UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

GUILHERME GRUNEWALD DE MAGALHÃES

Como Linguagens de Programação e Paradigmas Afetam Desempenho e Consumo Energético em Aplicações Paralelas

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Antonio Carlos Schneider Beck Filho
Co-orientador: Arthur Francisco Lorenzon

Porto Alegre
2016

AGRADECIMENTOS

Primeiramente gostaria de agradecer à minha família pelo apoio incondicional em todas as fases, seja nas melhores ou nas piores. Agradeço também aos amigos que fizeram parte da minha vida até aqui. São essas pessoas que fazem todas as dificuldades serem menores e as conquistas serem melhores.

Agradeço, também, ao meu orientador Caco, que mais do que professor, foi um grande mentor. Meu sincero obrigado pelas excelentes aulas, que não só me ensinaram muito, como também me inspiraram.

Agradeço também ao meu co-orientador Arthur, pelas horas discutindo e resolvendo *deadlocks* e *memory leaks*, além de sempre estar à disposição para ajudar com o que fosse.

Por fim, agradeço aos professores do Instituto de Informática da UFRGS por seu empenho em fazer o melhor trabalho possível para compartilhar seus conhecimentos e fazer do curso de Ciência da Computação, um curso tão reconhecido.

RESUMO

Este trabalho realiza um estudo de como o uso de diferentes paradigmas e linguagens de programação podem influenciar no consumo energético e desempenho de aplicações paralelas. Para isso, foram utilizados diferentes *kernels* do NAS Parallel Benchmark, implementados em linguagens de programação orientadas a objeto (C++ e Java) e procedural (C). Também foi avaliada a influência do compilador *Just-In-Time* (JIT) na execução de aplicações implementadas na linguagem Java.

A avaliação destas linguagens e paradigmas utilizou três métricas: desempenho, consumo energético e *energy-delay product* (EDP). Comparamos as implementações em C, C++, Java interpretada (sem o uso do compilador JIT) e Java compilada dinamicamente com o JIT. A partir da análise dos resultados obtidos da execução dos *benchmarks*, observou-se que a linguagem procedural C demonstrou melhor desempenho e menor consumo de energia que as linguagens orientadas a objeto Java e C++. Por exemplo, Java foi 50 vezes mais lenta que C, na execução da aplicação Lower-Upper (LU) sequencial, e C++ foi 7.98 vezes mais lento na execução da Integer Sort (IS) com 8 *threads*. Também observou-se que C++ foi superior a Java na maioria dos testes, por exemplo, Java foi 4.58 vezes mais lenta e gastou 4.16 vezes mais energia na execução da aplicação Scalar Pentadiagonal Solver (SP) com 8 *threads*. Observou-se também que Java tem uma dependência muito grande da eficiência do compilador JIT para obter um bom desempenho, por exemplo, na aplicação LU, onde o JIT não foi tão eficaz, Java foi 21.93 vezes mais lento que C++ na execução com 2 *threads*.

Palavras-chave: paradigmas de programação, avaliação de desempenho e energia, aplicações paralelas.

How Programming Languages and Paradigms Affect Performance and Energy in Multithreaded Applications

ABSTRACT

This work studies how different programming languages and paradigms influence performance and energy consumption in multithreaded applications. For that, we evaluate different kernels of NAS Parallel Benchmark, implemented in both procedural (C) and object-oriented programming languages (C++ and Java). Also, it was investigated the improvement that the Just-In-Time (JIT) compiler may provide.

The evaluation of this languages and paradigms has considered three different metrics: performance, energy consumption and energy-delay product (EDP). We compare C, C++, interpreted Java (without JIT) and dynamically compiled Java implementations. From the analysis of the obtained results, we show that C (a procedural language) has better performance and consumes less energy than C++ and Java (object-oriented languages). For instance, Java was 50 times slower than C, in Lower-Upper (LU) sequential execution, and C++ was 7.98 times slower in Integer Sort (IS) execution with 8 *threads*. We also show that in most cases C++ has better performance than Java, for instance, Java was 4.58 times slower and consumed 4.16 times more energy in Scalar Pentadiagonal Solver (SP) execution with 8 *threads*. Also, we show that Java depends heavily on the JIT compiler efficiency, for instance, in LU execution, when JIT was not very effective, Java was 21.93 times slower than C++ executing with 2 *threads*.

Keywords: programming paradigms, performance and energy evaluation, multithreaded applications.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLA

API	Application Programming Interface
CFD	Computational Fluid Dynamics
EDP	Energy-Delay Product
GCJ	GNU Compiler for Java
GPGPU	General Purpose GPU
GPU	Graphics Processing Unit
ILP	Instruction-Level Parallelism
IPC	Instruções Por Ciclo
IPP	Interface de Programação Paralela
JIT	Just-In-Time
JNI	Java Native Interface
JVM	Java Virtual Machine
NPB	NAS Parallel Benchmark
PI	Paradigma Imperativo
POO	Paradigma Orientado a Objetos
PP	Paradigma Procedural
RAPL	Running Average Power Limit
SHOC	Scalable Heterogeneous Computing
TLP	Thread-Level Parallelism

1 INTRODUÇÃO

Ao iniciar o desenvolvimento de uma aplicação, a escolha da linguagem de programação deve considerar aspectos importantes, tais como a facilidade de desenvolvimento, o paradigma de programação que a linguagem pertence e a plataforma na qual a aplicação será executada. A implementação não deve ser uma tarefa tortuosa, porém nesse quesito a escolha da linguagem não pode considerar apenas esse fator, uma vez que não se pode negligenciar desempenho ou consumo energética por comodidade. Por exemplo, no desenvolvimento de aplicações para servidores em geral, além do desempenho que é um fator relevante, o consumo de energia já se tornou uma questão fundamental (BARROSO e HÖLZLE, 2007).

As linguagens de programação modernas podem ser divididas em grupos pertencentes a paradigmas de programação, tais como Procedural e Orientado a Objetos, os paradigmas são como modelos de formas de programar. Por exemplo, o Paradigma Procedural (PP), ao qual a linguagem C faz parte, é um subconjunto do Paradigma Imperativo (PI) e baseia-se na chamada de procedimentos, que são construções que modularizam o código para fins de reuso (KAISLER, 2005). Por outro lado, o Paradigma Orientado a Objetos (POO) – ao qual as linguagens C++ e Java fazem parte – ajuda a aumentar a abstração aproximando a aplicação da realidade e tirando o foco de detalhes menos importantes, tais como gerenciamento de estruturas de dados e subtipos.

Considerando que as aplicações podem ser executadas em diversas plataformas substancialmente diferentes umas das outras, a escolha da linguagem também deve considerar as plataformas de destino. Por exemplo, no mercado de sistemas embarcados, aplicações para Android geralmente são baseadas em Java e são executadas em uma máquina virtual (GANDHEWAR e SHEIKH, 2010), o que confere portabilidade, ou seja, a possibilidade de executar o mesmo binário em diferentes plataformas de *hardwares*. Essa máquina virtual disponibiliza uma interface chamada *Java Native Interface* (JNI) que possibilita a execução de códigos C e C++ para tarefas de intenso processamento ao custo de portabilidade, uma vez que códigos C e C++ tendem a ser mais dependentes do *hardware* (GOOGLE, 2016).

Além da diferença na portabilidade, as três linguagens têm outras diferenças importantes. Enquanto Java é traduzida para *bytecode*, e posteriormente interpretada por uma máquina virtual, C e C++ são compiladas para código de máquina. Desta forma, as otimizações de C e C++ são feitas em tempo de compilação, o que permite inclusive

considerar características muito específicas do *hardware* para atingir níveis de otimização ainda maiores. Por outro lado, Java é otimizada em tempo de execução através do compilador *Just-in-time* (JIT) que é capaz de compilar trechos de código dinamicamente.

Com a popularização dos processadores *multicore*, é natural que as aplicações sejam implementadas para tirar melhor proveito desses múltiplos processadores. No entanto, para que isso ocorra, elas devem ser desenvolvidas visando o paralelismo, uma vez que um programa sequencial não usará todo o poder computacional disponível. Neste sentido, cada linguagem fornece ao programador diferentes maneiras de explorar o paralelismo. Por exemplo, em Java e C++ o paralelismo pode ser explorado com o uso de bibliotecas padrões de *Threads*, enquanto que na linguagem C, o programador deve fazer uso de Interfaces de Programação Paralela (IPPs), tais como OpenMP e POSIX threads.

Quando aplicações paralelas são executadas, a análise do consumo de energia torna-se importante uma vez que ele se comporta de forma diferente em cada uma das linguagens. Para melhor exemplificar esta situação, a Figura 1 apresenta uma análise do comportamento de desempenho e consumo de energia considerando a execução paralela de uma aplicação em um processador *multicore* com quatro núcleos (P_0, P_1, P_2 *e* P_3).

Na maioria das vezes, uma aplicação paralelizada obtém ganho de desempenho com relação à versão sequencial, pois o tempo total de computação é distribuído entre os processadores. Por exemplo, na Figura 1a, a aplicação sequencial levou 100 segundos para executar (*Ts*) enquanto que a versão paralela levou aproximadamente 30 segundos (*Tp*), representando ganho de aproximadamente 3.3 vezes no desempenho. No entanto, este mesmo comportamento não acontece no consumo de energia (Figura 1b), onde a energia total consumida pela execução paralela (*Ep*) corresponde à soma da energia consumida em cada um dos quatro processadores (P_0, P_1, P_2 e P_3), adicionada à energia

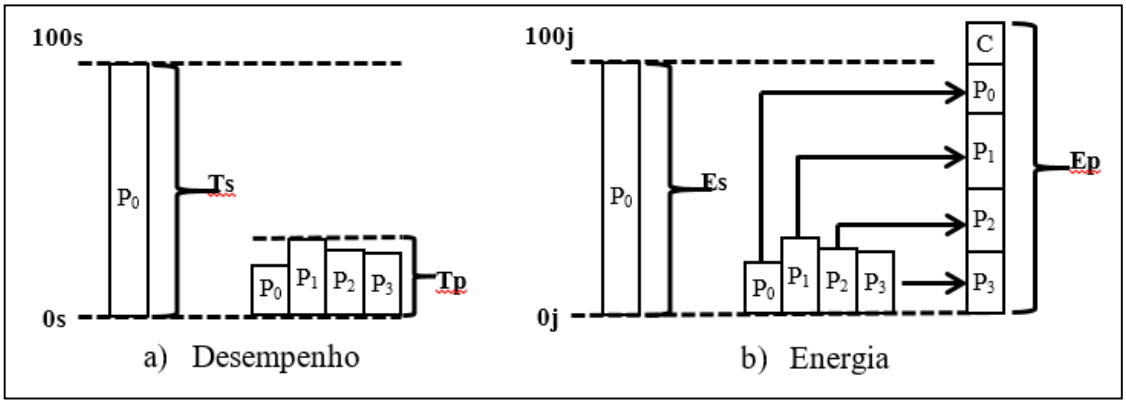


Figura 1 – Comportamento da Paralelização em termos de Desempenho e Energia

consumida para comunicação (*C*) entre os processadores – que varia de acordo com a aplicação. Por exemplo, aplicações com grande quantidade de comunicação terão, por consequência, consumo de energia maior que aplicações com pouca comunicação. Outro fator importante é o *overhead* que a execução em uma máquina virtual e com o compilador JIT podem ocasionar, uma vez que eles adicionam mais camadas de abstração e também consomem energia.

1.1 Objetivos

Conforme discutido anteriormente, além da plataforma alvo e do paradigma de programação, consumo energético e performance devem ser fatores determinantes na escolha de uma linguagem de programação. Com todas as diferenças mencionadas acima, é importante termos uma comparação de desempenho e consumo energético para avaliarmos as linguagens. Desse modo, esse trabalho avalia o impacto de uma determinada linguagem ou paradigma de programação no desempenho e consumo energético em aplicações paralelas. Para realizar essa análise:

- Serão comparadas diferentes implementações do NAS Parallel Benchmark (NPB), nas linguagens C, C++ e Java. Para tanto, implementou-se uma versão em C++ do conjunto de aplicações que originalmente foi desenvolvido em Java pela divisão de Supercomputação Avançada da NASA. A versão C, com OpenMP, foi desenvolvida e apresentada em KIM, SEO, *et al.* (2012).
- Também será avaliado o impacto do JIT e da máquina virtual *Java Virtual Machine* (JVM), verificando qual o impacto do compilador em tempo de execução no consumo energético e desempenho em aplicações Java assim como o impacto de executar em uma máquina virtual.
- Além disso, vamos comparar os paradigmas de programação Orientado a Objetos e Procedural, de modo que poderemos avaliar o impacto no desempenho e no consumo energético ocasionado pelas facilidades agregadas pelo aumento da abstração.

1.2 Estrutura do Texto

Na seção 2, serão apresentados os conceitos mínimos necessários para a compreensão do restante do trabalho. Abordaremos os paradigmas de programação

considerados nesse trabalho na seção 2.1 e posteriormente, na seção 2.2, explicaremos mais sobre computação paralela. Na seção 3, mostramos pesquisas desenvolvidas no campo da computação paralela comparando-as com as contribuições desse trabalho.

Na seção 4, explicamos a metodologia, os benchmarks utilizados (4.1), as métricas avaliadas (4.2) e o ambiente de execução (4.3). Na seção 5, mostraremos detalhes sobre a versão C++ do NAS Parallel Benchmark que implementamos. A seção de resultados (6) mostra uma análise dos dados obtidos considerando não só o desempenho e o consumo energético, mas também o *energy-delay product*, que é uma métrica que vem sendo amplamente utilizada para comparar ambientes diferentes como em BLEM, MENON e SANKARALINGAM (2013) e TIWARI, KEIPERT, *et al.* (2015), uma vez que ela nos possibilita em um único valor, analisar a relação entre energia e performance. Na seção de conclusão e trabalhos futuros, mostramos as conclusões a que chegamos e as perspectivas para trabalhos futuros. Por fim, mostramos as referências que auxiliaram nessa pesquisa.

2 CONCEITOS TEÓRICOS

Este capítulo apresenta os principais conceitos necessários para o entendimento do restante do trabalho. Na seção 2.1 serão explicados os Paradigmas de Programação utilizados e quais as suas diferenças. Inicialmente o PI será apresentado por se tratar do paradigma base tanto para o PP quanto para o POO. Posteriormente, os paradigmas foco dessa pesquisa serão expostos. Na seção 2.2, discorreremos sobre computação paralela, começando por processadores *multicore* e em seguida abordando a programação paralela.

