Estudo e Análise de Técnica de Computação Paralela em Ambiente de Alto Desempenho

Thiago Piccoli, Paulo Salum

Centro Universitário Anhanguera de Niterói – UNIAN
Niterói – RJ
2015

Aprovado por:

André Luís S. Farias, M. Sc.,

Resumo. *Este artigo descreve conceitos de computação paralela, citando e fazendo uso ferramentas comumente utilizadas para programação paralela e também exibe os resultados obtidos a partir de testes utilizando a tecnologia threads, implementado na linguagem de programação Java em um ambiente de alto desempenho.*

1. Introdução

No campo da ciência são feitos grandes cálculos, no qual o cérebro humano poderia levar dias, meses ou anos para chegar a algum resultado, por conta disso, o uso de computadores nessa área é algo essencial. Para que os computadores possam ser realmente úteis, são desenvolvidos algoritmos que, quando utilizados para problemas específicos, ajudam na solução de problemas que possuem um grande grau de complexidade. Atualmente, nos grandes centros de pesquisa ao redor do mundo, são utilizadas grandes máquinas, comumente em estruturas de cluster, que são capazes de disponibilizar diversos núcleos de processamento para cálculos científicos.

Com o avanço da tecnologia, novas técnicas de programação surgem constantemente, sendo assim possível criar e executar algoritmos cada vez mais eficazes, capazes de atingir um nível de velocidade de execução cada vez maior. Com a aplicação dessas técnicas somadas a algoritmos avançados e um ambiente computacional que permite a utilização dessas ferramentas a um alto nível, como clusters, obtemos resultados mais rápidos graças à performance que é conseguida através da soma desses fatores.

Entretanto, mesmo a computação paralela possui seus limites. À medida que o número de processadores aumenta, também aumenta a sobrecarga de gerenciar a distribuição de tarefas entre estes processadores. Alguns sistemas de processamento paralelo requerem processadores extras somente para gerenciar os demais processadores e os recursos a eles distribuídos. A fim de evitar gargalos, o melhor que podemos fazer para remediar isto é assegurar que as partes mais lentas do sistema sejam aquelas que são menos usadas. Esta é a ideia por trás da Lei de Amdahl. Esta lei afirma que o acréscimo de desempenho possível para uma dada melhoria é limitado pela quantidade do uso daquele recurso melhorado. A premissa subjacente é que cada algoritmo possui uma parte sequencial que, no final, limita o aumento de velocidade que pode ser obtido pela implementação de multiprocessadores. [Null & Lobur, 2009]

O objetivo por trás de todos esses componentes é a construção de ferramentas capazes de diminuir o tempo de resposta de tarefas exigidas, trazendo resultados satisfatórios. Neste artigo,

iremos descrever conceitos de computação paralela, apresentando resultados a partir da técnica *threads*, tendo como foco a linguagem de programação Java.

1.1. Objetivo

Este trabalho tem como objetivos descrever alguns dos conceitos de computação paralela e apresentar análises e resultados a partir de testes de desempenho em um ambiente de alta performance, utilizando uma das técnicas de processamento paralelo mais usadas: threads.

Serão apresentadas análises de desempenho e avaliação de comportamento da tecnologia threads em um cenário de computação de alta performance, utilizando a linguagem de programação Java. Este estudo não irá abordar de maneira prática outras técnicas de programação paralela.

2. Principais conceitos de computação paralela

O principal propósito de processamento paralelo é realizar computação mais rápida do que pode ser feita com um único processador, usando mais do que um processador concorrente. [JaJa,1992]. O processamento paralelo também pode ser definido como "uma forma eficiente de processamento da informação com ênfase na exploração de eventos concorrentes no processo computacional". [Hwang & Xu, 1998]

A maioria dos sistemas atuais é de processador único; ou seja, eles só possuem uma CPU principal. Entretanto, os sistemas multiprocessados (Multiprocessing systems), também conhecidos como sistemas paralelos ou sistemas fortemente acoplados (tightly coupled systems), têm importância cada vez maior. Esses sistemas possuem mais de um processador em perfeita comunicação, compartilhando o barramento do computador, o relógio, à vezes, a memória e os dispositivos periféricos. [Silberschatz *et al*, 2008]

Os sistemas multiprocessados possuem três vantagens principais.

1. **Maior vazão (throughput)**. Aumentando o número de processadores, esperamos realizar mais trabalho em menos tempo. A taxa de aumento de velocidade com N processadores, porém, não é N, e sim inferior a N. Quando vários processadores cooperam em uma tarefa, um certo custo adicional é contraído para fazer com que todas as partes funcionem corretamente. Esse custo adicional (ou overhead), mais a disputa pelos recursos compartilhados, reduz o ganho esperado dos demais processadores. [Silberschatz *et al*, 2008]
2. **Economia de escala**. Os sistemas multiprocessados podem custar menos do que múltiplos sistemas de processador único equivalente, pois compartilham periféricos, armazenamento em massa e fonte de alimentação. Se vários programas operam sobre o mesmo conjunto de dados, é mais barato armazenar esses dados em um disco e fazer todos os processadores compartilhá-los do que ter muitos computadores com discos locais e muitas cópias dos dados. [Silberschatz *et al*, 2008].
3. **Maior confiabilidade**. Se as funções podem ser distribuídas corretamente entre diversos processadores, então a falha de um processador não interromperá o sistema, só o atrasará. Se tivermos dez processadores e um falhar, então cada um dos nove processadores restantes terá de apanhar uma fatia do trabalho do processador que falhou. Assim, o sistema inteiro é executado apenas 10% mais lento, em vez de travar completamente. Essa capacidade de continuar oferecendo serviço proporcional

em um nível do hardware sobrevivente é chamada de degradação controlada. Os sistemas designados para essa degradação também são considerados tolerantes a falhas (fault tolerant).[Silberschatz *et al*, 2008]

