indústria de Itajubá (2005). É graduado em Ciência da Computação pela Universidade Federal de Lavras (2011). Na UFLA desenvolveu pesquisa nas áreas de Programação Paralela e Distribuída, Computação Evolutiva, Eletrônica e Programação para GPU com CUDA. É, atualmente, mestrando em Ciência e Tecnologia da Computação pela Universidade Federal de Itajubá (UNIFEI) também a partir de 2011.