2.1 Paradigmas de Programação

De acordo com o dicionário Michaelis de língua portuguesa, a palavra paradigma significa “Algo que serve de exemplo ou modelo; padrão” (EDITORA MELHORAMENTOS, 2015). Quando se trata de paradigmas de programação, eles podem ser entendidos como a forma de programar. Essa forma, vale ressaltar, não é relacionada às ferramentas ou individualidades das linguagens, mas à forma de resolver um problema. Por exemplo, o Paradigma Orientado a Objeto incentiva o desenvolvedor a modelar o problema de tal forma que o aproxime da realidade. Mas no que tange à forma como o programa será executado (se será compilado ou interpretado), é decisão de projeto das linguagens que nada têm a ver com os paradigmas aos quais elas pertencem.

Os conjuntos de linguagens pertencentes aos paradigmas não são necessariamente disjuntos, ou seja, as linguagens podem pertencer a mais de um paradigma. Um exemplo disso é a linguagem C que contém características do paradigma imperativo e também do procedural. Além disso, é importante notar que linguagens de programação encorajam o uso de um paradigma (FLOYD, 1979), não o tornam compulsório. Isso provém justamente do fato de os paradigmas estarem ligados à forma como o problema é entendido, e não às especificidades das linguagens.

Apesar de existirem muitos paradigmas de programação, os paradigmas foco deste trabalho serão o Paradigma Procedural e o Paradigma Orientado a Objetos, uma vez que as linguagens de programação mais populares hoje são pertencentes a esses paradigmas (TIOBE SOFTWARE BV, 2016). Nas subseções a seguir, esses paradigmas e o Paradigma Imperativo – por ser o paradigma base dos demais – serão explanados.

2.1.1 Paradigma Imperativo e Paradigma Procedural

O primeiro paradigma de programação (KAISLER, 2005) teve sua origem nos esforços iniciais pela automatização de cálculos. Em VON NEUMANN (1945) onde a Arquitetura de von Neumann foi introduzida, um sistema automático de cálculos (computador) foi descrito como sendo “um dispositivo capaz de executar instruções para realizar cálculos”. E sobre as instruções, é dito que elas “devem ser dadas ao dispositivo em detalhes absolutamente exaustivos”. Esse é exatamente o cerne do PI, o qual prega que o programa é um conjunto de comandos imperativos, como na linguagem natural, que indicam como realizar o cálculo esperado (VAN ROY e HARIDI, 2004) (GABBRIELLI e MARTINI, 2010). O PI é caracterizado pela existência de um estado e comandos que modificam esse estado (KAISLER, 2005).

Quando a programação imperativa é combinada com o uso de subprogramas, ela passa a ser chamada de programação procedural (KAISLER, 2005). Baseado na chamada de procedimentos, esse paradigma encoraja a modularização do código, na qual um problema maior é fracionado em procedimentos menores e reusáveis. Ao fazer a decomposição de um problema grande em partes menores, obtém-se problemas mais facilmente tratáveis os quais pode-se tratar um de cada vez considerando-se menos detalhes (NYGAARD e DAHL, 1981). Além disso, consegue-se limitar o escopo das variáveis, fazendo com que seu tempo de vida fique limitado ao menor possível.

Outra característica do PP, é aumentar o nível de abstração. A abstração permite a um analista pensar no que precisa ser feito, ao invés de como precisa ser feito (KAHATE, 2004). Esse aumento da abstração é obtido a partir do uso de procedimentos em determinadas parte do código nas quais sabemos o que precisa ser feito e deixamos que a forma como vai ser feito seja decidida em momentos posteriores do projeto (se já não tiver sido implementado). Por exemplo, no projeto de um sistema de controle de uma fornalha sabemos que quando a temperatura ficar abaixo de um valor *x*, é preciso aumentar sua temperatura. Nesse caso, em um momento precoce do projeto, o desenvolvedor definiria que essa ação será executada pelo procedimento *AumentarTermperatura()*. Porém, a implementação dessa função – definindo como o aumento de temperatura de fato será executado – somente precisa ser implementada posteriormente.

Além desse aumento do nível de abstração, criar procedimentos auxilia na reutilização de código, permitindo ao desenvolvedor não precisar implementar mais de

uma vez um mesmo trecho de código quando ele será usado em diversas partes do projeto. Por exemplo, no mesmo sistema da fornalha, ao abri-la para acrescentar mais quantidade do material sendo incinerado também é preciso aumentar sua temperatura. Essa verificação é feita em uma parte de código distinta, mas basta chamar novamente a função *AumentarTemperatura()* para que a mesma ação seja tomada sem que o código precise ser replicado. A reutilização de código nos leva a uma melhor manutenibilidade do código. Se a forma como o aumento de temperatura ocorre precisar ser modificado para que ele seja feito de forma mais gradual, por exemplo, basta corrigir a função *AumentarTemperatura().* Não sendo necessário modificar várias partes do projeto onde esse comportamento foi implementado.

Nesse trabalho, o PP será representado pela linguagem C. Ela é uma linguagem considerada próxima à máquina, de modo que a abstração introduzida é fundamentada em tipos de dados concretos. Por delegar a rotinas de bibliotecas as interações com o Sistema Operacional, ela ainda é suficientemente independente de hardware para que seja até certo ponto portável (RITCHIE, 1993).

2.1.2 Paradigma Orientado a Objetos

No POO, o desenvolvedor é encorajado a modelar a realidade de modo que ela seja dividida em classes. Essas classes são grupos de objetos com as mesmas características (atributos) e comportamentos (métodos). Essa modelagem implica em um nível maior de abstração, possibilitando que os problemas sejam melhor entendidos. Voltando ao exemplo da seção anterior e aplicando o conceito de classe e objetos, poderíamos entender fornalha como sendo uma classe, que temperatura seja um de seus atributos, *AumentarTemperatura()* um de seus métodos e a fornalha de uma determinada sala seja um objeto dessa classe.

No entanto, o POO vai além da simples divisão por classes: de acordo com VAN ROY e HARIDI (2004), programar com classes e objetos é chamado de Programação Baseada em Objetos e adicionando a isso a ideia de herança, temos Programação Orientada a Objetos. Os princípios básicos do POO são: encapsulamento, abstração, subtipos, herança e seleção dinâmica de métodos (GABBRIELLI e MARTINI, 2010). Vamos definir brevemente cada um desses princípios:

a) Encapsulamento – proteger dados sensíveis do objeto. Atributos privados só poderão ser modificados por métodos do objeto, ficando protegidos contra alterações externas. No exemplo da fornalha, o dado sensível é a temperatura e o método *AumentarTemperatura()* certifica-se de que a temperatura não será aumentada além do limite seguro.

b) Abstração – abstrair detalhes e considerar o macro. Semelhante à abstração obtida pelo uso de procedimentos no PP.

c) Subtipos – criar uma classe derivada de outra herdando atributos e métodos da classe superior. Esses métodos, porém, precisam ser implementados, uma vez que os comportamentos não serão necessariamente iguais. Dada uma classe S, a classe derivada D será seu subtipo se todos os métodos chamados em S puderem ser chamados em D, ainda que com resultados diferentes. O Subtipo é uma relação entre as interfaces de duas classes.

d) Herança – criar uma classe derivada de outra herdando atributos e métodos da classe superior. Diferente de subtipos, os métodos não precisam ser implementados e terão o mesmo comportamento da classe superior. Dada uma classe S, a classe derivada D será sua herdeira se todos os métodos chamados em S puderem ser chamados em D tendo resultados iguais. A herança é uma relação entre as implementações de duas classes.

e) Seleção dinâmica de métodos – também conhecido como despacho, é o coração do POO (GABBRIELLI e MARTINI, 2010). Como visto anteriormente, um método pertencente a um subtipo pode ter uma implementação diferente da classe superior. Devido à seleção dinâmica de métodos, quando um método M de um objeto é chamado, o método correto é selecionado dinamicamente dependendo do tipo do objeto. Por exemplo, vejamos o trecho de código (com sintaxe simplificada) na Figura 2:

```
class Super                    //super classe
class Sub extends Super        //subtipo
Sub objeto
Super ponteiro = objeto
objeto.M()                     //chamada 1
ponteiro.M()                   //chamada 2
```

Figura 2 – Trecho de código exemplificando seleção dinâmica de métodos

A classe Super é o tipo e a classe Sub é o subtipo que estende a classe Super. É criado um objeto do tipo Sub e um ponteiro do tipo Super que aponta para o objeto. Nas chamadas 1 e 2, o mesmo método M é chamado e, apesar de os ponteiros serem de tipos diferentes, ambas as chamadas invocarão o método M implementado na classe Sub pois o objeto chamado é do tipo Sub. Se a chamada fosse decidida estaticamente pelo compilador, ele poderia entender que o método M a ser chamado é o da classe Super, pois o ponteiro é dessa classe. Mas como a seleção é feita dinamicamente, o método correto é selecionado.

O exemplo acima é simples para fins didáticos e um compilador poderia ser capaz de identificar o método correto. Porém, ao considerar um programa muito maior onde a classe Super tenha vários subtipos – cada um com uma implementação diferente do método M – e o objeto seja criado em tempo de execução com o subtipo que depende da entrada do usuário, o compilador não seria capaz de saber o método correto a ser evocado. Num caso como esse podemos notar a grande importância da seleção dinâmica de métodos.

Explicados os 5 princípios básicos do POO, vale ressaltar que, como foi citado anteriormente, ainda que uma determinada linguagem seja dita Orientada a Objetos, ela não necessariamente precisa suportar todos esses princípios. Por exemplo, em C++ a seleção de métodos ocorrerá sempre de forma estática devido a decisões de projeto que priorizam eficiência e compatibilidade com C (GABBRIELLI e MARTINI, 2010). Adicionalmente, uma Linguagem Orientada a Objetos não precisa necessariamente ser imperativa, mas as mais bem sucedidas foram implementadas tendo como base ou sendo influenciadas por linguagens imperativas (KAISLER, 2005). Um bom exemplo disso é a

linguagem C++, uma das mais utilizadas Linguagens Orientadas a Objetos (TIOBE SOFTWARE BV, 2016) que foi criada a partir de C, que é uma linguagem imperativa.

Java e C++ são as mais comuns linguagens pertencentes ao POO (TIOBE SOFTWARE BV, 2016), sendo suas representantes nesse trabalho. Projetada com o objetivo de ser um C melhorado, C++ tinha como objetivo corrigir os problemas de C sem perder suas vantagens (STROUSTRUP, 1993). Um dos objetivos do projeto de Java era que ela parecesse e passasse uma sensação de C++, de modo que fosse familiar para os programadores C++ e de fácil migração. O projeto de Java também visava diminuir a complexidade desnecessária de C++ e tornar a sua portabilidade limitada em uma portabilidade verdadeira.

2.2 Computação Paralela

Durante muito tempo, a resposta para a necessidade de maior poder computacional foi o aumento da quantidade de transistores nos circuitos integrados e aumento da sua frequência de operação. A Lei de Moore (MOORE, 2006), uma das mais notórias leis da computação, passou a ser questionada recentemente devido a limitações físicas como miniaturização dos transistores, temperatura máxima suportada e consumo energético (FRANK, 2002). Para contornar essas limitações, passou-se a aumentar do número de unidades de processamento por chip.

Nesse cenário a computação paralela tem se popularizado. Ela se caracteriza pelo uso dessas múltiplas unidades de processamento por chip para executarem cálculos simultaneamente para completar uma determinada tarefa. A computação paralela visa principalmente: resolver cálculos cada vez mais complexos, reduzir o tempo de solução de cálculos, realizar cálculos mais precisos em um tempo razoável e usar de forma mais eficiente os recursos computacionais disponíveis (WILKINSON e ALLEN, 1999).

A computação paralela pode ser dividida em dois tipos: no nível de instruções (*Instruction-Level Parallelism* - ILP), e no nível de *thread*s (*Thread-Level Parallelism* - TLP). No ILP, arquiteturas superescalares executam simultaneamente instruções da aplicação em uma mesma unidade de processamento, enquanto que no TLP, conjuntos de instruções, ou tarefas, são paralelizadas em múltiplas unidades de processamento. Além disso, a computação paralela pode ocorrer em ambientes distribuídos ou em computadores com múltiplos núcleos de processamento. O foco desse trabalho é o

paralelismo em nível de *threads* em computadores *multicore*. Na seção 2.2.1 abordaremos os processadores *multicore*, na seção 2.2.2 explicaremos mais sobre programação paralela, modelos de comunicação e sincronização entre *threads* e na seção 2.2.3 descreveremos as IPPs utilizadas na implementação das versões do NPB consideradas nesse trabalho.

2.2.1 Processadores Multicores

Desempenho é uma combinação de frequência de operação e número de instruções por ciclo, ou seja, podemos obter uma melhor performance pelo aumento da frequência ou do número de instruções por ciclo (GEPNER e KOWALIK, 2006). O incremento da frequência, porém, implica em aumento de temperatura de operação e aumento de gasto energético, o que impõe limitações na escalabilidade (FRANK, 2002). Por outro lado, o aumento do número de instruções por ciclo (IPC) não tem essas limitações e pode ser alcançado pelo aumento do número de unidades de processamento, paralelizando a execução das tarefas. Apesar de computação paralela não ser um conceito novo, já que computadores interconectados trabalhando paralelamente existem há décadas, as arquiteturas *multicore*, com replicação de núcleos de processamento a nível de chip, começaram a popularizar-se a partir dos anos 2000 (GEPNER e KOWALIK, 2006).