Várias técnicas podem ser usadas para promover algum nível de paralelismo e melhoria de desempenho na resolução de problemas, uma delas é o *pipeline*. Possivelmente o *pipeline* é uma das técnicas mais antigas na busca pelo paralelismo em execução de instruções. Ainda na década de 60, o *pipeline* foi introduzido nas arquiteturas de processadores. Essa tecnologia consiste basicamente em dividir a execução de uma instrução em vários estágios e cada um deles processa uma etapa da instrução por ciclo de *clock*. Dessa forma várias instruções podem ser executadas simultaneamente, desde que cada instrução esteja em etapas diferentes. [SOUZA, M.G.M., 2013]

Assim como é utilizado o *pipeline* (em nível de hardware) para que mais de uma instrução possa ser trabalhada ao mesmo tempo pelo processador, existe também o escalonamento de processos (feita pelo sistema operacional). Esse recurso fornece uma rápida mudança de contexto que faz com que o usuário que está utilizando o sistema se sinta em um ambiente paralelizado, mesmo em máquinas com somente um processador, ao oferecer uma fatia do tempo para cada processo.

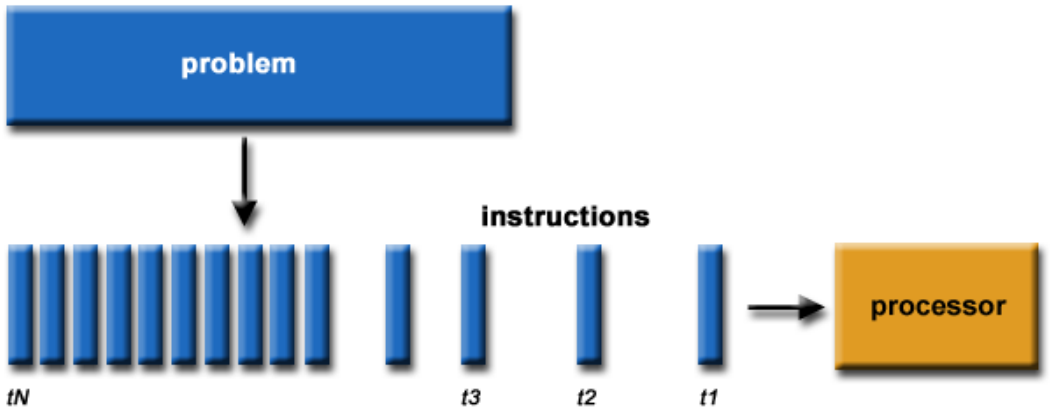

Figura 2.1: Imagem representando a divisão de um problema em pequenas instruções (Computação Serial).
Fonte: computing.llnl.gov

Tradicionalmente, o software que é escrito em serial funciona pegando um problema e dividindo em pequenas partes (instruções,) que são executadas sequencialmente, uma após a outra, conforme visto na figura 2.1. Somente uma instrução pode ser executada a qualquer momento e em um único processador. A divisão do problema em blocos independentes pode promover um paralelismo mais efetivo, como visto na figura 2.2.

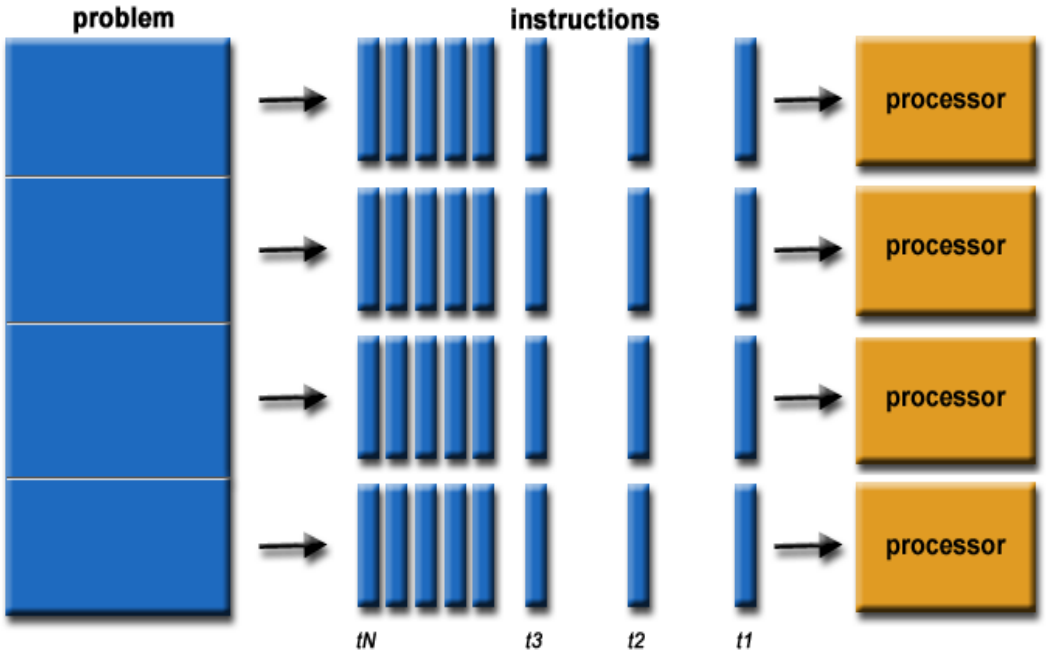

Figura 2.2: Imagem representando a divisão de problemas em partes independentes, e em seguida, instruções menores. (Computação Paralela). Fonte: computing.llnl.gov

Muitos problemas/tarefas não permitem sua divisão em partes independentes, um exemplo disso é o cálculo da série de Fibonacci em que o elemento seguinte depende do conhecimento dos dois elementos imediatamente anteriores a ele. O cenário encontrado mais comumente quando se trata de computação paralela são os problemas parcialmente paralelizáveis. Nesses casos, as partes independentes são entregues a unidades processadoras diferentes e há necessidades de sincronismo entre as etapas, forçando-as a aplicar o correto tratamento à correta fonte de dados. [SOUZA, M.G.M.,2013]