Diferentemente dos *clusters*, onde as unidades de processamento encontram-se distribuídas em máquinas distintas, nos processadores *multicore* as unidades de processamento estão em um mesmo chip e compartilham memória principal e *cache* L2

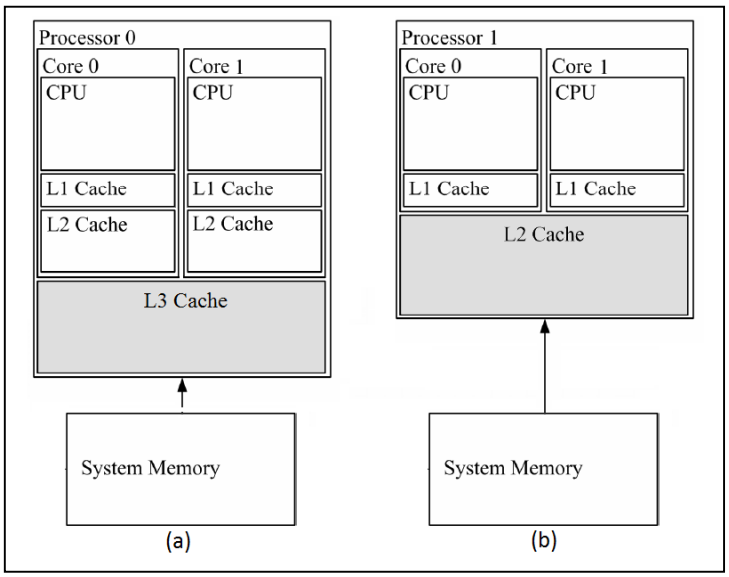


Figura 3 – Exemplos de arquiteturas de processadores *multicore*

ou L3 (HENNESSY e PATTERSON, 2007), ficando a critério do projeto da arquitetura quais níveis serão compartilhados. Por exemplo, como podemos ver na Figura 3, a arquitetura (a) apresenta dois níveis de *cache* exclusivos (L1 e L2) para cada núcleo além de um nível L3 compartilhado. Enquanto isso, a arquitetura (b) conta com apenas o *cache* nível L1 exclusivo e nível L2 já é compartilhado entre os núcleos.

A arquitetura da Figura 3a) é típica de um processador de propósito geral que precisa de uma hierarquia de memória mais robusta, enquanto a Figura 3b) assemelha-se mais a uma arquitetura de sistemas embarcados. Sistemas embarcados costumam ter hierarquias de memória de menor profundidade, uma vez que os acessos às regiões mais distantes levam mais tempo e são mais custosas energeticamente, além do fato que maiores quantidades de *cache* ocupariam uma área maior do chip e área é um recurso escasso em sistemas embarcados (GE, LIM e WONG, 2005)

2.2.2 Programação Paralela

A programação paralela caracteriza-se pelo uso de múltiplas unidades de processamento simultaneamente para realizar os cálculos necessários para uma aplicação. Como muitas vezes os valores de entrada para os cálculos de um processador serão a saída dos cálculos de outro, a corretude funcional dessa aplicação paralela requer duas interações entre os processadores: a sincronização de tarefas concorrentes e a comunicação de resultados parciais (GRAMA, GUPTA, *et al.*, 2003).

A comunicação entre os processadores pode se dar de duas formas: pela troca de mensagens ou pelo compartilhamento de memória (GRAMA, GUPTA, *et al.*, 2003). O foco desse trabalho são IPPs que fazem uso de memória compartilhada, mas a seguir vamos definir brevemente os dois métodos de comunicação e como é feita a sincronização das execuções em cada um deles.

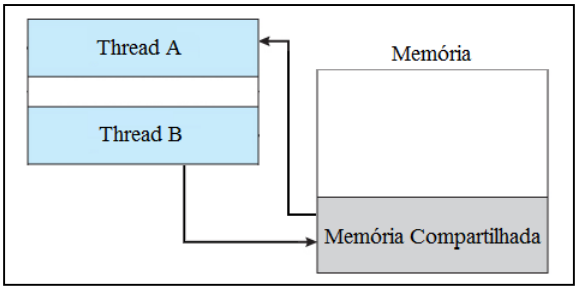


Figura 4 – *Thread* A recebendo resultado da *Thread* B com memória compartilhada

```
01. incrementar(){
02.   varCompart++;
03. }
```
(a)

```
01. decrementar(){
02.   varCompart--;
03. }
```
(b)

Figura 5 – Acesso concorrente à variável compartilhada

2.2.2.1 Memória Compartilhada

A comunicação por memória compartilhada baseia-se na existência de um espaço de endereçamento ao qual todas as *threads* tenham acesso. A comunicação por memória compartilhada ocorre de forma implícita, uma vez que parte (ou todo) o espaço de endereçamento é compartilhado entre as *threads* (GRAMA, GUPTA, *et al.*, 2003). Na Figura 4 é possível ver um exemplo no qual a computação da *thread* A recebe como entrada o resultado da *thread* B.

Sejam os exemplos de código da Figura 5, (a) sendo executado pela *thread* A e o código da (Figura 5b) sendo executado pela *thread* B, podemos constatar que problemas podem ocorrer nessa comunicação. Por exemplo, se ambas as *threads* chegarem à linha 2 simultaneamente, o resultado da variável compartilhada `varCompart` é indeterminado, pois tendo ela sido inicializada com o valor zero, dependendo da forma como ocorrer a execução das *threads*, o seu valor final pode ser:

a) `varCompart = 0` se a *thread* que começar a instrução não for interrompida.

b) `varCompart = 1` se a *thread* A começar a execução e antes de escrever o resultado da sua operação, for interrompida pela *thread* B.

c) `varCompart = -1` se a *thread* B começar a execução e antes de escrever o resultado da sua operação, for interrompida pela *thread* A.

Esse tipo de problema é conhecido como condição de corrida e pode ser resolvido pelo uso de sincronização. Nesse caso, a linha 2 de ambos os códigos são seções críticas,

```
01. incrementar(){
02.   exclusaoMutua{
03.     varCompart++;
04.   }
05. }
```
(a)

```
01. decrementar(){
02.   exclusaoMutua{
03.     varCompart--;
04.   }
05. }
```
(b)

Figura 6 – Uso de exclusão mútua para proteger variável compartilhada

ou seja, zonas onde uma variável compartilhada é modificada por uma *thread*. Acrescentando-se uma área de exclusão mútua, como pode ser visto na Figura 6, esse problema pode ser evitado, uma vez que a exclusão mútua impede que mais de uma *thread* acessem a seção crítica simultaneamente. Com o uso da exclusão mútua, a corretude funcional da aplicação será mantida.

A sincronização tem outras funcionalidades além da exclusão mútua, ou seja, impedir que uma zona crítica seja acessada por mais de uma *thread* concorrentemente. Consideremos o caso em que uma função cria *threads* de trabalho para que executem as tarefas e posteriormente retorna o valor calculado pelas *threads*. Consideremos os trechos de código na Figura 7, onde quatro *threads* são criadas. Elas realizarão os cálculos e ao final atualizam a variável compartilhada `resultado`. Ao terminar a criação e em seguida já retornar o valor de `resultado`, como na Figura 7a) pode ocasionar no retorno do valor errado, uma vez que as *threads* de trabalho podem não ter finalizado suas execuções. Por outro lado, ao acrescentar junções, como nas linha 5 até 7 da Figura 7b), a função principal esperará pela execução de cada uma das *threads*, somente retornando o valor em `resultado` ao final do cálculo.

```
01. calcular(){
02.   for(i=0; i<4; i++){
03.     criarThread();
04.   }
05.   return resultado;
06. }
```

(a)

```
01. calcular(){
02.   for(i=0; i<4; i++){
03.     criarThread(i);
04.   }
05.   for(i=0; i<4; i++){
06.     joinThread(i);
07.   }
08.   return resultado;
09. }
```

(b)

Figura 7 – Uso de junções

2.2.2.2 Troca de Mensagens

O paradigma de comunicação por troca de mensagens assume que o espaço de endereçamento de memória é particionado, ou seja, os processos não têm acesso à uma memória central compartilhada. Por esse motivo, a troca de mensagens é mais utilizada

Figura 8 – Troca de mensagens assíncronas

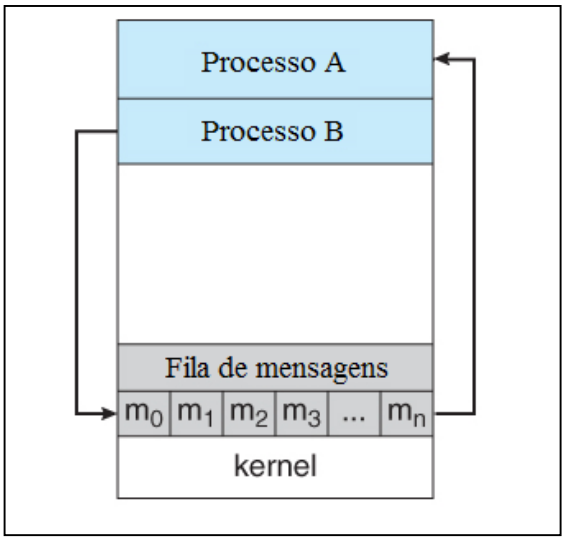
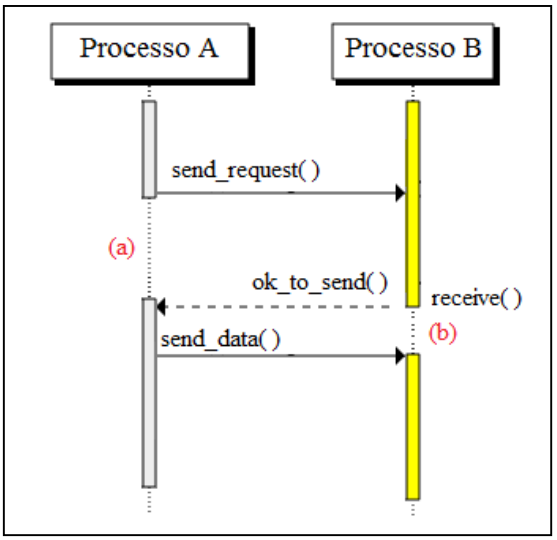


Figura 9 – Troca de mensagens síncrona



em paralelismo a nível de processos, pois eles de fato não têm acesso ao espaço de endereçamento de outros processos apesar de estarem em um mesma máquina (GRAMA, GUPTA, *et al.*, 2003), além de ambientes distribuídos, onde os nodos estão geograficamente afastados. A comunicação por troca de mensagens pode ocorrer de forma síncrona ou assíncrona. Na Figura 8 é possível observar a troca de mensagens síncrona, na qual o processo A fica bloqueado no momento (a), quando faz a solicitação de envio de mensagem e o processo B fica bloqueado em (b), quando começa a espera pra receber a mensagem. Na Figura 9 temos um exemplo de troca de mensagens assíncrona no qual o processo A busca mensagens de uma fila de mensagens e o Processo B insere suas mensagens na fila.

Diferentemente da comunicação por variáveis compartilhadas, a comunicação por troca de mensagens é explícita, ou seja, para que a interação entre dois processadores ocorra, ambos precisam chamar explicitamente as rotinas de *read/receive*. Essa necessidade agrega uma grande complexidade na programação das aplicações e também implica que o processo detentor dos dados participe da interação, mesmo que ele não tenha conexão lógica com os eventos ocorrendo no processo solicitante (GRAMA, GUPTA, *et al.*, 2003). A comunicação por troca de mensagens também pode ocorrer de forma coletiva, ou seja, um processo envia uma mensagem e dois ou mais processos recebem.

2.2.3 Interfaces de Programação Paralela

O desenvolvimento de aplicações (ou programas) capazes de explorar melhor o paralelismo potencial das arquiteturas multiprocessadas depende de uma série de aspectos específicos da organização destas arquiteturas, entre eles, o tamanho e a estrutura hierárquica da memória. Os Sistemas Operacionais fornecem certa transparência com relação à existência de múltiplos núcleos e responsabilizam-se pela distribuição das tarefas solicitadas pelos usuários entre os núcleos disponíveis. Entretanto, contar apenas com o Sistema Operacional, em muitos casos, pode levar a um desperdício do poder computacional disponível. Por exemplo, estar executando apenas um processo de uma aplicação que poderia ser dividida em vários processos em um sistema *multicore* de 8 núcleos.

Os processos de comunicação entre processos/*threads* apresentados na seção 2.2.2 e a extração do paralelismo não são transparentes ao programador. Isto é, é necessário que o mesmo indique, explicitamente, regiões do programa que podem ser executadas em paralelo, e como será feita a troca de dados: distribuição dos dados de entrada entre as partes paralelas; comunicações durante a computação paralela; e composição da solução final a partir dos resultados parciais computados nas partes paralelas. Para facilitar esta tarefa, com o intuito de diminuir o tempo de desenvolvimento e o deixar menos propenso a erros, IPPs têm sido utilizadas. Como já foi citado anteriormente, o foco desse trabalho são IPPs que fazem uso de comunicação por compartilhamento de memória, a seguir apresentamos as IPPs utilizadas no desenvolvimento das aplicações em cada uma das implementações do NPB.

2.2.3.1 OpenMP

Open-Multi Processing ou OpenMP, como é mais conhecida, é uma *application programming interface* (API) para processamento paralelo que pode ser usada com as linguagens de programação FORTRAN, C e C++. OpenMP fornece suporte a concorrência, sincronização, balanceamento de carga e gerenciamento de acesso a dados compartilhados a partir de diretivas ao compilador (GRAMA, GUPTA, *et al.*, 2003). Essa IPP objetiva ser amigável para o usuário, portável e eficiente; e com a inserção das diretivas OpenMP apropriadas, ela permite que a maioria das aplicações sequenciais

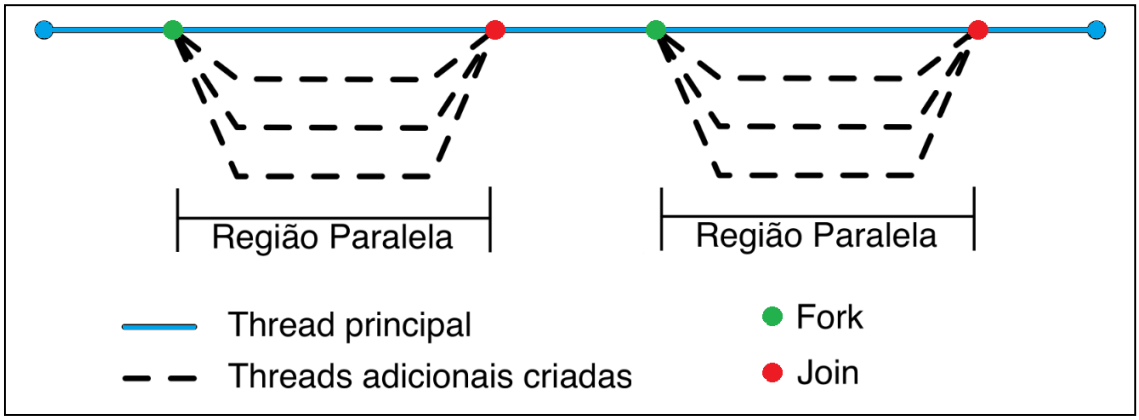


Figura 10 – Fluxos de execução em OpenMP

possam tirar proveito dos benefícios de arquiteturas paralelas com mínima alteração do código (CHAPMAN, JOST e VAN DER PAS, 2008).

O número de *threads* que uma aplicação OpenMP cria pode ser definido em tempo de execução ou pelo uso de uma variável de ambiente. As aplicações começam executando de forma sequencial até encontrarem uma diretiva `#pragma omp parallel`, nesse momento a *thread* principal torna-se a *thread* mestra de um grupo de *threads* criadas a partir de um *fork* no fluxo de execução (GRAMA, GUPTA, *et al.*, 2003). Na Figura 10 é possível observar melhor esse comportamento. Nos pontos verdes, a diretiva foi encontrada e como o número de *threads* estava definido como sendo 4, então 3 *threads* foram criadas e a principal (na cor azul) também prossegue executando o código da região paralela. Nos pontos vermelhos existem *joins* que são barreiras implícitas nas quais a *thread* principal espera pelo término da execução das demais para prosseguir pela próxima região sequencial.

Vale notar que diretivas `#pragma omp parallel` podem ser aninhadas, criando grupos de *threads* que serão mestres de outros grupos de *threads* (CHAPMAN,

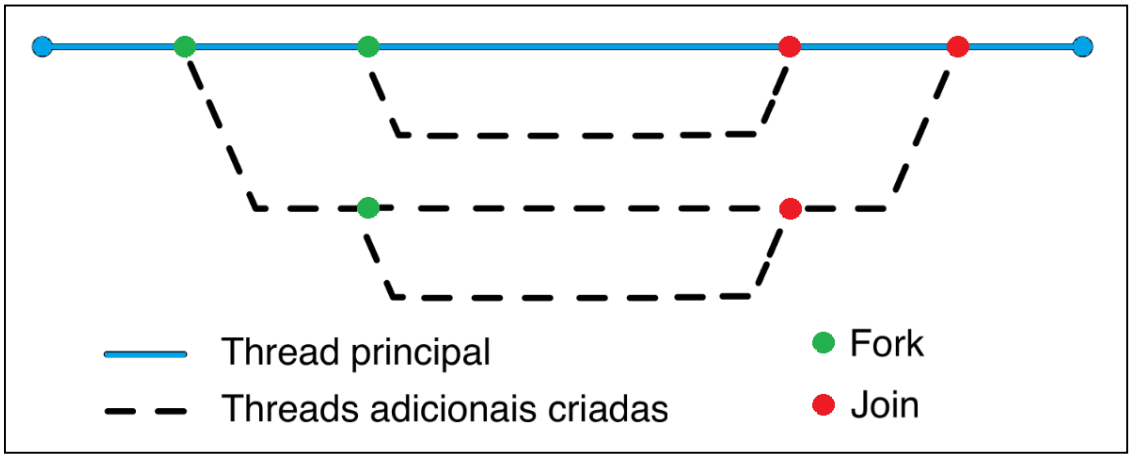


Figura 11 – *forks* aninhados em OpenMP

JOST e VAN DER PAS, 2008). Como por exemplo, na Figura 11 onde tanto a *thread* principal (em azul) quanto sua filha realizaram um novo *fork*.

Além disso, na diretiva `#pragma omp parallel` é possível passar uma lista de cláusulas que acrescentem informações ao compilador com relação à forma como as *threads* deverão ser criadas. Com essas informações, é possível definir quantas *threads* filhas serão criadas no *fork*, quais variáveis devem ser privadas e até definir valores de inicialização para as variáveis.

2.2.3.2 Java Threads

A paralelização de aplicações em Java pode ser alcançada de diferentes formas, é possível criar uma classe que estende a classe `Thread`, criar uma classe que implementa a interface `Runnable` ou ainda abstrair mais o gerenciamento de *threads* e passar essas atividades para um *executor* (ORACLE, 2016). A versão Java do NPB foi implementada a partir da criação de classes que estendem a classe `Thread` e por esse motivo, no restante dessa seção, vamos nos ater a esse método.

Ao criar uma aplicação paralela estendendo a classe `Thread`, é preciso especificar na declaração da classe que ela `extends Thread` e então implementar o método `run` como ocorre na Figura 12 – Criando classe que estende Java `Thread`. O método `run` é um método da classe `Thread` que é sobrecarregado na implementação da classe e é executado no momento que o método `start` do objeto for chamado após a sua construção.

A classe Java `Thread` baseia-se na comunicação por memória compartilhada e está sujeita aos problemas de acesso simultâneo a dados citados na seção 2.2.2.1. Métodos de sincronização de acesso a seções críticas foram implementados para evitar condições de corrida. Para exemplificar, consideremos o problema apresentado na Figura 5, onde

```
01. public class ThreadBase extends Thread
02. {
03.   public void run()
04.   {
05.     //implementar tarefas da thread
06.   }
07. }
```

Figura 12 – Criando classe que estende Java `Thread`

Figura 13 – Utilização de `synchronized` em zona crítica

```
01. public class SynchronizedVarCompart {
02.    private int varCompart = 0;
03.
04.    public synchronized void incrementar(){
05.        varCompart ++;
06.    }
07.
08.    public synchronized void decrementar(){
09.        varCompart --;
10.    }
11.    ...
12.}
```

uma *thread* fazia o incremento e a outra fazia o decremento da variável compartilhada `varCompart`. Na Figura 13, `varCompart` é um atributo privado da classe `SynchronizedVarCompart` e sempre que as *threads* quiserem modificar esse atributo deverão chamar os métodos `incrementar()` e `decrementar()`, métodos que foram modificados pela palavra reservada `synchronized`. Essa palavra reservada tem como um de seus efeitos tornar impossível que dois métodos sincronizados sejam invocados ao mesmo tempo em um mesmo objeto (ORACLE, 2016). Ou seja, se a primeira *thread* está incrementando e a segunda tentar decrementar, ela será bloqueada até que a primeira termine sua tarefa.

Todos os objetos em Java têm um *lock* intrínseco associado a eles e tudo que é derivado da classe `Object` é um objeto, ou seja, tem um *lock* intrínseco que pode ser bloqueado. Em Java não só as classes criadas pelo programador que explicitamente

```
01. public class SynchronizedVarCompart {
02.    private int varCompart = 0;
03.    private int otherVar = 0;
04.
05.    public void incrementar(){
06.        synchronized(varCompart){
07.          varCompart ++;
08.        }
09.    }
10.
11.    public int getOtherVar(){
12.        synchronized(otherVar){
13.          return otherVar;
14.        }
15.    }
```

Figura 14 – Bloco sincronizado em Java

derivam de `Object`, como as classes que derivam de outras classes da biblioteca padrão e os tipos primitivos são derivado da classe `Object`, logo todos podem ser bloqueados. O bloqueio dos *locks* intrínsecos ocorre por meio de blocos sincronizados, como apresentado na Figura 14. Os blocos sincronizados são indicados para situações onde não é necessário bloquear o acesso a todo o objeto, mas apenas a um atributo dele. Dessa forma, uma *thread* pode executar a função `incrementar()` enquanto outra acessa a função `getOtherVar()` e isso é importante para não reduzir o paralelismo, uma vez que as funções são logicamente disjuntas, ou seja, uma não interfere em nada o funcionamento da outra. O bloco sincronizado também pode ser usado para bloquear o objeto como um todo a partir da chamada de `synchronized(this)`, essa chamada tem a mesma função do método sincronizado, mas é útil para reduzir a granularidade da exclusão mútua, evitando assim, reduzir o paralelismo (ORACLE, 2016).

Os objetos da classe `Object` também possuem um monitor, de modo que eles possam usar as funções `wait()`, `notify()` e `notifyAll()`. Quando uma *thread* chama a função `wait()` de um objeto, ela ficará bloqueada até que outra *thread* execute alguma das funções de notificação. Vale ressaltar que as filas de bloqueio dos monitores não são do tipo *first-in-first-out*, sendo aleatória a escolha da *thread* acordada pela função `notify()`.

2.2.3.3 C++11 Threads

O padrão C++11 é o segundo padrão da linguagem, ele levou aproximadamente uma década para ser lançado oficialmente e após tanto tempo de desenvolvimento trouxe evoluções significativas para a linguagem; antes do padrão C++11, não havia suporte à programação paralela na linguagem e na biblioteca padrão (JOSUTTIS, 2012). Em C++, assim como em Java, existe mais de uma forma de construir uma aplicação paralela. Uma das opções é trabalhar com interfaces de mais alto nível, a classe `future` e a *template function* `async`, podemos também utilizar as interfaces de baixo nível, a classe `thread` e a *template class* `promises`. Implementamos a versão C++ do NPB usando a classe `Thread` e por esse motivo, no restante dessa seção, vamos nos ater a esse método.

```
01.  void tarefa();
02.  ...
03.  std::thread novaThread(tarefa);
04.  ...
05.  novaThread.join();
```

Figura 15 – Criando *thread* em C++

Para criar uma *thread* usando a classe `thread`, basta declarar um objeto da classe e passar uma função a ser executada pelo objeto como argumento (JOSUTTIS, 2012). Na Figura 15, um exemplo de uso de *thread*, onde é criado o objeto `novaThread` na linha 3, que recebe como parâmetro a função `tarefa` declarada na linha 1. Na linha 5, a *thread* executa a função `join()` da thread criada, ou seja, vai aguardar o término da sua execução. Essa última linha é importante pois ao final da execução da função principal, todas as *threads* que não terminaram ainda ou não foram destacadas com a função `detach()`, são abortadas abruptamente, não finalizando suas tarefas (JOSUTTIS, 2012).

A classe `thread` de C++ também baseia-se na técnica de comunicação memória compartilhada e sendo assim, ela dispõe de algumas operações importantes para a sincronização das *threads*. Entre elas, a função `yield()` pode ser utilizada para fazer a *thread* atual abrir mão do processador em situações onde ela não deve seguir seu processamento, por exemplo se uma *flag* indicar que uma dependência de dados ainda não foi suprida. No exemplo da Figura 16, a *thread* só deve prosseguir quando outra *thread* modificar o conteúdo da variável `prosseguir` para verdadeiro, então ela acaba sua execução prematuramente, evitando o uso de ciclos de processador sem realizar operações.

```
01.  bool prosseguir = false;
02.  ...
03.  while(!prosseguir)
04.       this_thread.yield();
```

Figura 16 – Função `yield` da classe `thread`

Para auxiliar na sincronização de *threads*, tipos atômicos foram criados no padrão C++11 de modo que os problemas detalhados na seção 2.2.2.1 sejam evitados sem a necessidade de criar uma zona de exclusão mútua para cada parte de código em que variáveis compartilhadas forem modificadas. Um tipo atômico garante que as *threads* não sofrerão interrupções durante leituras ou escritas nas variáveis desse tipo (JOSUTTIS, 2012). Quando a seção crítica for maior do que apenas uma instrução de leitura ou escrita, *mutexes* e *locks* podem ser usados para criar zonas de exclusão mútua, como na Figura 17, onde a função `tarefa_1` e algumas partes da função `tarefa_2` não devem ser executadas concorrentemente e por isso usam *locks* e *mutex* para exclusão mútua. Vale ressaltar que cada uma das funções está usando um tipo diferente de *lock*: a `tarefa_1` usa o `lock_guard` que é um *lock* que é automaticamente liberado em caso de exceção ou ao final do escopo; a `tarefa_2`, o `unique_lock` que é um tipo de *lock* mais flexível que pode ser liberado e travado à medida que for necessário (JOSUTTIS, 2012). Os *locks* sempre são utilizados associados a um *mutex*, quando uma função trava um *mutex* e precisa chamar outra função que trava o mesmo *mutex*, para que não ocorra um *deadlock* deve-se usar um *mutex* do tipo `recursive_mutex`, que é um tipo de *mutex* que pode ser travado mais de uma vez pela mesma *thread* (JOSUTTIS, 2012).

As variáveis de condição, a última ferramenta para auxílio à sincronização de *threads* que abordaremos, visam solucionar o mesmo problema da Figura 16, mas de uma

```
01. std::mutex mutex;
02.
03. void tarefa_1()
04. {
05.     std::lock_guard<mutex> lock(mutex);
06.     ...
07. }
08.
09.void tarefa_2()
10. {
11.     std::unique_lock<mutex> lock(mutex);
12.     ...
13.     lock.unlock();
14.     ...
13.     lock.lock();
14.     ...
15.     lock.unlock();
16.     ...
17. }
```

Figura 17 – Uso de *mutex* e *lock* em C++

Figura 18 – Uso de variáveis de condição em C++

```
01.  bool prosseguir = false;
02.  std::condition_variable prossegueCondVar;
02.  ...
03.  std::unique_lock<std::mutex> lock(prosseguir);
04.
05.  while(!prosseguir)
06.       prossegueCondVar.wait(lock);
```

forma mais eficiente. A Figura 18 apresenta a declaração e o uso de uma variável de condição chamada `prossegueCondVar`. A melhor eficiência dessa técnica, em relação ao uso de `yield()`, reside no fato que a *thread* adormecida só acordará quando outra *thread* modificar o valor da variável `prosseguir` para verdadeiro e então notificar a *thread* adormecida (JOSUTTIS, 2012), enquanto que na técnica do `yield()`, a *thread* nunca ficava adormecida e ela mesma eventualmente verificava se a variável `prosseguir` havia sido modificada, gastando assim tempo de processador e o tempo necessário para a troca de contexto. Vale ressaltar que, como as linhas 5 e 6 do exemplo não são atômicas, esse tipo de construção pode gerar *deadlocks*, que são situações nas quais todas as *threads* estão bloqueadas aguardando uma notificação que pode não ser recebida. Isso ocorre, por exemplo, se a *thread* principal executar a linha 5 e for interrompida pela *thread* de trabalho, que por sua vez vai modificar a variável `prosseguir` e então notificar a principal. Nessa situação, quando a *thread* principal retomar sua execução, ela irá bloquear e aguardar pela notificação que já foi perdida.

3 TRABALHOS RELACIONADOS

Apesar de alguns trabalhos avaliarem o desempenho das linguagens de programação C, C++ e Java, poucos avaliam o desempenho considerando aplicações paralelas. Outro fator pouco explorado é o *speedup* no desempenho de aplicações paralelas Java ocasionado pelo compilador JIT. A seguir mostraremos os trabalhos relacionados e as contribuições desse trabalho.