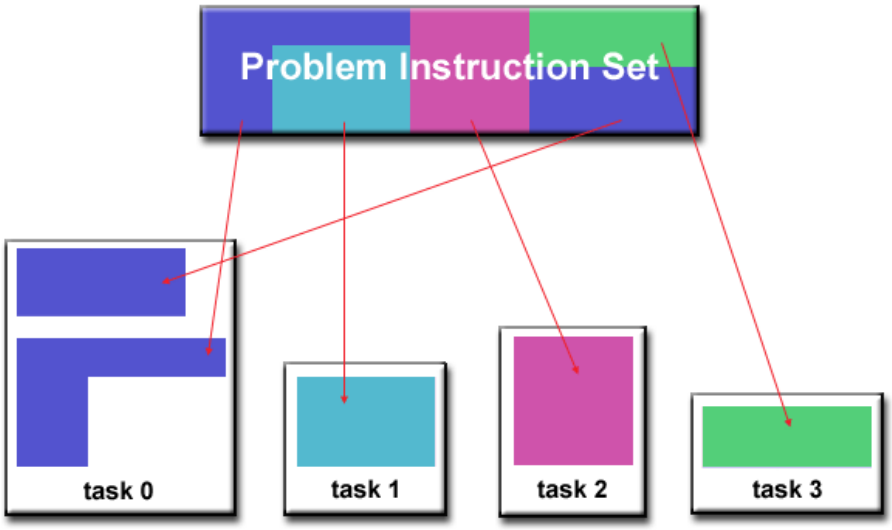

Figura 2.3: Exemplo de um problema parcialmente paralelizável. Divisão do problema em partes semi-independentes (permitindo o paralelismo), subdivididas para permitir o escalonamento pelo sistema operacional. Fonte: computing.llnl.gov

A Figura 2.3 ilustra um problema parcialmente paralelizável, onde algumas etapas necessitam que outras tenham cumprido certa parte da tarefa previamente.

2.1. Processos leves (*Threads*)

Threads, ou processos leves, é uma tecnologia que permite que tarefas possam ser executadas simultaneamente, comumente em arquiteturas que suportam multiprocessamento. *Threads* pode ser uma forma simplificada de se implementar paralelismo, dividindo um processo em fluxos de execuções distintos. Em um sistema operacional, ao se dividir um processo em fluxos distintos de execução, criando *threads* distintos, elas passam a usufruir do compartilhamento de memória de programa e dados do processo originário. Em decorrência disso cada *thread* pode acessar qualquer posição de memória dentro do espaço de endereçamento do processo. Essa característica os threads ler, escrever ou até apagar informações usadas por outros threads, exigindo um maior cuidado por parte do programador no manuseio e tratamento dos dados e tarefas do programa. [SOUZA, M.G.M.,2013]

O sistema operacional deve oferecer o suporte a *threads*, e as aplicações devem fazer o uso de alguma biblioteca para implementação desta tecnologia. No caso do Java, a disponibilidade de implementação da concorrência se dá por meio da linguagem e APIs, ao contrário das linguagens que não possuem essa capacidade integrada (como C e C++), o Java possui primitivos de *multithreading* (ou multiescalonamento) como parte da própria linguagem e de suas bibliotecas. Isso facilita a manipulação de threads de maneira portável entre plataformas. [Deitel 2010]

2.1.1 Benefícios

Os benefícios da programação multithread podem ser divididos em quatro categorias principais:

1. **Responsividade:** O uso de multithreads em uma aplicação interativa pode permitir que um programa continue funcionando mesmo que parte dele esteja bloqueada ou realizando uma operação longa, aumentando assim a responsividade ao usuário. Por exemplo, um navegador Web multithreads ainda poderia permitir a interação do usuário em uma thread enquanto uma imagem é carregada em outra thread. [Silberschatz *et al*, 2008]
2. **Compartilhamento de recursos:** Como padrão, as threads compartilham memória e os recursos do processo ao qual pertencem. O benefício do compartilhamento de código é que isso permite que uma aplicação tenha várias threads de atividades diferentes dentro do mesmo espaço de endereços. [Silberschatz *et al*, 2008]
3. **Economia:** A alocação de memória e recursos para criação de processos é dispendiosa. Como as threads compartilham recursos do processo ao qual pertencem, é mais econômico criar e trocar o contexto das threads. A avaliação empírica da diferença no custo adicional pode ser difícil, mas, em geral é muito mais demorado criar e gerenciar processos do que threads. No Solaris, por exemplo, a criação de um processo é cerca de trinta vezes mais lenta do que a criação de uma thread, e a troca de contexto é cerca de cinco vezes mais lenta. [Silberschatz *et al*, 2008]
4. **Utilização de arquiteturas multiprocessadas:** os benefícios do uso de multithreads podem ser muito maiores em uma arquitetura multiprocessada, na qual as threads podem ser executadas em paralelo nos diferentes processadores. Um processo dotado de única thread só pode ser executado em uma CPU, não importa quantas estejam à disposição. O uso de

múltiplas threads em uma máquina de múltiplas CPUs aumenta a concorrência. [Silberschatz *et al*, 2008]

2.2. Arquitetura NUMA

Máquinas com Acesso Não Uniforme à Memória (NUMA – Non uniform Memory Access) contornam problemas inerentes às arquiteturas UMA[1] fornecendo a cada processador sua própria porção de memória. O espaço de memória da máquina é distribuído para todos os processadores, mas os processadores veem esta memória como uma entidade de endereçamento contíguo. Embora memória NUMA constituam uma única entidade endereçável, sua natureza distribuída não é completamente transparente. Memórias próximas levam menos tempo para ser lidas do que a memória que está mais longe. Portanto, o tempo de acesso à memória é inconsistente ao longo do espaço de endereçamento da máquina. [Null & Lobur, 2009]