3.1 Comparação entre Linguagens de Diferentes Paradigmas

Os autores em BULL, SMITH, *et al.* (1999) comparam a performance entre C e Java a partir da execução do *benchmark* Linpack que avalia o quão rápido um computador é capaz de solucionar um sistema denso de equações lineares. Os resultados mostraram que Java foi 2.25 vezes mais lento que C no caso mais significativo. Em outra comparação entre C e Java, o autor em SESTOFT (2014) avaliou o desempenho a partir da execução de quatro aplicações: multiplicação de matrizes, laço intensivo de divisões, avaliação polinomial e uma função de distribuição. Os resultados mostraram que C teve um melhor desempenho que Java. Em BERNARDIN, CHAR e KALTOFEN (1999), uma comparação entre paradigmas de programação é realizada considerando as linguagens de programação C (como uma linguagem procedural), C++ e Java (como linguagens orientadas a objeto). Os resultados mostraram que, em multiplicação polinomial, Java foi 17% mais rápido que o C padrão e 2.6 vezes mais lento que C++.

Considerando tanto desempenho quanto energia dissipada, os autores em CHATZIGEORGIOU e STEPHANIDES (2002) comparam os paradigmas procedural e orientado a objetos. O trabalho investiga o impacto do uso de técnicas de orientação a objetos em C++ em comparação ao estilo de programação procedural na linguagem C em sistemas embarcados. O trabalho usou o *benchmark* OOPACK para fazer a análise e obtiveram resultados que mostram que o tempo de execução e a energia consumida são significativamente impactados pelo uso de orientação a objetos em comparação com o paradigma procedural. Os autores em SARTOR, LORENZON e BECK (2015) avaliaram o impacto da máquina virtual Dalvik nas arquiteturas suportadas da plataforma Android (i.e., ARM, MIPS e x86). Os autores compararam o desempenho de aplicações Java com versões das aplicações nas quais os métodos mais demorados eram implementados em C, sendo compilados para a plataforma alvo, não sendo interpretados pela máquina virtual.

Os resultados mostraram que a máquina virtual apresenta *overheads* diferentes dependendo da arquitetura do processador.

3.2 Comparação entre Linguagens de Mesmo Paradigma

Em GHERARDI, BRUGALI e COMOTTI (2012), os autores quantificam a diferença de desempenho entre C++ e Java em aplicações para robótica. Os resultados mostraram que Java foi de 1.09 até 1.91 vezes mais lento que C++. Apesar de C++ ter uma melhor performance, os autores concluíram que Java oferece benefícios interessantes, como portabilidade, reusabilidade e manutenibilidade. Os tempos de execução de Java e C++ também são comparados em WENTWORTH e LANGAN (2000). Usando os algoritmos de ordenamento Bubble Sort, Insertion Sort, Quick Sort e Heap Sort, os resultados demonstraram que C++ foi de 1.45 até 2.91 vezes mais rápido que Java. Da mesma forma, os autores em ALNASER, ALHEYASAT, *et al.* (2012) mostram que na média, Java precisa de 10% mais tempo que C++ para executar o mesmo segmento de código.

Considerando aplicações *multithread*, os autores em GU, LEE e CAI (1999) avaliam o *speedup* alcançado por meio da classe Thread de Java, na aplicação *Embarassingly Parallel* (EP) do NPB. Os testes foram realizados em duas plataformas *multicore* e os resultados mostraram que o comportamento da aplicação Java é determinado pelo mapeamento das *threads* Java para as *threads* nativas do sistema. Além disso, os autores constataram que o uso de um número excessivamente grande de *threads* deve ser evitado sempre que possível, devido ao *overhead* consequente do gerenciamento das *threads*. Considerando diferentes IPPs para a linguagem C, os autores de LORENZON, CERA e BECK (2016), LORENZON, CERA e BECK (2014) e LORENZON, SARTOR, *et al.* (2015) investigaram o desempenho e o consumo energético quando executando aplicações paralelas em diferentes processadores *multicore* e arquiteturas.

Considerando a execução de aplicações em arquiteturas massivamente paralelas, os autores em GUPTA, AGRAWAL e MAITY (2013) compararam duas versões implementadas em Java rodando em *Graphics Processing Unit* (GPU) e em processador. Os resultados mostraram que a versão rodando em uma GPU tem um *speedup* de seis vezes contra a versão serial rodando em um processador e um *speedup* de duas vezes na

implementação paralela em Java. Da mesma forma, os autores em DOCAMPO, RAMOS, *et al.* (2013) avaliam o desempenho e a produtividade de Java. Como um caso de estudo, eles consideram dois projetos representativos de GPGPU (*General Purpose GPU*): jCuda (YAN, GROSSMAN e SARKAR, 2009), baseada em Cuda e Aparapi (UCHIYAMA, ARAKAWA, *et al.*, 2012), baseada em OpenCL. Para avaliar o desempenho, os autores usaram quatro *kernels* do *benchmark* Scalable Heterogeneous Computing (SHOC): multiplicação de matrizes, *stencil 2D,* transformada rápida de Fourier com ponto flutuante de precisão simples e dupla. Os autores concluíram que enquanto o Aparapi apresenta um *tradeoff* entre produtividade e desempenho, jCuda proporciona melhor desempenho ao custo de maior esforço de programação.

3.3 Avaliação do Impacto do Compilador JIT

Em WANG e BHARGAVA (1999), os autores além de compararem o desempenho das linguagens C e Java, também comparam o desempenho de Java usando o compilador JIT e Java sem usar o JIT na execução de um decodificador de vídeos no formato MPEG. Os resultados mostraram que Java com JIT é, em média, 12 vezes mais lento que C, enquanto que Java sem usar o compilador JIT é, em média, aproximadamente 3 vezes mais lento que Java com JIT.

Em DA SILVA e COSTA (2005), a eficiência do compilador JIT é testada contra versões compiladas das próprias aplicações Java. No experimento, os autores compararam aplicações do *benchmark* Java Grande Forum compiladas com o compilador *GNU Compiler for Java* (GCJ) (GNU TEAM, 2016) com versões puramente interpretadas e versões compiladas dinamicamente com o compilador JIT. Os resultados dos experimentos mostram que a versão interpretada de Java é entre 3 e 11.6 vezes mais lenta que a versão compilada com o GCJ, enquanto que a versão JIT obteve melhor desempenho do que a versão compilada estaticamente para quase todas as aplicações.

3.4 Contribuições Desse Trabalho

Considerando os trabalhos relacionados, observamos que apesar de existirem significativos trabalhos que comparam os paradigmas procedural e orientado a objetos, a maioria deles considera somente o desempenho de aplicações sequenciais. Outro fator

não tão explorado é a diferença de consumo energético entre os paradigmas. O desempenho do compilador JIT no *speedup* de aplicações também carece de estudos em aplicações paralelas e impacto no consumo energético. Sendo assim, esse trabalho busca avaliar o desempenho e consumo de energia das aplicações paralelas nos paradigmas procedural e orientado a objetos, além de avaliar o impacto do compilador JIT nas aplicações paralelas. Para conseguirmos avaliar esses itens, desenvolvemos uma versão na linguagem C++ do conjunto de *benchmarks* NPB.

4 METODOLOGIA

Esse trabalho busca comparar paradigmas e linguagens de programação a partir do seu consumo energético e desempenho. Na seção 4.1 o conjunto de aplicações será apresentado e as particularidades de cada um dos *benchmarks* serão discutidas. Na seção 4.2, as métricas utilizadas na avaliação dos experimentos serão descritas e por fim, na seção 4.3 apresentaremos o ambiente de execução onde os experimentos foram realizados.

4.1 NAS Parallel Benchmark

Desenvolvido originalmente pela divisão de Supercomputação Avançada da NASA, esse conjunto de *benchmarks* foi desenvolvida para auxiliar na avaliação de computadores altamente paralelos (BAILEY, BARSZCZ, *et al.*, 1991). A versão 3.0, considerada nesse trabalho, é composta por 7 *benchmarks* e possui implementações oficiais disponíveis em Fortran (usando OpenMP) e Java. A seguir discutiremos as características dos *benchmarks* que compõe o conjunto de acordo com (BAILEY, BARSZCZ, *et al.*, 1992).

4.1.1 Integer Sort – IS

A partir do ordenamento de uma grande quantidade de pequenos inteiros, esse *benchmark* testa não só o desempenho da computação de inteiros, como também a velocidade de acesso à memória. Esse *kernel* simula aplicações de "partículas em células" onde partículas são associadas a células, mas podem escapar e são recolocadas nas células apropriadas pelo método de ordenamento. Esse é o único problema no qual aritmética de ponto flutuante não está envolvida. Grande quantidade de comunicação de dados, entretanto, é necessário.

4.1.2 Conjugate Gradient – CG

A partir do uso do método Gradiente Conjugado para calcular o menor valor próprio de uma grande matriz positiva e simétrica, esse *benchmark* avalia o desempenho de acesso irregular à memória aplicando multiplicação de vetor por matriz.

4.1.3 Multigrid – MG

O MG é *kernel multigrid* simplificado que avalia principalmente o acesso à memória, considerando transferência de dados. Em contraste com o CG, a comunicação apresenta padrões bastante estruturados.

4.1.4 3D Fast Fourier Transform – FT

Esse *benchmark* faz uma avaliação rigorosa do desempenho de acesso à memória compartilhada a partir da solução de uma equação parcial diferencial de 3 dimensões usando a Transformada Rápida de Fourier. A comunicação é utilizada principalmente para transposições e ocorre em três etapas: local, global e local novamente.

4.1.5 Lower-Upper Gauss-Seidel Solver – LU

Também conhecido como Lower-Upper Diagonal, ele aplica uma relaxação simétrica sucessiva e representa as computações associadas às novas classes de algoritmos de dinâmica de fluídos (*computational fluid dynamics* – CFD) e tem como uma de suas características uma certa limitação no grau de paralelismo.

4.1.6 Block Tridiagonal Solver – BT

Nessa aplicação, múltiplos sistemas independentes de equações bloco-tridiagonais não diagonalmente dominantes de tamanhos de blocos 5x5 são solucionados.

4.1.7 Scalar Pentadiagonal Solver – SP

Nessa aplicação, múltiplos sistemas independentes de equações pentadiagonais escalares não diagonalmente dominantes são solucionados.

Estes dois últimos *benchmarks*, assim como o LU, também são aplicações que simulam computação de dinâmica de fluídos e têm como principal objetivo representar de forma fiel as necessidades de computação de modernas aplicações de CFD. Os *benchmarks* BT e SP apresentam dependência global de dados e são bem parecidos, tendo no raio de comunicação a sua maior diferença.

Tabela 1 – Tamanhos das entradas para as classes de cada *benchmark*

		S	W	A	B	C
IS	Tamanho Vetor (2^n)	16	20	23	25	27
	Valor Máximo (2^n)	11	16	19	21	23
CG	Nº Iterações	15	15	15	75	75
	Tamanho Problema	1400	7000	14000	75000	15000
MG	Nº Iterações	4	40	4	20	20
	Dimensões	32x32x32	64x64x64	256x256x256	256x256x256	512x512x512
FT	Nº Iterações	6	6	6	20	20
	Dimensões	64x64x64	128x128x128	256x256x128	512x256x256	512x512x512
LU	Nº Iterações	50	300	250	250	250
	Dimensões	12x12x12	33x33x33	64x64x64	102x102x102	162x162x162
BT	Nº Iterações	60	200	200	200	200
	Tamanho Problema	12	24	64	102	162
SP	Nº Iterações	100	400	400	400	400
	Tamanho Problema	12	36	64	102	162

A execução do NPB em diferentes arquiteturas, com diferentes capacidades computacionais, depende de entradas de tamanho variável, sendo assim, os *benchmarks* contém cinco entradas padrões de diferentes tamanhos. Na Tabela 1, estão apresentados os tamanhos para cada classe de entradas dos *benchmarks.*

4.2 Métricas

Esse trabalho avalia a execução dos *benchmarks* nas diferentes linguagens e paradigmas com base no seu desempenho (tempo de execução), consumo energético e *energy-delay product* (EDP). O EDP é uma métrica que vem sendo amplamente utilizada para comparar ambientes diferentes como em (BLEM, MENON e SANKARALINGAM, 2013) e (TIWARI, KEIPERT, et al., 2015), uma vez que ela nos possibilita em um único valor, analisar a relação entre energia e desempenho. O EDP é calculado como mostra a Equação 1, onde *Energy* é a quantidade de energia consumida pela aplicação e *Delay* é o tempo total de execução da aplicação.

$$EDP = Energy \times Delay$$

Equação 1 – Cálculo do EDP.

Tabela 2 – Principais características do ambiente de execução

Intel Core i7	
Microarquitetura	Skylake
Frequência	3.4 GHz
# Cores	4
# Threads	8
L1 Cache Dados	4 x 32 KB
L1 Cache Instr.	4 x 32 KB
L2 Cache	4 x 256 KB
L3 Cache	8 MB
Memória RAM	32GB

4.3 Ambiente de Execução

Os experimentos foram conduzidos no processador Inter Core i7 como mostra a Tabela 2. O consumo energético foi obtido a partir da biblioteca *Runnig Average Power Limit* (RAPL) da Intel. A biblioteca RAPL fornece um conjunto de contadores de *hardware* que possibilita medir o consumo energético dos componentes de *hardware* a nível de processador. Os seguintes pacotes foram utilizados: *package*, que fornece o total de energia consumida por todo o conjunto do processador (*core, cache*, barramento, etc.); controlador de memória *DRAM* (ROTEM, NAVEH, *et al.*, 2012). O tempo de execução foi obtido em Java com a função `currentTimeMillis()` da classe `System`, em C++ com a biblioteca `chrono`, padrão da versão C++11 e na linguagem C, o tempo foi obtido com a função `omp_get_wtime` de OpenMP.

Os resultados apresentados na seção a seguir consideram a média de 10 execuções, com um desvio padrão inferior a 1% para cada *benchmark*. Os testes foram executados com os seguintes números de *threads*: 1, 2, 3, 4 e 8; utilizando as classes de entrada conforme a Tabela 3.

Aplicação	Classe
LU	A
BT	
SP	
FT	B
CG	C
IS	
MG	

Tabela 3 – Classes de entradas por aplicação

5 IMPLEMENTAÇÃO

Como mencionado anteriormente, implementamos uma versão em C++ do conjunto de *benchmarks* NAS Parallel Benchmark 3.0 que buscou ser o mais fiel possível à implementação oficial em Java. A seguir discutiremos as estratégias adotadas na implementação, bem como as diferenças para a versão Java e as dificuldades enfrentadas.

5.1 Classe NPBThread

Uma das diferenças entre a implementação de aplicações paralelas em Java e em C++ é a forma como o paralelismo é integrado à aplicação. Como já discutido na seção 2.2.3, uma das formas de criar uma aplicação paralela em Java é estendendo a classe `Thread` padrão de Java. A classe contém um monitor de modo que basta chamar os métodos `wait()` e `notify()` para bloquear ou acorda as *threads*. Além disso, basta sobrecarregar a função `run()` da classe `Thread` para que o comportamento da *thread* seja definido.

Em C++, por outro lado, um objeto da classe `Thread` padrão do C++11 precisa ser criado e uma função deve ser associada a ele. Ou seja, cada classe paralela das aplicações precisa conter um objeto `Thread` e iniciá-lo definindo uma função que ele deve executar. Além disso, cada uma dessas classes precisa conter um objeto `mutex` e um objeto `condition_variable` além de implementar o uso desses objetos para conseguir definir a sincronização entre as *threads*. Considerando que, de um total de 7 aplicações, 4 delas (MG, BT, LU e SP) possuem pelo menos 4 classes paralelas, é evidente a necessidade de diminuir a replicação de código na implementação das aplicações. Tendo esse cenário em vista, a classe `NPBThread` foi criada para unificar esses trechos que a priori ficariam espalhados pelos códigos dessas diversas classes paralelas.

A classe funciona de forma análoga à classe `Thread` de Java, todas as classes paralelas herdarão os atributos e métodos da `NPBThread`, que contém o *mutex* e a variável de condição necessárias, bem como os métodos `wait()`, `wait_for()` e `notify()` que utilizam esses atributos para fazer a sincronização de forma transparente para as aplicações. Além disso, a classe `NPBThread` possui um método virtual `run()`

que deve ser implementado com o comportamento esperado nas classes filhas. O método `run()` é associado ao objeto `Thread` da classe na sua inicialização.

5.2 Classes das Aplicações

Todas as aplicações são compostas de, no mínimo, três classes: uma classe base, uma classe principal e, no mínimo, uma classe paralela (conforme mostrado na Tabela 4). A classe base contém todos os métodos e atributos que são comuns a todas as classes da aplicação. A classe principal tem duas funções: quando executando a versão sequencial da aplicação, a classe principal é executada realizando todo o trabalho. Quando a versão paralela está sendo executada, a classe principal é responsável por criar as *threads* de trabalho, sincronizá-las, definir valores iniciais e verificar se a solução está correta.

As classes paralelas correspondem às seções paralelas das aplicações, ou seja, os trechos de código que podem ser executados concorrentemente sem comprometer a corretude funcional da aplicação. Desse modo, ao executar uma aplicação com *n* classes paralelas solicitando *x threads*, na realidade serão criadas *x*n threads*, mas nunca mais do que *x threads* executarão simultaneamente. Por exemplo, ao executar a aplicação SP – que tem 6 classes paralelas, vide Tabela 4 – com 2 *threads*, serão criadas 12 *threads*, mas no máximo 2 *threads* executarão simultaneamente. A seguir, vamos discutir brevemente as classes paralelas das aplicações.

a) CG – Essa aplicação possui apenas uma classe paralela chamada `CGWorker`. Nela, os trabalhos realizados pelas *threads* estão divididos em cinco estados e não possuem dependência de dados com outras *threads*. Nessa aplicação, as funções

Aplicação	Número de Classes Paralelas
CG	1
IS	1
FT	2
MG	4
BT	5
LU	5
SP	6

Tabela 4 – Número de classes paralelas por aplicação

`wait` foram substituídas por `wait_for`, que contém um *time-out* que acorda a *thread* principal após um certo tempo. Essa medida foi tomada devido a *deadlocks* que ocorriam devido à não atomicidade da verificação e bloqueio, como foi comentado na seção 2.2.3.3.

b) IS – O *Integer Sort* também possui apenas uma classe paralela chamada `RankThread`. O trabalho realizado pelas *threads* está dividido em 2 passos. No primeiro passo, elas contam quantas vezes cada valor aparece no seu respectivo trecho do vetor original. No segundo passo, cada *thread* é responsável por atualizar uma parte diferente do vetor de posições da *thread* principal com o somatório de aparições de cada chave calculado pelas *threads* de trabalho.

c) FT – Duas classes paralelas foram criadas para essa aplicação: `FTThread` e `EvolveThread`. Ambas operam diretamente na matriz da *thread* principal, mas cada *thread* opera sobre conjuntos de linhas diferentes, não havendo a necessidade de sincronização entre as *threads* de uma mesma classe. Ocorre sincronização apenas entre as etapas dos cálculos, por exemplo, antes das *threads* da classe `FTThread` começarem as suas execuções, as da classe `EvolveThread` devem ter finalizado as suas.

d) MG – Para essa aplicação, foram desenvolvidas quatro classes paralelas que têm acesso direto ao vetor da classe principal. O acesso ao vetor da classe principal é para leitura e escrita, mas cada *thread* executando paralelamente acessa uma região diferente das demais. As classes `Inter`, `Rprj` e `Resid`, fazem leitura do vetor original e salvam ele em vetores locais e as classes `Inter`, `Rprj` e `Psinv` fazem escrita direto no vetor original (memória compartilhada). Existe sincronização para criar barreiras lógicas que impedem o início de uma etapa antes que a anterior termine de ser executada.

e) LU – Conforme mencionado na seção 4.1, a LU é uma aplicação que tem um paralelismo limitado, uma vez que das 5 classes paralelas, 2 (classes `LowerJac` e `UpperJac`) funcionam de forma semelhante a um *pipeline*. Essa característica pode ser observada nas Figura 19, onde o paralelismo só é totalmente explorado no instante de tempo 4, quando todas as *threads* estão executando alguma tarefa simultaneamente. Nos demais instantes de tempo, algumas *threads* ficam bloqueadas esperando que suas vizinhas lhes notifiquem que alguma tarefa foi

passada para que comecem a executar. Nessas classes, a *thread* principal notifica a primeira *thread* de que ela deve executar e então ela avisa sua vizinha na hora em que ela deve executar, e ao final da zona paralela, a última *thread* a executar notifica a *thread* principal de que pode seguir a execução.

f) BT e SP – Conforme mencionado na seção 4.1, essas aplicações são semelhantes. Ambas têm classes paralelas que fazem acessos aos vetores da classe principal, seja para leitura ou escrita. As aplicações diferenciam-se no fato de que a BT realiza mais acessos à memória compartilhada. Essas aplicações são duas das que possuem mais classes paralelas, ou seja, possuem mais zonas paralelizáveis. E nelas, novamente, a classe principal tem como função sincronizar a execução dessas classes paralelas de modo que a execução de uma zona paralela não sobreponha a execução da anterior, o que causaria resultados errados.

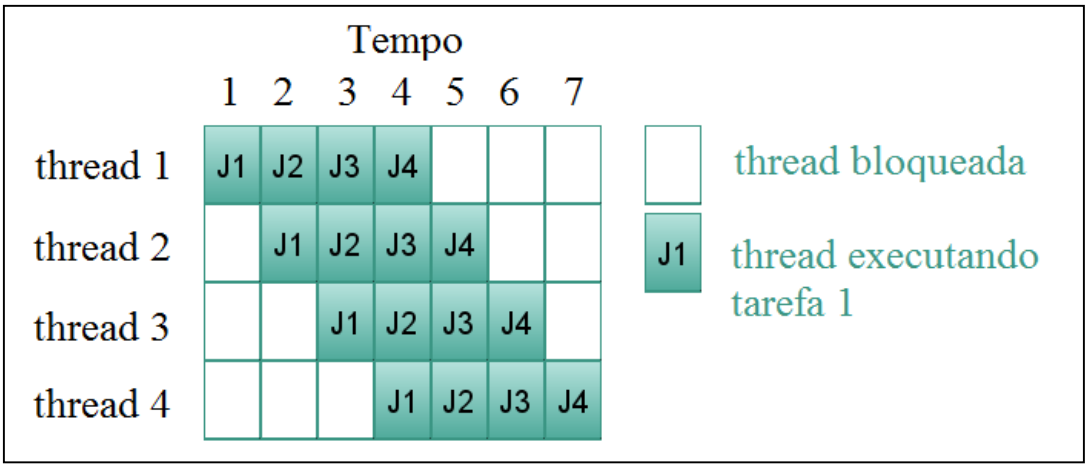


Figura 19 – Paralelismo nas classes `LowerJac` e `UpperJac` da aplicação LU

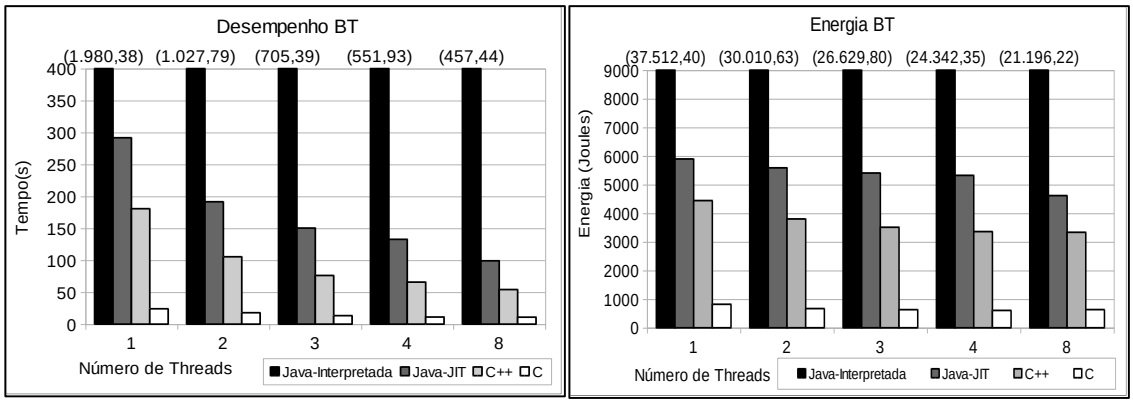
6 RESULTADOS

Nessa seção serão apresentados os resultados obtidos da execução das aplicações no ambiente de execução apresentado na seção 4.3. Nos gráficos, serão apresentados os tempos de execução (segundos) e energia consumida (Joules) das aplicações em C, C++, Java sem JIT (Java-Interpretada) e Java com JIT (Java-JIT). Na seção 6.1 apresentaremos os resultados de desempenho e consumo energético e na seção 0, o EDP.

6.1 Desempenho e Consumo Energético

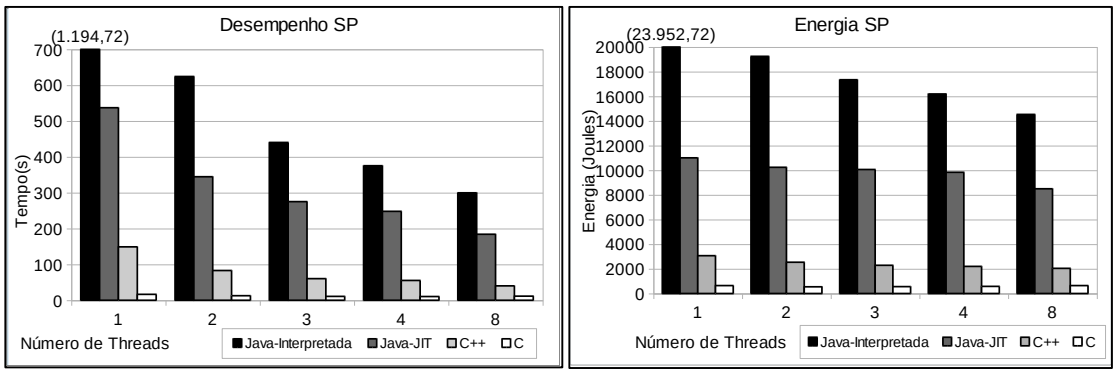
Nos gráficos que apresentam os resultados, as versões executadas (C, C++, Java-JIT e Java-Interpretada) estão distribuídas no eixo x agrupadas por número de *threads* executadas. Nos gráficos de desempenho, o eixo y representa o tempo de execução em segundos e nos de consumo energético, o eixo y representa a quantidade de energia gasta em Joules.

Nas Figura 20 e Figura 21, são apresentados os resultados de desempenho e energia obtidos das aplicações BT e SP. Nelas pode-se observar que o uso do compilador JIT possibilitou em média *speedups* de aproximadamente 8 e 2.4 vezes, respectivamente. Além disso, C teve o melhor desempenho entre as linguagens consideradas, porém tendo uma escalabilidade limitada. Entende-se por escalabilidade, a característica de diminuir o tempo de execução conforme o número de *threads* executando em paralelo aumenta. Por exemplo, na aplicação SP (Figura Figura 21a) executando com 2 *threads*, o *speedup* (a melhoria de desempenho com relação a sua versão sequencial) em C foi de 1.29 e com 8 *threads* foi de apenas 1.43. Nessa mesma aplicação, a versão C++ foi em média 4.27 vezes mais rápida que a Java com JIT, enquanto na aplicação BT (Figura 20a) foi em média 1.79 vezes mais rápida.