Máquinas NUMA são suscetíveis a problemas de coerência de cache. Para reduzir o tempo de acesso à memória, cada processador NUMA mantém uma cache privada. Entretanto, quando um processador modifica um elemento de dados que está em sua cache local, outras cópias do lado se tornam inconsistentes. Por exemplo, suponha que o Processador A e o Processador B tenham uma cópia do elemento de dado *x* em suas memórias cache. Digamos que *x* tem valor 10. Se o Processador A altera *x* para 20, a cache do processador B (que ainda contém o valor 10) tem um valor, ou estado, antigo de *x*. Inconsistências em dados como está não podem ser permitidas, de modo que devem ser providenciados mecanismos para assegurar a coerência de cache. Unidades de hardware especificamente projetadas, conhecidas como controladores de cache snoopy, monitoram todas as caches do sistema. Elas implementam o protocolo de consistência de cache do sistema. Máquinas NUMA que adotam caches snoopy e mantêm consistência de cache são denominadas arquiteturas NUMA com caches coerentes (CC-NUMA). [Null & Lobur, 2009]

A figura 2.4 exemplifica o funcionamento do compartilhamento de memória na arquitetura NUMA.

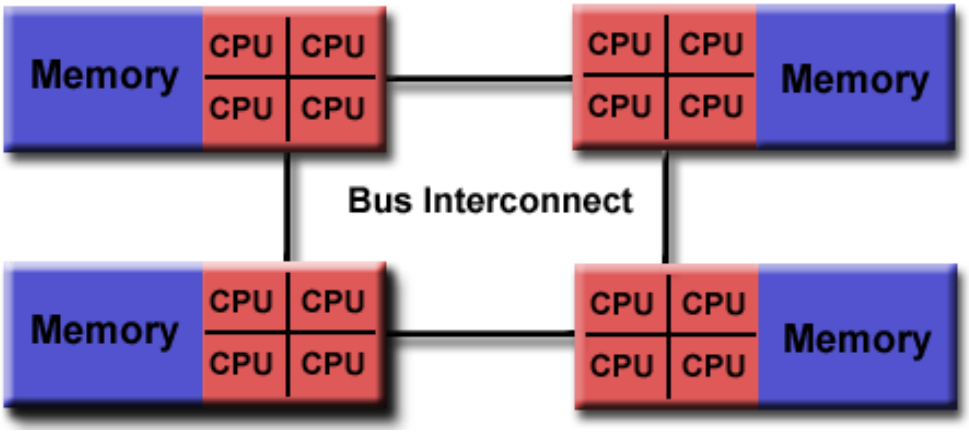

Figura 2.4: Imagem representando a arquitetura NUMA, onde possuem espaços de memória compartilhados entre os processadores. Fonte: computing.llnl.gov

3. Metodologia

Para se alcançar os objetivos propostos e observar o comportamento da programação em paralelo, utilizando uma das técnicas de paralelismo, decidimos por desenvolver um algoritmo que seja facilmente paralelizável e que ao mesmo tempo bastante custoso para o sistema processar. Os algoritmos que serão testados consistem em gerar duas matrizes quadradas, "A" e "B", 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. À medida que os programas forem executados, não mais de maneira sequencial, cada tarefa será dividida entre as CPUs. Isto acontece para que o tempo gasto, na geração das matrizes e no cálculo do produto seja dividido pelo número de núcleos, que é a principal vantagem da programação em paralelo.

3.1 Ambiente de teste

Para a realização dos testes e obtenção dos tempos de execução dos programas, foi utilizada a mesma plataforma computacional utilizada do CBPF (Centro Brasileiro de Pesquisas Físicas) e é gerenciado pela equipe da Coordenação de Atividades Técnicas.

As configurações da plataforma computacional utilizada nos testes possuem as seguintes características:

Placa mãe: Supermicro H8QG6 (4x slots, totalizando 64 núcleos)
Processadores: AMD Opteron 6376 (16 núcleos)
Memória RAM: 128GB Samsung DDR3-1600MHz ECC Registred (8x16GB)
Disco Rígido: Seagate ST1000NM0023
Sistema Operacional: CentOS 6.4 64b
Kernel: 2.6.32-358.e16.x86_64

3.2 Desenvolvimento do Algoritmo

Para se evitar o favorecimento da 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 javac, versão 1.8.0_66.

O código desenvolvido na linguagem de programa Java, quando executado, cria duas matrizes quadradas de tamanho 5120 x 5120, as povoam de forma paralela e em seguida faz o produto entre elas. Mais detalhadamente, as matrizes A e B são inicialmente povoadas com zero para garantir que não tenha conteúdo na memória, em seguida são preenchidas em um looping respeitando as seguintes formulas, para garantir a homogeneidade das matrizes em todos os testes com diferentes quantidades de threads.

$$Celula\ matriz\ A = 10 * (Pos\ linha + 1) + (Pos\ Coluna + 1)$$

$$Celula\ matriz\ B = 11 * (Pos\ linha + 1) + (Pos\ Coluna + 1)$$

Após isso, as *threads* de povoamento das matrizes serão destruídas e novas *threads* são criadas para fazer o produto matricial entre as duas matrizes.

```
for (int i = 0; i < NUM_THREADS; i++) {
    int posicao = i * numLinhas;
    Thread thread = new Thread(() -> {
        povoar(m1, posicao);
        povoar(m2, posicao);
        povoarComZero(m3, posicao);
    });

    thread.start();
    threads.add(thread);
}
```

Figura 3.1: Trecho de código que demonstra a criação de instancia das threads que povoam as matrizes.

3.3 Execução

Primeiro Executamos a versão do algoritmo em um único processador para fins de comparação. Em seguida executamos o programa para 16 quantidades de threads diferentes, sendo elas, 1, 2, 4, 5, 8, 10, 16, 20, 32, 40, 80, 128, 160, 256, 320, 512. Para se considerar uma medição válida, cada valor de thread foi executado cinco vezes gerando cinco tempos distintos de execução. O intervalo entre as execuções foi de 10 segundos por segurança, para evitar lixo em memória, atrapalhando o desempenho. Os tempos finais consistem de uma média de tempo das execuções.