(a) (b)

Figura 20 – Desempenho e Energia BT

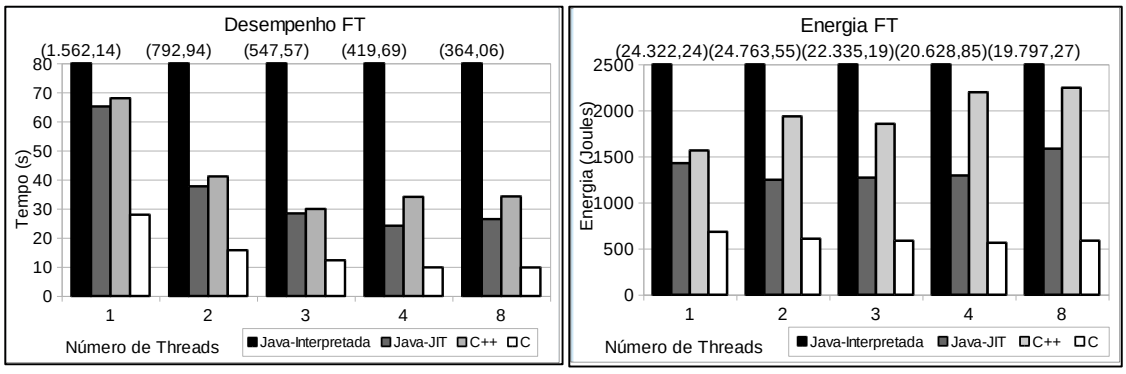


(a) (b)

Figura 21 – Desempenho e Energia SP

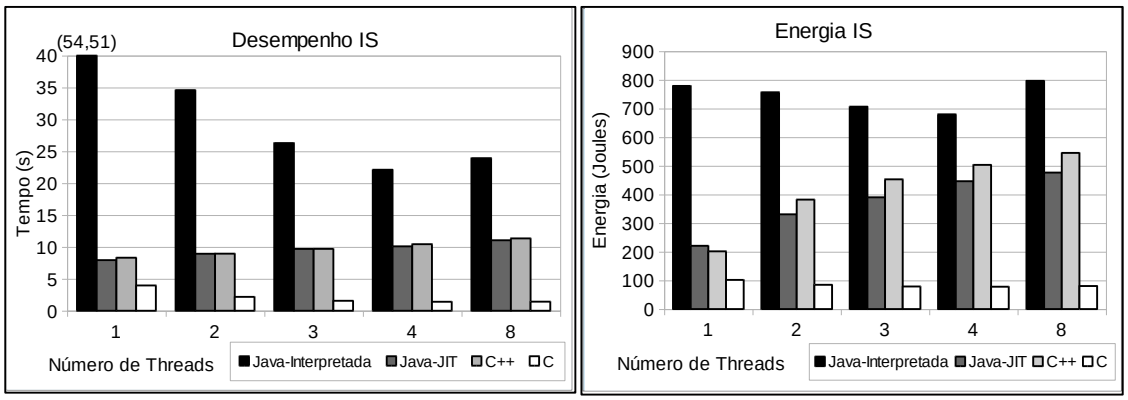
Com relação ao consumo energético das aplicações BT e SP, pode-se notar que a escalabilidade observada nos desempenhos, não se repete no consumo de energia nas versões C, C++ e Java-JIT. Por exemplo, na versão C++ da aplicação BT (Figura 20b), ao compararmos a versão sequencial (1 thread) com a execução com 8 *threads*, enquanto o tempo diminui aproximadamente 3.33 vezes, a energia consumida reduziu apenas 1.33 vezes. Na versão Java-JIT da mesma aplicação, foi observado 2.94 de *speedup* contra apenas 1.27 vezes menos energia consumida. Esse comportamento está ligado ao fato que as aplicações BT e SP estão entre as 3 aplicações que possuem mais classes paralelas e elas realizam muitas leituras de vetores da *thread* principal. O aumento no número de *threads* implica num aumento do número de acessos à memória que é uma operação custosa do ponto de vista energético.

Nas Figura 22 e Figura 23 são apresentados os tempos de execução e os consumos energéticos das aplicações FT e IS. A aplicação FT foi a que apresentou o segundo maior *speedup* com o uso do compilador JIT em relação à execução de Java interpretada, aproximadamente 19 vezes. Essa eficiência do JIT deve-se ao fato de os trabalhos das



(a) (b)

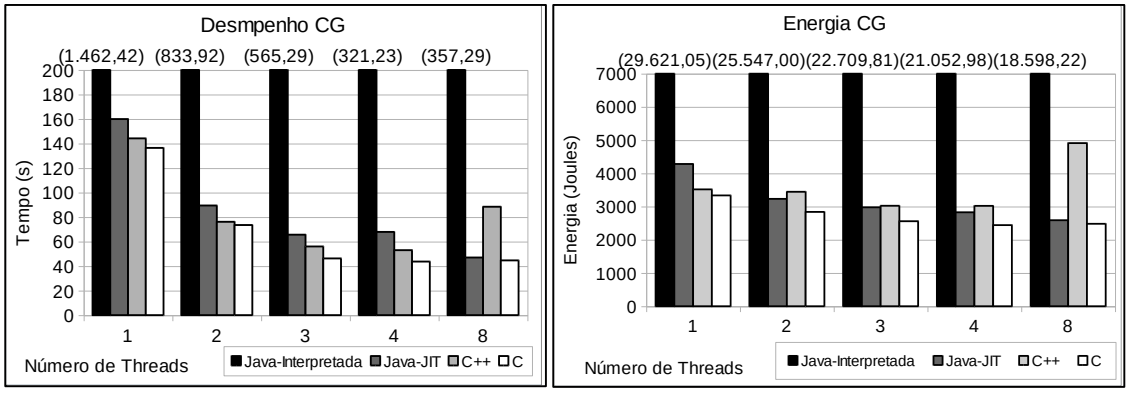
Figura 22 – Desempenho e Energia FT



(a) (b)

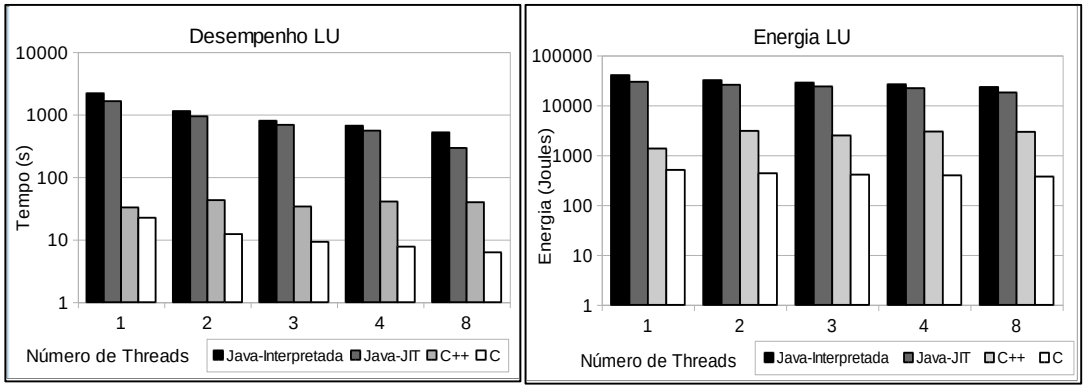
Figura 23 – Desempenho e Energia IS

threads paralelas serem basicamente poucos *loops*, sendo bastante fácil para o compilador achar *hotspots*, ou seja, pontos cuja execução se repete várias vezes. É explorando esses *hotspots* que o JIT obtem seu melhor desempenho. Além disso, nas aplicações FT e IS, as versões Java-JIT, C++ e C tiveram uma escalabilidade bastante limitada, chegando a aumentar o tempo de execução em C++ e Java-JIT com o aumento do número de *threads*. Esse comportamento nas aplicações FT e IS já era esperado, uma vez que a quantidade de trabalho realizado por cada *thread* nessas aplicações é bem menor em comparação com as demais aplicações. Devido a essa pequena quantidade de trabalho, o *overhead* ocasionado pelo gerenciamento das múltiplas *threads* acaba superando o ganho com o paralelismo. A pouca escalabilidade também fica evidente nos consumos energéticos,



(a) (b)

Figura 24 – Desempenho e Energia CG

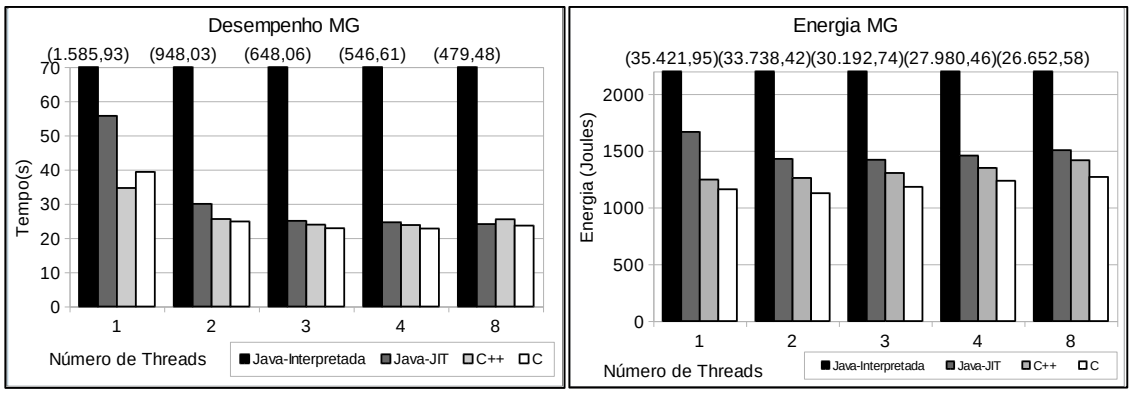


(a) (b)

Figura 25 – Desempenho e Energia LU

como pode ser observado nas Figura 22a e Figura 23a que apresentam os resultados do consumo energético das aplicações FT e IS, respectivamente. Nelas, o consumo de energia aumenta numa proporção maior que o aumento no tempo de execução.

As Figura 24a) e Figura 24b) apresentam os resultados de desempenho e consumo energético da aplicação CG. Essa aplicação teve um comportamento um pouco distinto das anteriores. As quatro versões (C, C++, Java-JIT e Java-Interpretada) tiveram redução do tempo de execução com o aumento do número de *threads*. A única exceção foi a execução com 8 *threads* da versão C++, a qual perdeu desempenho e consumiu mais energia que a execução com 4 *threads*. Esse comportamento está ligado à forma como a versão C++ foi desenvolvida, pois a CG é uma aplicação com uma região paralela pequena. Às vezes a execução muito rápida de algumas *threads* ocasionava *deadlokcks* devido a perdas de sinalizações pela *thread* principal, ou seja, antes da principal executar o `wait()`, a *thread* de trabalho envia a notificação (conforme mencionado na seção 2.2.3.3). Para contornar esse problema, a função `wait()` foi substituída por



(a) (b)

Figura 26 – Desempenho e Energia MG

`wait_for(1)` que insere um *time-out* para acordar a *thread* principal após 1 segundo. Esse *time-out* implica em acordar a *thread* mesmo nos passos mais longos do que 1 segundo, quando a *thread* não precisaria ser desbloqueada ainda, levando a gasto de tempo e energia com troca de contexto com uma *thread* que ainda não deveria executar.

Como podemos observar na Figura 25 (com eixo y em escala logarítmica), na aplicação LU o compilador JIT teve um desempenho bem inferior ao seu desempenho nas aplicações anteriores. O *speedup* proporcionado pelo JIT foi prejudicado pela característica das classes paralelas da aplicação (conforme apresentado na seção 5.2). O comportamento das classes paralelas `LowerJac` e `UpperJac` em forma de *pipeline* faz com que a execução do JIT seja desperdiçada pois não é possível encontrar *hotspots* de forma eficiente. Desse modo, o JIT acaba concorrendo por recursos com a aplicação para realizar uma compilação que não vai conseguir diminuir o seu tempo de execução. Sendo assim, nessa situação, o JIT além de não contribuir, ele também concorre por recursos com a execução da aplicação.

A aplicação MG destaca-se por ser aquela na qual o JIT foi mais eficaz, alcançando 25.6 de *speedup* médio. Na MG, assim como na FT, as *threads* de trabalho executam basicamente *loops* curtos, sendo esses *hotspots* a origem da eficácia do JIT. Os gráficos de desempenho e gasto energético da MG podem ser vistos na Figura 26. A aplicação apresenta uma boa escalabilidade na versão Java-Interpretada, mas nas demais versões a escalabilidade é limitada, ficando praticamente sem ganho de desempenho a partir da execução com 3 *threads.*

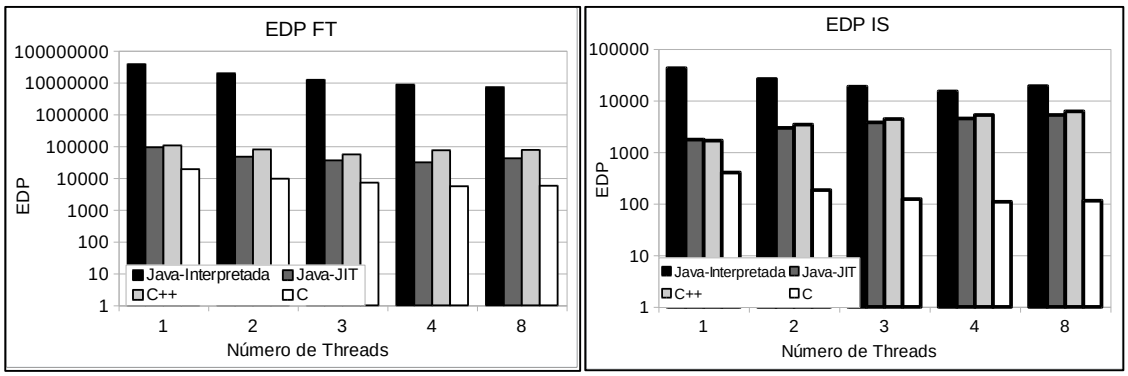
6.2 *Energy-Delay Product* (EDP)

O *energy-delay product* é uma métrica muito importante pois possibilita a comparação entre as execuções de diferentes versões das aplicações a partir de duas métricas (tempo de execução e consumo de energia) considerando apenas um valor. Dessa forma, os gráficos na Figura 27, Figura 28 e Figura 29 foram obtidos a partir da Equação 1 (apresentada na seção 4.2), e estão representados com o eixo Y em escala logarítmica para melhor visualização dos dados.

A partir da análise do EDP das aplicações, notamos que em todas as aplicações a versão C teve melhor desempenho e menor consumo energético. As aplicações CG e MG

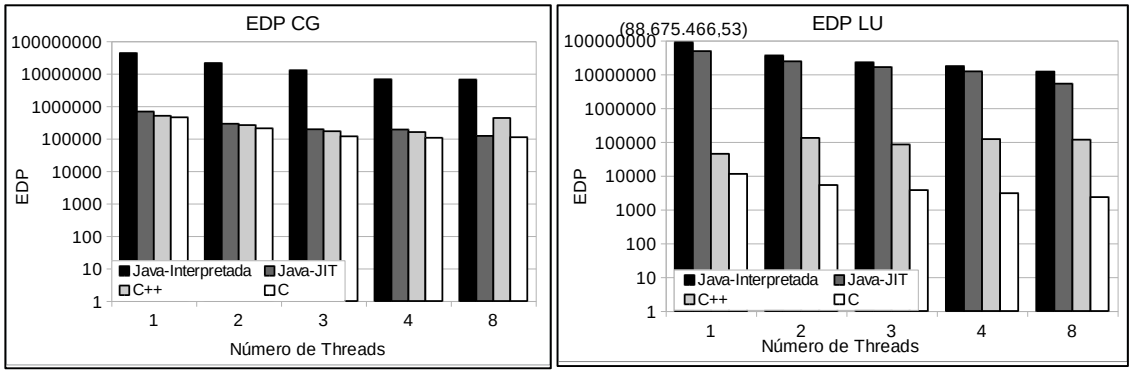
foram as únicas nas quais o valor do EDP de C ficou próximo dos valores para C++ e Java-JIT. Com base nos resultados encontrados considerando as linguagens escolhidas, podemos avaliar que, para o conjunto de *benchmarks* utilizado, a linguagem procedural (C) teve melhor escalabilidade em relação às orientadas a objetos. No melhor caso, IS com 8 *threads*, o EDP de C ficou 54.14 vezes menor que o de C++. Comparando com Java-JIT, o melhor caso, LU com 2 *threads*, C teve o EDP 4552.31 vezes menor. Além disso, entre as linguagens orientadas a objeto, C++ mostra-se uma linguagem que tem melhor desempenho e menor consumo energético que Java-JIT de maneira geral, sendo FT e IS as únicas aplicações nas quais Java-JIT obteve melhor EDP.