Foi tomado o devido cuidado para que apenas a aplicação estivesse rodando no momento da execução, dessa forma evitando concorrência por CPU com outras aplicações.

3.4 Medição

Para realizar a medição do tempo de execução dos programas, será usada a função time presente nas distribuições GNU/LINUX. A resposta do comando tem como conteúdo o tempo real, com a resolução na escala de nano segundos. O intervalo de tempo será medido com a colocação deste comando junto ao comando na chamada de execução do programa.

Uma análise adicional também será feita com a ferramenta Java VisualVM para avaliar o comportamento das *threads* durante a execução.Com ela conseguimos visualizar de maneira gráfica o processo de criação e destruição de threads durante a execução, assim conseguindo garantir de forma mais precisa que o programa executa da forma que se foi pensada. O Java VisualVM é uma ferramenta que fornece uma interface gráfica para visualização de informações detalhadas sobre as aplicações Java enquanto estão em execução em uma JVM (Java Virtual Machine). Foi escolhida essa ferramenta por se tratar da ferramenta oficial da linguagem de programação Java e por trazer quase nenhum impacto durante a execução dos testes.

3.5. Speedup

Uma das formas mais utilizadas para medir desempenho de um algoritmo paralelo é comparando-o com o da sua versão serial, "O Speedup quantifica o fator de redução do tempo de execução de um programa paralelizado em um número P de processadores comparado à sua forma não paralelizada. É obtido pelo cálculo 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 utilizando P processadores em paralelo, denominado Tp, esta relação é expressa na equação 3.1"[SOUZA, M.G.M.,2013].

$$Speedup = \frac{T_S}{T_P} \tag{3.1}$$

A lei de Amdhal [2], vista na equação 3.2 na forma parametrizada, detalha o speedup para programas totalmente ou parcialmente paralelizáveis. Nesta forma a lei pode ser obtida pela relação entre o tempo total de execução de suas partes (Ts + Tp) em uma única unidade de processamento sob a soma do tempo de sua parte serial (Ts) com o tempo total das partes paralelas (Tp) dividido pelo número de CPUs usadas nelas (n), onde 1 é a forma parametrizada da soma total de tempo de execução de forma serial (Ts +Tp). [SOUZA, M.G.M.,2013]

$$Speedup = \frac{1}{T_S + \frac{T_P}{n}} \tag{3.2}$$

A tarefa proposta para a avaliação da tecnologia *thread*, 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. Neste caso específico a lei de Amdhal ficaria:

$$SpeedupT_{S\to 0} = \frac{Ts + Tp}{Ts + \frac{Tp}{n}} = \frac{Tp}{\frac{Tp}{n}} = \frac{n\,Tp}{Tp} \to Speedup = n \tag{3.3}$$

A equação 3.3 descreve em teoria o comportamento esperado nos gráfico 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.

4. Resultados Obtidos

A seguir, serão apresentados os resultados das simulações realizadas com o programa desenvolvido a partir de uma das técnicas de computação paralela.

Inicialmente foi observado e obtido o comportamento do tempo de execução em função do tamanho da matriz, conforme apresentado na Figura 4.1. Esta Figura apresenta as médias de tempo obtidas para o cálculo do produto matricial de matrizes quadradas com 5120 linhas, fazendo o uso de 1, 2, 4, 5, 8, 10, 16, 20, 32, 40 e 64 CPUs (representado por núcleos diferentes) e utilizando a tecnologia *thread*.

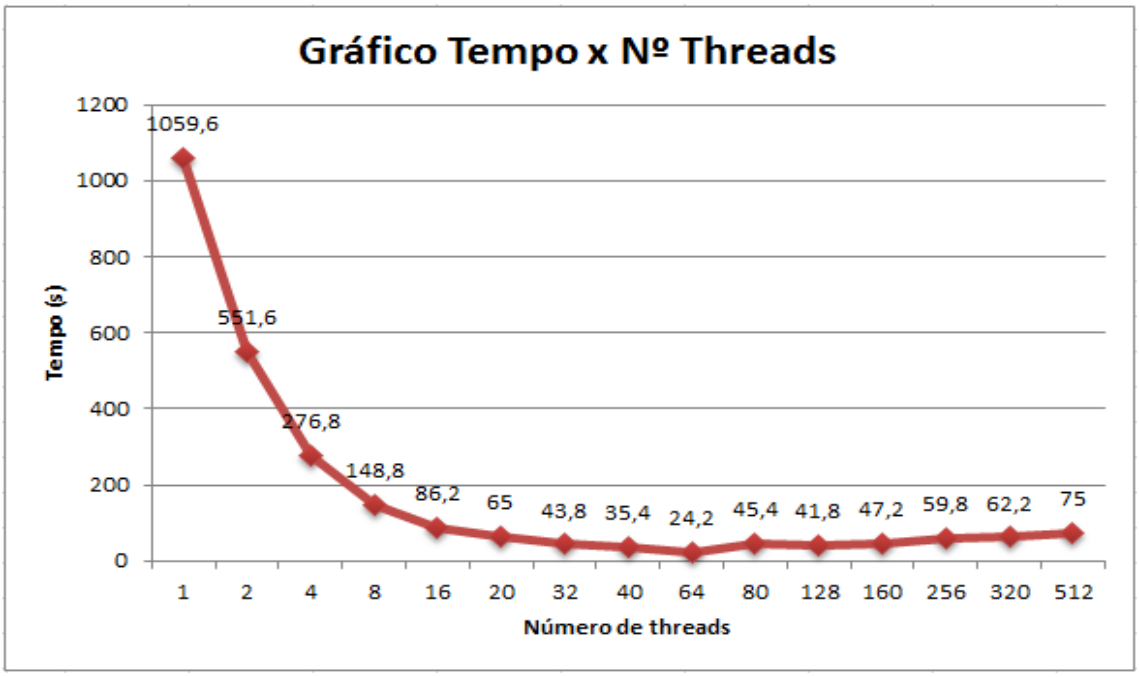

Figura 4.1: Comportamento do desempenho da tecnologia *thread* no cálculo de produto matricial para matrizes de 5120 linhas em 1,2,4,5,8,10,16,20,32,40 e 64 núcleos.

O resultado com todos os tempos obtidos e quantidade de *threads* utilizadas encontra-se na figura 4.2.

Tabela de Tempo de execução x Quantidade de Threads										
Nº Threads	Tempo 1	Tempo 2	Tempo 3	Tempo 4	Tempo 5	Média	Maximo	Minimo	ErroMax	ErroMin
1	1092	1101	933	1083	1089	1059,6	1101	933	41,4	126,6
2	549	548	559	550	552	551,6	559	548	7,4	3,6
4	282	282	283	280	257	276,8	283	257	6,2	19,8
8	149	149	151	148	147	148,8	151	147	2,2	1,8
16	94	85	95	67	90	86,2	95	67	8,8	19,2
20	56	56	65	73	75	65	75	56	10	9
32	42	34	50	56	37	43,8	56	34	12,2	9,8
40	44	32	33	38	30	35,4	44	30	8,6	5,4
64	26	22	24	24	25	24,2	26	22	1,8	2,2
80	33	39	60	51	44	45,4	60	33	14,6	12,4
128	47	40	61	27	34	41,8	61	27	19,2	14,8
160	47	50	35	45	59	47,2	59	35	11,8	12,2
256	47	85	59	42	66	59,8	85	42	25,2	17,8
320	82	53	47	33	96	62,2	96	33	33,8	29,2
512	114	60	100	51	50	75	114	50	39	25

Figura 4.2: Tabela com os tempos de execução dos cinco testes consecutivos.

A partir dos gráficos e resultados obtidos é possível perceber que a proporcionalidade entre o tempo total de execução e o número de CPUs usadas não é linear. Este comportamento é observado mais claramente nas Figuras 4.3 e 4.4.

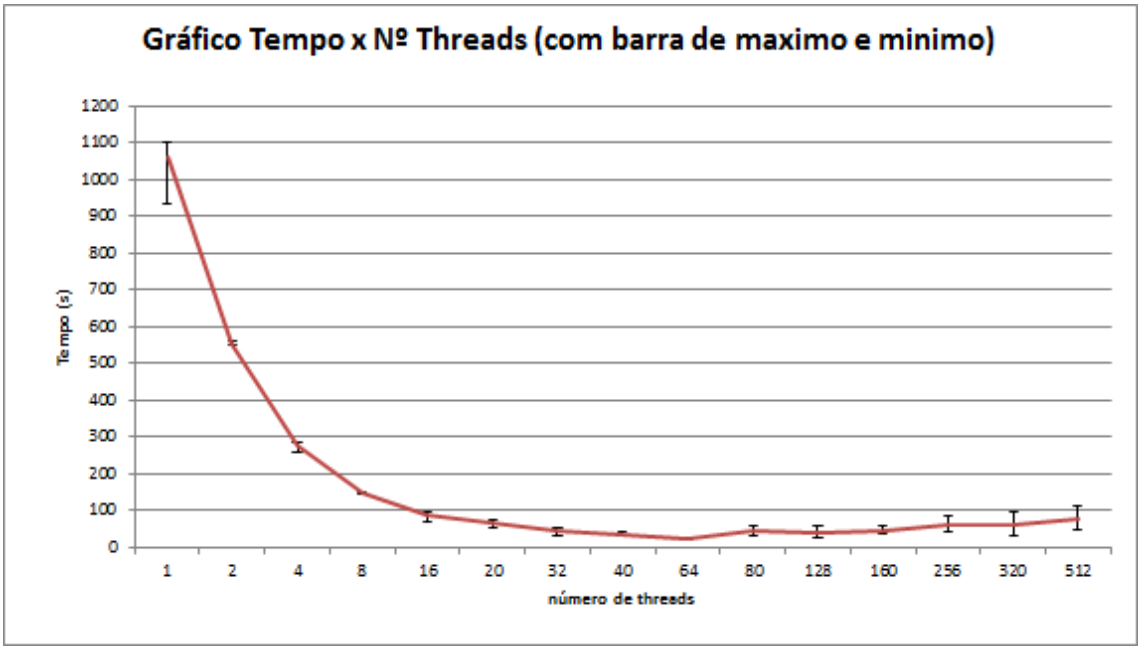

Figura 4.3: Comportamento do desempenho da tecnologia *thread* no cálculo de produto matricial utilizando 1, 2, 4, 5, 8, 10, 16, 20, 32, 40 e 64 núcleos, com barra de erros.