Também fica bastante evidente o grande impacto positivo causado pelo uso do compilador JIT no desempenho e consumo energético das aplicações desenvolvidas em Java. O compilador dinâmico consegue entregar uma otimização muito significativa. A única aplicação na qual sua otimização não teve o efeito esperado é a LU, onde o EDP de Java-JIT é, no melhor caso (execução com 8 *threads*), apenas 2.27 vezes menor que o de Java-Interpretado e 45.43 vezes maior que o de C++. O *overhead* da máquina virtual (que agrega portabilidade ao código) e das facilidades de Java, como o *Garbage Collector* (que simplifica o controle de memória alocada) por exemplo, acaba sendo compensado pelo excelente desempenho do compilador dinâmico. Essa grande eficiência do JIT faz com que, dependendo das características da aplicação, Java torne-se uma opção ao C++ devido à pequena diferença de consumo energético e desempenho entre as linguagens e as muitas facilidades de programação que a linguagem possui. Por exemplo, na aplicação IS, Java-JIT foi em média 1.02 vezes mais rápida que C++ e na aplicação FT, foi em média 1.17 vezes mais rápida.

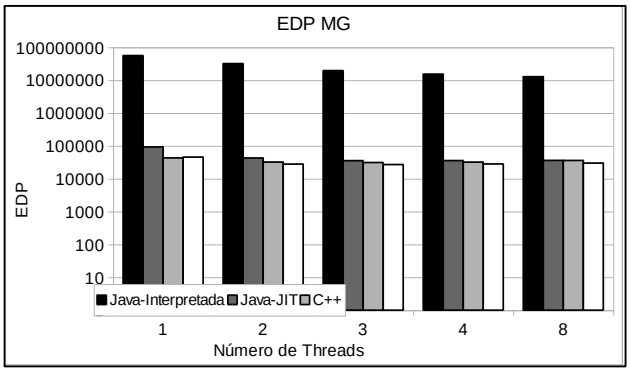


(a) (b)

Figura 27 – EDP FT e IS

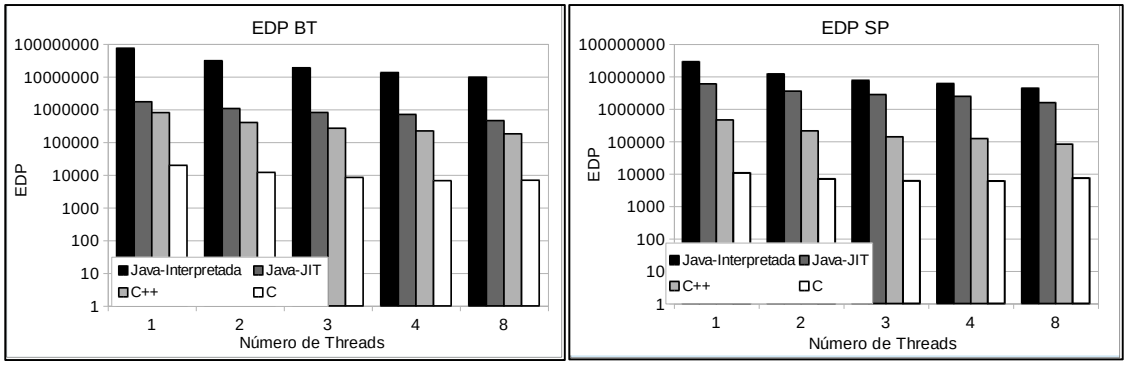


(b) (b)



(c)

Figura 28 – EDP CG, LU e MG



(b) (b)

Figura 29 – EDP BT e SP

7 CONCLUSÃO E TRABALHOS FUTUROS

Esse trabalho analisou o impacto de diferentes paradigmas (procedural e orientado a objetos), linguagens de programação (C, C++ e Java) e compilação dinâmica (JIT) em aplicações paralelas considerando diferentes métricas: desempenho, energia consumida e EDP. Nesse trabalho confirmamos que a linguagem procedural C tem melhor desempenho e consumo energético que as linguagens orientadas a objeto (Java e C++).

Em muitos casos os desempenhos de Java e C++ é muito semelhante, mas dependendo do comportamento da aplicação, C++ tem melhor desempenho. Por exemplo, em aplicações paralelas de comportamento semelhante a um *pipeline*, o compilador JIT não consegue equiparar o desempenho de uma aplicação compilada estaticamente. Em compensação, em aplicações com regiões paralelas muito pequenas, como nas aplicações IS e FT, o *overhead* de gerenciar as muitas *threads* e operações de comunicação em C++ foi maior que em Java. Se considerarmos as facilidades de programação agregadas a Java e o seu bom desempenho médio em comparação à linguagem C++, podemos considerá-la uma boa opção.

Como trabalhos futuros, para obtermos resultados mais conclusivos, vamos considerar outros fatores, tais como paralelismo a nível de instrução e aplicações mais orientadas a controle ou a fluxo de dados. Também vamos considerar o impacto de diferentes compiladores (por exemplo, GCC e LLVM – *Low Level Virtual Machine*) com diferentes níveis de otimização.

REFERÊNCIAS

ALNASER, A. M. et al. Time Comparing between Java and C++ Software. **Journal of Software Engineering and Applications**, v. 5, n. 8, p. 630-633, 2012.

BAILEY, D. et al. **The NAS Parallel Benchmark**. NASA Ames Research Center. [S.l.]. 1991. (Report RNR-91-002 revision 2).

BAILEY, D. et al. **NAS Parallel Benchmark Results**. NAS Applied Research Branch. Moffett Field. 1992.

BARROSO, L. A.; HÖLZLE, U. The case for energy-proportional computing. **IEEE Computer, vol. 40**, 2007.

BERNARDIN, L.; CHAR, B.; KALTOFEN, E. Symbolic Computation in Java: An Appraisement. **Proceedings of the 1999 International Symposium on Symbolic and Algebraic Computation**, p. 237-244, 1999.

BLEM, E.; MENON, J.; SANKARALINGAM, K. Power struggles: Revisiting the RISC vs. CISC debate on contemporary ARM and x86 architectures. **Int. Symp. High-Performance Comput. Archit.**, 2013. 1–12.

BULL, J. M. et al. A Methodology for Benchmarking Java Grande Applications. **Proceedings of the ACM 1999 Conference on Java Grande**, p. 81-88, 1999.

CHAPMAN, B.; JOST, G.; VAN DER PAS, R. **Using OpenMP:** Portable Shared Memory Parallel Programming. Cambridge: MIT Press, 2008.

CHATZIGEORGIOU, A.; STEPHANIDES, G. Evaluating Performance and Power of Object-Oriented Vs. Procedural Programming in Embedded Processors. **Proceedings of the 7th Ada-Europe International Conference on Reliable Software Technologies**, p. 65-75, 2002.

DA SILVA, A. F.; COSTA, V. S. An Experimental Evaluation of JAVA JIT Technology. **Journal of Universal Computer Science**, v. 11, n. 7, Julho 2005.

DOCAMPO, J. et al. Evaluation of Java for general purpose GPU computing. **2013 27th International Conference on Advanced Information Networking and Applications Workshops (WAINA)**, p. 1398–1404, 2013.

EDITORA MELHORAMENTOS. **Dicionário Michaelis**, 2015. Disponivel em: <http://michaelis.uol.com.br/busca?r=0&f=0&t=0&palavra=paradigma>. Acesso em: 27 out. 2016.

FLOYD, R. W. The paradigms of programming. **Communications of the ACM**, v. 22, 1979.

FRANK, D. J. Power-constrained CMOS scaling limits. **IBM Journal of Research and Development**, v. 46, p. 235-244, Março 2002.

GABBRIELLI, M.; MARTINI, S. **Programming Languages:** Principles and Paradigms. [S.l.]: Springer, 2010.

GANDHEWAR, N.; SHEIKH, R. Google Android: An emerging software platform for mobile devices. **International Journal on Computer Science and Engineering**, v. 1, p. 12-17, 2010.

GE, Z.; LIM, H. B.; WONG, W. F. Memory Hierarchy Hardware-Software Co-design in Embedded Systems, 2005.

GEPNER, P.; KOWALIK, M. F. Multi-core Processors: New Way to Achieve High System Performance. **International Symposium on Parallel Computing in Electrical Engineering (PARELEC'06)**, Bialystok, p. 9–13, 2006.

GHERARDI, L.; BRUGALI, D.; COMOTTI, D. **A Java vs. C++ Performance Evaluation:** A 3D Modeling Benchmark. Simulation, Modeling, and Programming for Autonomous Robots: Third International Conference, SIMPAR 2012. Tsukuba, Japan: Springer Berlin Heidelberg. November 2012. p. 161-172.

GNU TEAM. GNU Compiler for Java. **GCC, the GNU Compiler Collection**, 2016. Disponivel em: <https://gcc.gnu.org/java/>. Acesso em: 13 Novembro 2016.

GOOGLE. Getting Started with NDK. **Android Developers**, 2016. Disponivel em: <https://developer.android.com/ndk/guides/index.html>. Acesso em: 20 out. 2016.

GRAMA, A. et al. **Introduction to Parallel Computing**. 2ª. ed. [S.l.]: Pearson Addison Wesley, 2003.

GU, Y.; LEE, B. S.; CAI, W. Evaluation of Java thread performance on two different multithreaded kernels. **Oper. Syst. Rev.**, v. 33, n. 1, p. 34–46, Janeiro 1999.

GUPTA, K. G.; AGRAWAL, N.; MAITY, S. K. Performance analysis between aparapi (a parallel API) and JAVA by implementing sobel edge detection Algorithm. **2013 National Conference on Parallel Computing Technologies (PARCOMPTECH)**, p. 1-5, 2013.

HENNESSY, J. L.; PATTERSON, D. A. **Computer Architecture - A Quantitative Approach**. 4ª. ed. [S.l.]: Morgan Kaufmann, 2007.

JOSUTTIS, N. M. **The C++ Standard Library:** A Tutorial and Reference. 2ª. ed. [S.l.]: Addison-Wesley, 2012.

KAHATE, A. **Object Oriented Analysis and Design**. [S.l.]: Tata McGraw-Hill Education, 2004.

KAISLER, S. H. **Software Paradigms**. New Jersey: Wiley, 2005.

KIM, et al. SnuCL: An OpenCL Framework for Heterogeneous CPU/GPU Clusters. **Proceedings of the 26th ACM International Conference on Supercomputing**, 2012. 341–352.

LORENZON, A. F. et al. Optimized Use of Parallel Programming Interfaces in Multithreaded Embedded Architectures. **2015 IEEE Computer Society Annual Symposium on VLSI**, p. 410–415, 2015.

LORENZON, A. F.; CERA, M. C.; BECK, A. C. S. Performance and Energy Evaluation of Different Multi-Threading Interfaces in Embedded and General Purpose Systems. **J. Signal Process. Syst.**, p. 295-307, 2014.

LORENZON, A. F.; CERA, M. C.; BECK, A. C. S. Investigating different general-purpose and embedded multicores to achieve optimal trade-offs between performance and energy. **J. Parallel Distrib. Comput.**, v. 95, p. 107–123, 2016.

MOORE, G. E. Cramming more components onto integrated circuits. **Eletronics. Solid-State Circuits Society Newsletter**, 2006.

NYGAARD, K.; DAHL, O. J. **History of Programming Languages**. NY: Ed. New York, v. I, 1981.

ORACLE. Concurrency. **The Java Tutorials**, 2016. Disponivel em: <http://docs.oracle.com/javase/tutorial/essential/concurrency>. Acesso em: 08 nov. 2016.

RITCHIE, D. M. The Development of the C Language. **Second History of Programming Languages Conference**, Abril 1993.

ROTEM, E. et al. Power-management architecture of the intel microarchitecture code-named sandy bridge. **IEEE Micro**, v. 2, p. 20-27, 2012.

SARTOR, A. L.; LORENZON, A. F.; BECK, A. C. S. The Impact of Virtual Machines on Embedded Systems. **2015 IEEE 39th Annual Computer Software and Applications Conference**, v. II, p. 626–631, 2015.

SESTOFT, P. Numeric performance in C, C# and Java, 2014.

STROUSTRUP, B. A History of C++: 1979−1991. **ACM SIG-PLAN Notices**, v. 28, n. 3, p. 271-297, Março 1993.

TIOBE SOFTWARE BV. TIOBE Index. **TIOBE The Software Quality Company**, 2016. Disponivel em: <http://www.tiobe.com/tiobe-index/>. Acesso em: 2 nov. 2016.

TIWARI, A. et al. Performance and energy efficiency analysis of 64-bit ARM using GAMESS. **Proceedings of the 2nd International Workshop on Hardware-Software Co-Design for High Performance Computing**, 2015. 1-10.

UCHIYAMA, K. et al. Heterogeneous Multicore Processor Technologies for Embedded Systems, p. 11-19, 2012.

VAN ROY, P.; HARIDI, S. **Concepts, Techniques, and Models of Computer Programming**. 2. ed. [S.l.]: MIT Press, 2004.

VON NEUMANN, J. **First Draft of a Report on the EDVAC**. Moore School of Electrical Engineering, University of Pennsylvania. Philadelphia. 1945.

WANG, S.-Y.; BHARGAVA, B. Experimental Evaluation of Design Tradeoff in Specialized Virtual Machine for Multimedia Traffic in Active Networks. **Computer Science Technical Reports**, Dezembro 1999.

WENTWORTH, S.; LANGAN, D. D. **Performance Evaluation:** Java vs

WILKINSON, B.; ALLEN, M. **Parallel Programming:** Techniques and Applications Using Networked Workstations and Parallel Computers. 2ª. ed. New Jersey: Prentice-Hall, 1999.

YAN, Y.; GROSSMAN, M.; SARKAR, V. JCUDA: A Programmer-Friendly Interface for Accelerating Java Programs with CUDA. **Proceedings of the 15th International Euro-Par Conference on Parallel Processing**, p. 887–899, 2009.