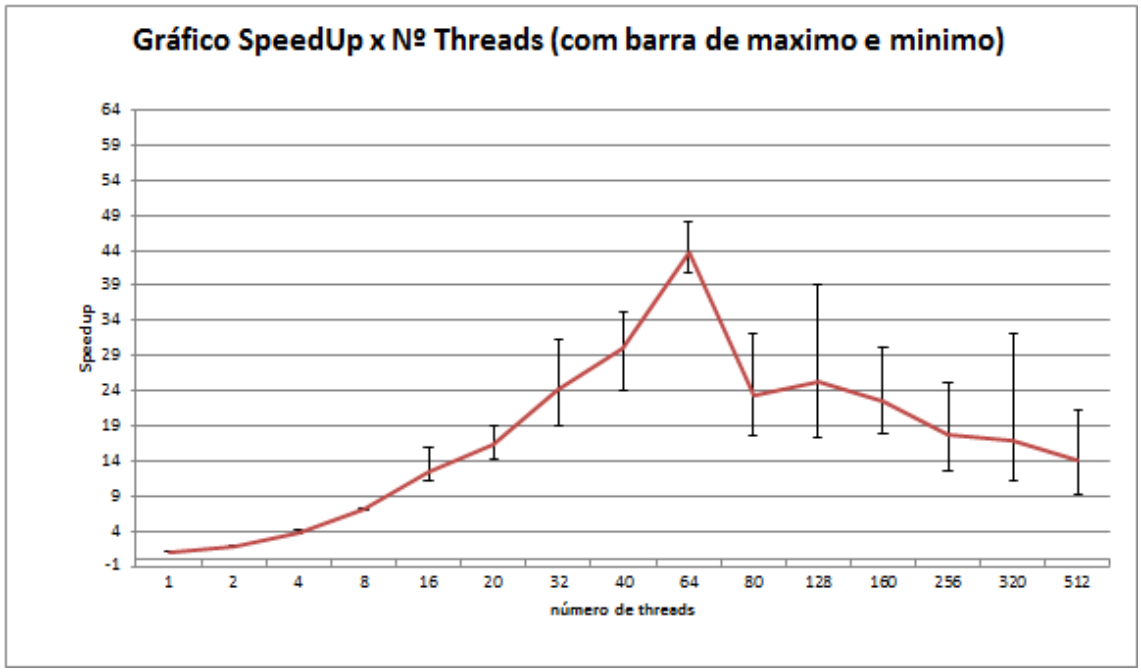

Figura 4.4: Gráfico dos speedups para a tecnologia thread no cálculo de produto matricial com 5120 linhas, fazendo uso de 1, 2, 4, 5, 8, 10, 16, 20, 32, 40 e 64 CPUs respectivamente.

O resultado com todos os speedups obtidos e quantidade de *threads* utilizadas encontra-se na figura 4.5. A tabela exibe os maiores e menores speedups atingidos, assim como os seus valores médios.

Tabela de Valor do SpeedUp X Quantidade de Threads					
Threads	speedup	Speedup MIN	Speedup MAX	ErroMAX	ErroMIN
1	1,00	0,96	1,14	0,14	0,04
2	1,92	1,90	1,93	0,01	0,03
4	3,83	3,74	4,12	0,29	0,08
8	7,12	7,02	7,21	0,09	0,10
16	12,29	11,15	15,81	3,52	1,14
20	16,30	14,13	18,92	2,62	2,17
32	24,19	18,92	31,16	6,97	5,27
40	29,93	24,08	35,32	5,39	5,85
64	43,79	40,75	48,16	4,38	3,03
80	23,34	17,66	32,11	8,77	5,68
128	25,35	17,37	39,24	13,90	7,98
160	22,45	17,96	30,27	7,83	4,49
256	17,72	12,47	25,23	7,51	5,25
320	17,04	11,04	32,11	15,07	6,00
512	14,13	9,29	21,19	7,06	4,83

Figura 4.5: Tabela com os tempos dos speedups obtidos durante os testes.

Percebemos algumas variações no tempo de execução em alguns testes, para identificar o que estava acontecendo realizamos algumas medições utilizando o Java visualVm, nela vimos que o comportamento de cada thread era diferente do esperado. Espera-se que para um programa onde todos os threads possuem o mesmo conteúdo e a mesma quantidade de valor a serem calculados, os tempos de execução seja o mesmo para todas.

Nas Figuras 4.6 e 4.7 é possível observar o comportamento e os estados dos threads no momento em que estão sendo executadas, através da ferramenta Java VisualVM.

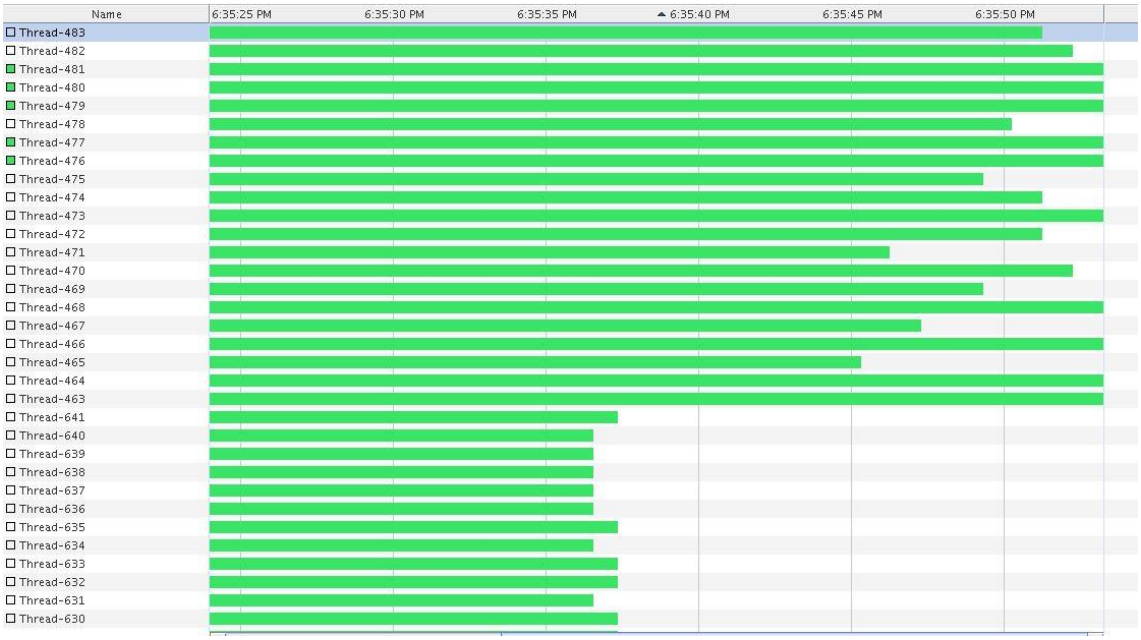

Figura 4.6: Pequeno trecho que mostra os estados dos threads ao longo da execução, utilizando a ferramenta Java VisualVM. Na imagem, execução de produto matricial de 5120 linhas utilizando 320 threads.

Figura 4.7: Pequeno trecho que mostra os estados dos threads ao longo da execução, utilizando a ferramenta Java VisualVM. Na imagem, execução de produto matricial de 5120 linhas utilizando 5120 threads.

5. Conclusão

O presente trabalho teve como objetivo principal fazer estudo e análise de técnica de computação paralela em ambientes de alto desempenho, utilizando a tecnologia *threads*. O estudo partiu de um resumo dos conceitos de computação paralela, seguido de testes realizados a partir de um algoritmo de multiplicação de matrizes, desenvolvido de forma paralelizada. Em seguida foram obtidas médias dos tempos de execução dos programas sob diferentes números de *threads*, cálculo de speedup e então feitas as avaliações de desempenho a partir desses dados.

O trabalho não teve como objetivo destacar a melhor técnica, e sim apresentar uma das mais utilizadas e o que ela pode oferecer no desempenho final, respeitando as suas próprias restrições na comunicação de informação das informações entre os processos e no formato de programação. A partir disto, foi possível estabelecer o funcionamento e desempenho da técnica em um ambiente de alto desempenho e com um problema altamente paralelizável, o produto matricial.

Era esperado um aumento desempenho conforme o número de núcleos fosse aumentando, até chegar a um ponto onde não haveria mais melhora e em seguida começaria levemente a cair e em seguida estabilizar o desempenho. A evolução no desempenho com o aumento do número de CPUs se manteve conforme o esperado até chegar ao valor de 64 núcleos (número total de CPUs da máquina utilizada no teste), onde teve seu melhor resultado atingindo o menor tempo (Figura 4.1 e 4.3) e speedup (Figura 4.4). As tabelas com todos os tempos dos testes são exibidas nas Figuras 4.2 e 4.5. Nas Figuras 4.6 e 4.7, alguns threads terminam antes de outras. Essa flutuação de tempo ocorre por diversos motivos, um deles é que uma vez que foi usada a arquitetura NUMA no ambiente de teste, o posicionamento em memória das matrizes A e B podem possivelmente ter influenciado nos tempos de execução.

Com relação à técnica de paralelização, pode-se concluir que o custo computacional para promover o paralelismo com *threads* é viável, uma vez que promove um melhor desempenho comparado à escrita serial, comunicação rápida através área de memória compartilhada, menor custo computacional por ser um processo leve e que faz rápidas trocas de contexto, fazendo com que o tempo de criação do processo e tempo de resposta do sistema seja mais rápido e uso eficiente de arquiteturas de multiprocessadores.

Apesar disso, a técnica não se limita a quantidade de processadores presentes em um computador, uma vez que quando o número de *threads* passa o de processadores, o desempenho tende a cair, mas em seguida é estabilizado. É comum hoje em dia termos acesso a clusters computacionais com diversos processadores distribuídos por vários computadores. O próximo passo para dar continuidade ao presente trabalho poderá focar na implementação do processamento paralelo em ambientes de cluster de computadores, partindo da atual simulação em um computador com processadores multicore para um conjunto de computadores multicores conectados em rede. Este processamento distribuído poderia ser feito utilizando a própria linguagem de programação Java, que foi usada nesse trabalho, com a inclusão de bibliotecas (como MPI, por exemplo).

O código desenvolvido para este trabalho encontra-se hospedado no repositório de códigos GitHub: http://www.github.com/

Considerações finais

Durante o desenvolvimento do trabalho algumas ideias de trabalhos futuros surgiram podendo assim acrescentar ao trabalho já desenvolvido neste artigo. São eles:

Granularidade: Estudo entre a relação entre carga de trabalho (frações) das frações paralelas de tarefas em relação à carga de trabalho para realização total das tarefas e que também tem ligação com *parallel overhead [3]*.

Utilização de computação distribuída: Realizar os testes desse artigo implementando MPI [4], com PVM [5] afim de realizar testes com mais de um nó de cluster.

Um comparativo entre thread e outra técnica de paralelização: Realizar um comparativo de desempenho entre a tecnologia Threads do Java e outra técnica de paralelismo da mesma linguagem, e realizar a análise dos resultados, verificando qual é a mais eficiente.

Comparativo entre threads em Java e threads em outra linguagem: Realizar analise de performance de threads em Java, C e Python e verificar a diferença entre os tempos de execução de cada uma em um ambiente igual.

Estudo da Técnica Forks em Java: Aprofundar-se no funcionamento da tecnologia Forks em Java, que utiliza conceitos como *work-stealing [6*] e verificar a diferença entre a técnica em Java e outras linguagens.

Referências

[1] Linda Null, Julia Lobur (2009), Princípios Básicos de Arquitetura e Organização de Computadores, Bookman Editora.

[2] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. Proceeding AFIPS ’67 (Spring) Proceedings of the April 18-20, 1967, spring joint computer conference, pages 483–485, August 1967.

[3] Barney, B (Lawrence Livermore National Laboratory), disponível em: https://computing.llnl.gov/tutorials/parallel_comp/#Concepts. Acesso em 01/11/2015 .

[4] Barney, B (Lawrence Livermore National Laboratory), disponível em: https://computing.llnl.gov/tutorials/parallel_comp/#Models. Acesso em 01/11/2015

[5] PVM research group, disponível em: http://www.csm.ornl.gov/pvm/pvm_home.html Acesso em 20/11/2015

[6] Robert D. Blumofe, Charles E. Leiserson. (1999), Scheduling multithreaded computations by work stealing, Journal of the ACM (JACM), Volume 46 Issue 5, Sept. 1999.

Souza, M.G.M. (2013), Paralelismo computacional de processamento digital de imagens aplicado à detecção de MARFEs no JET Alinhamento de Sequências Biológicas, Dissertação de Mestrado, Centro Brasileiro de Pesquisas Físicas, CBPF,Brasil.

Barney, B (Lawrence Livermore National Laboratory), disponível em: https://computing.llnl.gov/tutorials/parallel_comp/. Acesso em 01/11/2015.

K, Hwang , Z, Xu, Scalable Paralel Computing, McGraw-Hill, 1998

Paul J. Deitel, Harvey M. Deitel (2010), Java: como programar, Prentice Hall.

Silberschatz, A., Galvin, P. B. Gagne, G. (2008), Sistemas operacionais com Java, Elsevier.